Foundations and Trends® in Electronic Design Automation
Vol. 8, No. 4-3 (2014) 199–356
© 2014 S. Chattopadhyay, A. Roychoudhury, J. Rosén, P. Eles, Z. Peng
DOI: 10.1561/1000000037

Time-Predictable Embedded Software on Multi-Core Platforms: Analysis and Optimization

Sudipta Chattopadhyay
Linköping University
sudipta.chattopadhyay@liu.se

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

Jakob Rosén
Linköping University
jakob.rosen@gmail.com

Petru Eles
Linköping University
petru.eles@liu.se

Zebo Peng
Linköping University
zebo.peng@liu.se

Contents

Abstract **199**

1 Introduction **200**

2 WCET analysis and multi-core platforms **204**
2.1 A background on WCET analysis 204
2.2 Challenges in WCET analysis for multi-core architectures 214

3 WCET analysis for multi-core platforms **217**
3.1 Modeling shared caches 218
3.2 Modeling shared buses 236
3.3 Modeling timing interactions 259
3.4 Discussion about analysis complexity 284
3.5 Experimental evaluation 287
3.6 Data caches and branch target buffers 301
3.7 A survey of related techniques 303

4 WCET optimization for multi-core platforms **305**
4.1 Optimization of worst-case response time 305
4.2 WCRT optimization approach 306
4.3 Cost function 308
4.4 Optimization algorithm 310

4.5 Simplified algorithm 319
4.6 Memory consumption 320
4.7 Experimental results 321
4.8 A survey of related techniques 327

5 Time-predictable multi-core architecture 330
5.1 Resource isolation 330
5.2 Usage of software controlled memory 333
5.3 Extension of instruction set architecture (ISA) 336

6 Discussion and future work 338
6.1 Summary of recent development 338
6.2 Limitations imposed by current approaches 339
6.3 Other limitations 340
6.4 Analysis pessimism 341
6.5 Research challenges in future 342

7 Conclusions 346

Acknowledgements 347

References 348

Abstract

Multi-core architectures have recently gained popularity due to their high-performance and low-power characteristics. Most of the modern desktop systems are now equipped with multi-core processors. Despite the wide-spread adaptation of multi-core processors in desktop systems, using such processors in embedded systems still poses several challenges. Embedded systems are often constrained by several extra-functional aspects, such as time. Therefore, providing guarantees for time-predictable execution is one of the key requirements for embedded system designers. Multi-core processors adversely affect the time-predictability due to the presence of shared resources, such as shared caches and shared buses. In this contribution, we shall first discuss the challenges imposed by multi-core architectures in designing time-predictable embedded systems. Subsequently, we shall describe, in details, a comprehensive solution to guarantee time-predictable execution on multi-core platforms. Besides, we shall also perform a discussion of different techniques to provide an overview of the state-of-the-art solutions in this topic. Through this work, we aim to provide a solid background on recent trends of research towards achieving time-predictability on multi-cores. Besides, we also highlight the limitations of the *state-of-the-art* and discuss future research opportunities and challenges to accomplish time-predictable execution on multi-core platforms.

S. Chattopadhyay, A. Roychoudhury, J. Rosén, P. Eles, Z. Peng. *Time-Predictable Embedded Software on Multi-Core Platforms:*
Analysis and Optimization. Foundations and Trends® in Electronic Design Automation, vol. 8, no. 4-3, pp. 199–356, 2014.

DOI: 10.1561/1000000037.

1

Introduction

Real-time, embedded systems often need to satisfy several extra-functional constraints, such as timing. In particular, for hard real-time systems, such timing constraints are strictly enforced. Violation of these timing constraints may have serious consequences, potentially costing human lives. Therefore, static timing-analysis of hard real-time systems has emerged to be a critical problem to solve.

In general, a real-time, embedded application is made of several components, usually called *tasks*. Therefore, timing analysis of embedded software is typically performed in two separate phases: *(i)* a low-level analysis which derives the *worst case execution time* (WCET) and *best case execution time* (BCET) of individual tasks, and *(ii)* a system-level schedulability analysis which uses the WCET/BCET derived for each task and computes the overall timing characteristics of the application. In this monograph, we shall primarily focus our discussion on low-level WCET analysis.

WCET analysis of an embedded software is typically performed in three stages: *(i)* a flow-analysis using the control flow graph (CFG) of the program (to determine infeasible paths and loop bounds), *(ii)* micro-architectural modeling (to determine the worst case execution time of each basic block in the CFG) and *(iii)* a calculation phase which combines the outcome of

flow-analysis and micro-architectural modeling to derive the worst case execution time (WCET) of the entire program. Micro-architectural modeling systematically considers the timing effects of underlying processor features, such as pipeline, caches, branch prediction and so on. For single-core processors, such a micro-architectural modeling involves the analysis of a single program occupying the processor. However, this criterion no longer holds with multi-core processors. Since their inception, multi-core processors have widely been adopted due to their high-performance and low-power characteristics. Unfortunately, multi-core processors pose some significant challenges in terms of time-predictability. Basically, these challenges arise due to the presence of shared resources, such as shared caches and shared buses [5]. The presence of shared resources makes the WCET analysis significantly more complex than the WCET analysis on single-core processors. In particular, micro-architectural modeling is affected due to the presence of inter-core interferences, such as shared cache conflicts or bus contention. Through this monograph, we primarily aim to highlight the recent advances to address such challenges.

As mentioned in the preceding paragraph, shared resources are the key bottlenecks to build time-predictable embedded software on multi-core platforms. The content of a shared cache is modified by several programs running in parallel on different cores. Therefore, the modeling of inter-core cache conflicts is important to estimate the shared-cache latency accurately. For bus-based systems, shared buses introduce variable access latency to the shared resources (*e.g.* shared caches and main memory). Such a variable access latency highly depends on the *bus contention*, which in turn depends on the amount of memory traffic generated by different cores. In this monograph, we shall first describe an approach to model the timing behavior of shared caches [21]. Such a modeling systematically combines abstract interpretation with *state-of-the-art* program-verification techniques (*e.g.* model checking and symbolic execution). In particular, such an approach leverages both the *scalability* offered by abstract interpretation and the *accuracy* offered by program-verification methods to build a tight modeling of shared caches. We then describe works on analyzing timing behavior for static bus-arbitration policies, such as *time division multiple access* (TDMA). Even with static bus-arbitration policies, an accurate analysis of shared-bus delay is complex. This

is due to the reason that bus delay highly depends on the *context*, such as individual loop iterations and procedure calls. In the worst-case, each loop iteration may experience different bus delay. We describe works [69, 9, 22] in this direction whose requirements range from *full-fledged loop unrolling* to *avoiding loop unrolling altogether*, depending on the analysis accuracy.

Subsequently, we discuss the development of a full-fledged WCET analysis framework by combining the modeling of shared resources [18]. Such a combination is non-trivial due to the possible presence of *timing anomalies* [59]. In the presence of timing anomalies, a local worst-case (*e.g.* a cache miss or maximum bus delay) may not lead to the overall WCET of a program. As a result, it is *unsound* to model the timing behavior of each micro-architectural component and get the overall timing behavior by a simple composition of individual timing models. This framework systematically models the timing interaction of shared resources with the rest of the micro-architectural features (*e.g.* pipeline, branch prediction) and it does not assume a *timing-anomaly-free* architecture. The WCET analysis framework is built on top of `Chronos` [52], a freely-available, open-source WCET analysis tool. We show the evaluation of this analysis framework via several experiments.

Besides modeling individual micro-architectural features in multi-core processors, predictability of embedded software can also benefit from customized compiler optimizations and time-predictable multi-core hardware. In this direction, we discuss an optimization of bus schedules to improve time-predictability. Specifically, we describe the generation of customized bus schedules that may greatly improve the WCET of a program [69]. Finally, we discuss several designs of time-predictable hardware to reduce the pessimism in the WCET analysis on multi-core platforms.

The main purpose of this monograph is to give the readers a thorough background on time-predictability for multi-core platforms. Therefore, we have also performed a discussion of research activities by several research groups in this area. This discussion provides a comprehensive overview of the state-of-the-art solutions in the respective topic. In particular, our discussion reveals that the area is fast evolving and there is an active interest by real-time research groups on the topic discussed in this monograph. Finally, in the concluding section of this monograph, we have highlighted a set of open challenges in achieving high-performance and time-predictable

embedded software on multi-core platforms. We hope that this monograph will provide a foundation of building time-predictable software on multi-core platforms and it will help the research community to address the existing challenges in this area.

2

WCET analysis and challenges with multi-core architecture

In this section, we shall first give the readers a general background on WCET analysis. Subsequently, we shall discuss the specific challenges that appear in the context of multi-core processors.

2.1 A background on WCET analysis

WCET analysis aims to obtain an upper bound on the execution time of a program. Execution time of a program critically depends on the provided input. Since the set of all possible inputs is often unbounded, it is, in general, impossible to explore the entire input space. For instance, the execution time of a video player cannot be analyzed by considering all possible videos. Besides, it is essential to have a clear domain knowledge about the software to understand its input space. On the other hand, static WCET analysis [83] is a powerful mechanism which analyzes a program irrespective of its input and provides an upper bound on the program's execution time. Such a static WCET analysis works on an abstract representation of the program, usually, the control flow graph (CFG). As a result, a *sound* upper bound on the execution time of a program can only be obtained via static WCET analysis. We shall now discuss the different stages of a static WCET analysis framework.

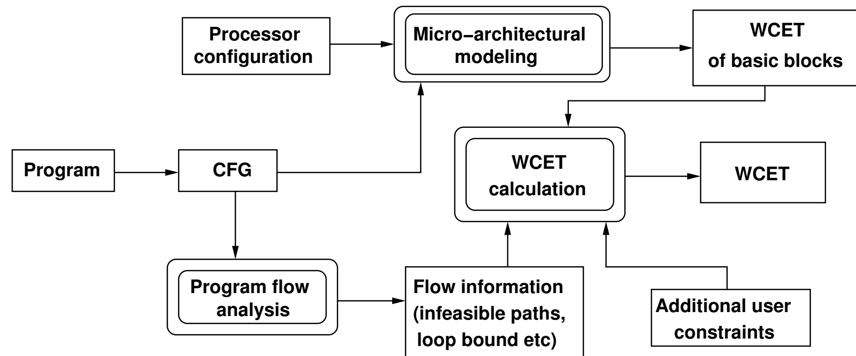


Figure 2.1: Overview of a typical WCET analysis framework

As mentioned in the preceding paragraph, static WCET analysis typically works on the control flow graph (CFG) of a program. Such a static WCET estimation involves three phases: program flow analysis (to find infeasible program paths and loop bounds), micro-architectural modeling (to determine 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.

Figure 2.1 captures an overview of a typical WCET analysis process. Micro-architectural modeling usually works at the level of basic blocks in the program CFG and it 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 paths) are used to compute the WCET of the entire program, as shown in Figure 2.1.

Now we shall explain each of the three stages of WCET analysis.

2.1.1 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 more precise WCET estimation. Consider an example program and its corresponding control flow graph (CFG) shown in Figure 2.2.

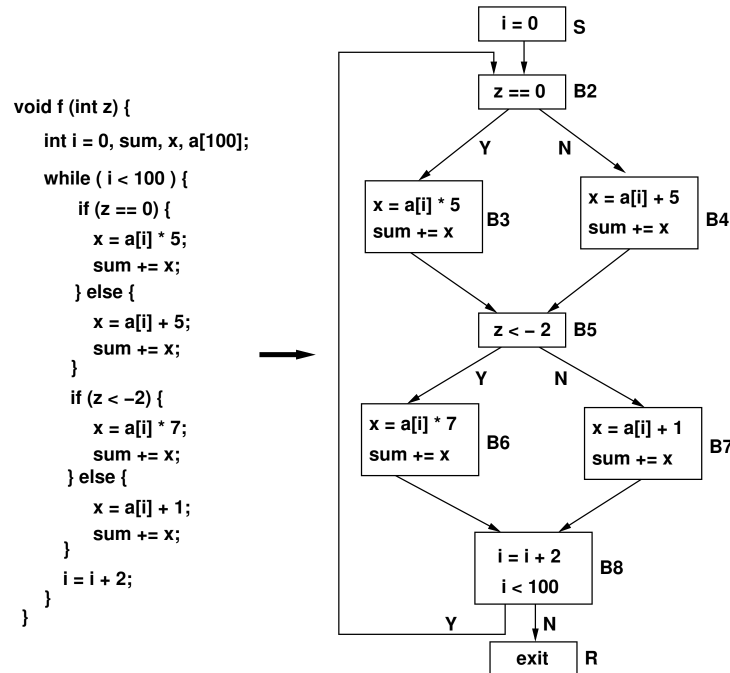


Figure 2.2: An example program and its corresponding control flow graph (CFG)

Without any knowledge of infeasible paths, assume that the WCET analyzer computes `B2-B3-B5-B6-B8` as the worst-case execution 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 precise WCET estimate by focusing on a reduced 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 2.2, 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 critical for estimating the WCET.

The research on flow analysis has focused on automatic discovery of infeasible paths, as well as loop bounds [78, 38, 42, 58]. Note that the discovery of loop bounds is an undecidable problem. Therefore, 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 framework to get a more precise WCET estimation.

2.1.2 Micro-architectural modeling

The 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 a WCET analysis process. To better understand the importance of micro-architectural modeling in WCET analysis, let us consider the example shown in Figure 2.3. Through the example in Figure 2.3, we shall show why the timing effects of the underlying micro-architecture cannot be ignored for a *sound* WCET analysis. Figure 2.3(a) shows the CFG of a pro-

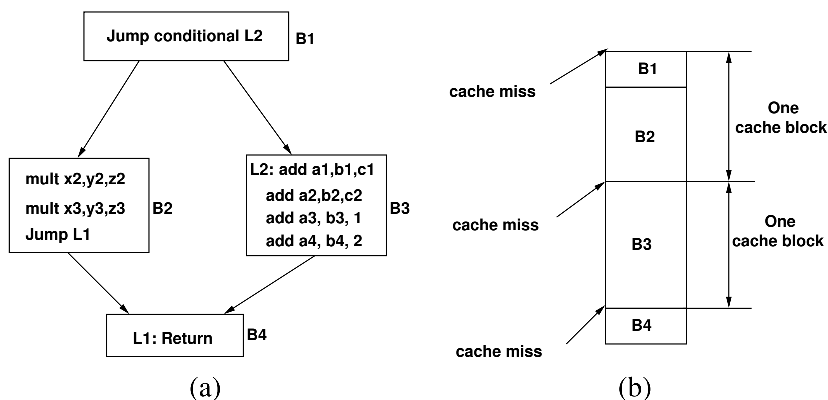


Figure 2.3: (a) A program CFG with two paths, (b) memory layout of the program code

gram 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`) in-

structions 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 might 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 2.3(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 the 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 are of prime importance for a *sound* WCET estimate.

In the past two decades, an extensive amount of research effort has been put forward for micro-architectural modeling. One of the first few approaches includes the use of integer linear programming (ILP) [55], but the use of ILP poses scalability issues due to the presence of a huge number of ILP constraints. Subsequently, the work in [80] proposes a scalable approach of using abstract interpretation for micro-architectural modeling. Since its inception [29], abstract interpretation has been successfully applied to handle several challenges, including functionality testing and compiler optimization. In [80], abstract interpretation was proposed to be used for WCET analysis. The basic framework proposed in [80] has later been extended by many research efforts to analyze advanced micro-architectural features, such as data caches [45], multi-level caches [40], pipeline [51] and branch predictor [28].

Compositional vs non-compositional architecture

In the context of WCET analysis, we distinguish between two architectures: *(i)* compositional architectures, and *(ii)* non-compositional architectures [84]. For compositional architectures, we can build timing models of each micro-architectural components (*e.g.* pipeline, caches, branch predictors) in isolation and obtain the timing model of the overall architecture by a simple *composition* (*e.g.* adding the worst-case delays suffered in each component). Be-

sides, we can be sure that a local worst-case scenario always contributes to the worst-case globally. For instance, a cache miss (instead of a cache hit) can always be considered during micro-architectural modeling to compute the global WCET. In a similar fashion, if a basic block in the CFG has different starting time, the worst case starting time can always be taken into account (during the WCET calculation phase as shown in Figure 2.1) to compute the overall WCET of the program.

However, modern embedded processors may exhibit complex timing interactions between different micro-architectural components (*e.g.* between pipeline and caches). In general, for such architectures, it is not sufficient to consider each micro-architectural components in isolation. We call such architectures *non-compositional*, in the context of WCET analysis.

Non-compositional architectures exhibit *timing anomalies* [59], which makes the micro-architectural modeling substantially more complex than compositional architectures. Timing anomaly is defined as follows: assume a sequence of instructions containing a particular instruction I. Further assume that instruction I has 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. *Timing anomalies* occur when $L_1 < L_2$, but $E_1 > E_2$. The following example illustrates timing anomalies for a non-compositional architecture.

Figure 2.4(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 of $1 \sim 4$ cycles. The 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*.

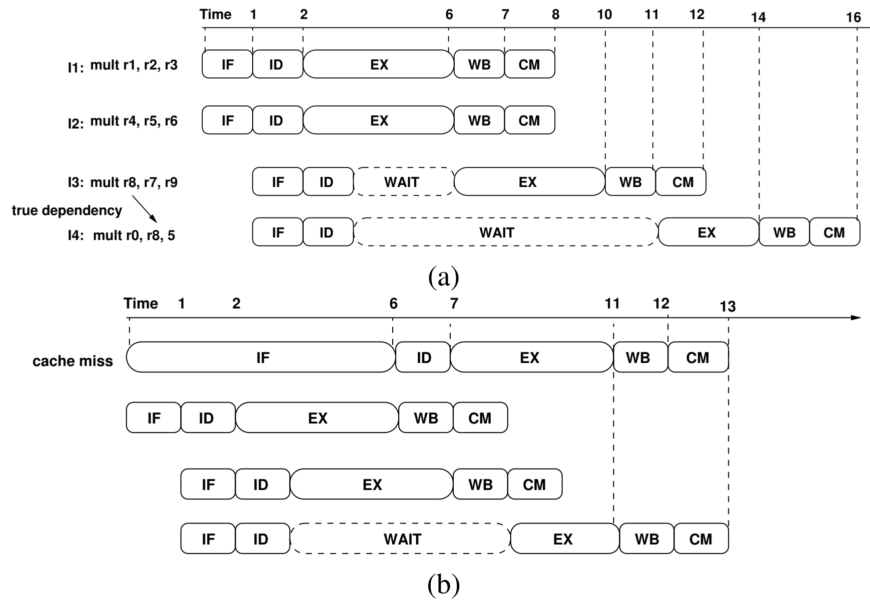


Figure 2.4: An example showing timing anomalies. (a) Execution scenario with $I1$ facing instruction cache hit, (b) execution scenario with $I1$ facing instruction cache miss

In EX_1 (shown in Figure 2.4(a)), instruction $I3$ has to wait until the 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 2.4(b)). In this case, $I3$ can finish execution at the 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 a counter intuitive phenomenon appears due to the complex timing interactions between cache and pipeline. The example in Figure 2.4 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 all possible micro-architectural states. Unfortunately, capturing all possible micro-architectural states is, in general, infeasible. Therefore, existing works use abstract micro-architectural states via abstract interpretation [80, 51] or timing interval abstraction to capture the time taken by each pipeline stage [54, 53].

In section 3 of this monograph, we discuss WCET analysis methodologies for compositional, as well as non-compositional architectures.

2.1.3 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 techniques* 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 the 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 [43]. However, path-based techniques suffer from scalability problems, as they enumerate a huge number of paths. The work of [76] 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 [46]). 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 [80] 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 paths, loop bound) can easily be encoded as linear constraints and they can be integrated into the

WCET formulation (as shown in [38, 47]). Consequently, ILP-based WCET computation has become popular in the research community. Many WCET analyzers currently employ an ILP-based (such as Chronos [52], aiT [1]) calculation phase.

2.1.4 WCET calculation via ILP: an illustrative example

In this section, we shall illustrate the WCET computation by revisiting the example shown in Figure 2.2. We shall use the *implicit path enumeration based* WCET calculation for the illustration.

WCET analysis is usually carried out on the executable code to take into account all the compiler optimizations. But for the sake of simplicity, in this discussion, we shall show the process at the source code level. Figure 2.5 revisits the CFG of the example program in Figure 2.2 and it also shows the ILP constraints.

Let us assume that C_B denotes the WCET of basic block B derived via micro-architectural modeling. Note that, for a non-compositional architecture (*e.g.* an architecture that exhibits *timing anomalies*), WCET computation of each basic block B takes into account all possible execution contexts of B. 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 2.5(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

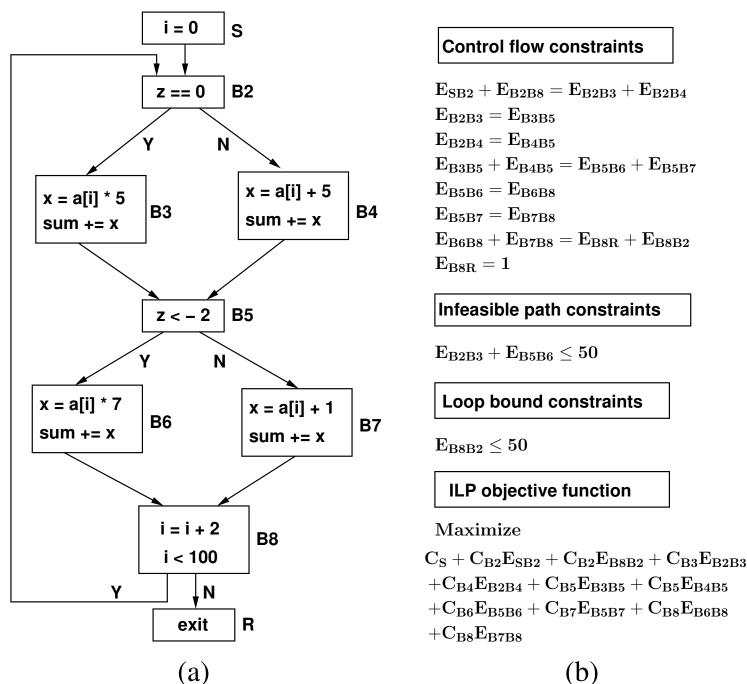


Figure 2.5: An example showing ILP-based WCET calculation (a) program control flow graph, (b) ILP formulation

be explicitly specified by the user or it can also be derived through a complex analysis of the program. For instance, the work proposed in [58] uses a novel combination of abstract interpretation and program slicing to precisely compute loop bounds of a program.

Representing infeasible paths: Certain infeasible path informations can be represented as linear constraints and therefore, they can easily be integrated into the ILP-based calculation. Note that basic blocks `B3` and `B5` cannot both 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 2.5(c).

An ILP solver (*e.g.* CPLEX) maximizes the objective function (as specified in Equation 2.1) considering all specified constraints to it (Figure 2.5(c)).

2.2 Challenges in WCET analysis for multi-core architectures

There exists a vast variety of multi-core processors in the market. However, instead of going into the specific hardware implementation of a multi-core processor, we shall mainly concentrate on an abstract architecture that is common for most multi-core platforms. Figure 2.6 shows one such architecture. Each core has a private L1 cache. Therefore, L1 cache contents are not affected by inter-core interferences. This L1 cache might be a split-cache (*i.e.* instruction and data memory do not share space in L1 cache) or a unified cache. Besides, all the cores share an L2 cache, which acts as a back-up memory for L1 caches. If a memory access misses in both the L1 and L2 cache, the respective memory block has to be fetched from the main memory (usually a DRAM). This off-chip memory is several magnitudes slower than the caches. We assume a bus-based system. Therefore, all traffic to the shared cache and off-chip memory has to access the shared bus, which in turn is controlled by a bus arbiter.

It is worthwhile to note that the shared bus, in commercial processors, is usually located between the L2 cache and main memory. In Figure 2.6, the shared bus primarily captures a medium to access shared caches or main memory. In the context of WCET analysis, our intention is to convey the information that this shared medium may introduce additional delay to access shared caches or main memory. In general, this shared medium might be implemented using a very complex protocol. However, our only intention is to portray the fact that such a shared medium should exist. For shared caches, such a shared medium is needed when several threads want to access the same L2 cache bank. In commercial processors, the cache controller usually serializes such requests. In Figure 2.6, for simplicity, it is assumed that the shared bus serializes such accesses to the shared cache as well. In the context of hard real-time processing, such an architecture has also been implemented [3] and it follows time-division-multiple-access (TDMA) arbitration scheme to serialize shared cache access requests.

We argue that WCET analysis in the presence of multi-core platforms is substantially more complex than WCET analysis for single-core architectures. The key to such complexity arises due to resource sharing. Analyzing the WCET on a single-core processor we can exclusively concentrate

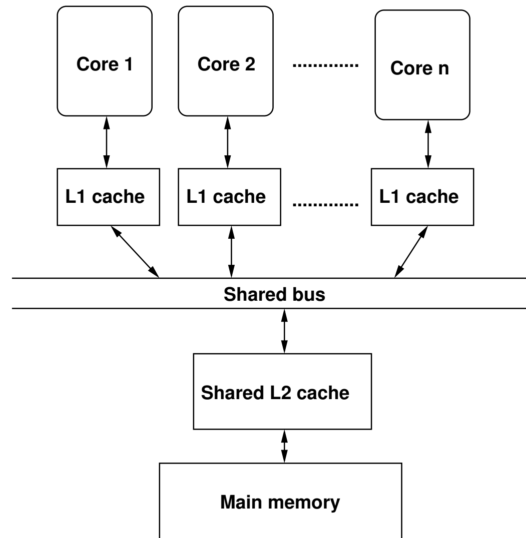


Figure 2.6: A typical multi-core architecture with resource sharing (*i.e.* shared cache and shared bus)

on the program under analysis. Unfortunately, this simple property does not hold when analyzing WCET for multi-core platforms. The content of shared caches (*cf.* Figure 2.6) might be modified by different programs running on multiple cores. As a result, to analyze the WCET on multi-core platforms, we must know the amount of interferences generated on shared caches. However, inter-core interference on shared caches highly depends on the interleaving patterns of programs that are running on multiple cores. To illustrate the problem more deeply, let us first go through a simple example. Assume that two programs T and T' are running on two different cores. Program T accesses memory blocks m twice in sequence, while program T' accesses memory blocks m' twice in sequence. Finally, assume that m and m'map to the same shared cache set. If the shared cache is direct-mapped and all accesses miss the L1 cache, an interleaving access pattern $m \rightarrow m \rightarrow m' \rightarrow m'$ will introduce two shared cache misses. On the contrary, an interleaving access pattern $m \rightarrow m' \rightarrow m \rightarrow m'$ will lead to four shared cache misses.

However, it is impossible to enumerate all interleaving access patterns due to the huge exponential complexity. Therefore, an exhaustive solution

is impossible in practice. To avoid this exponential complexity, a suitable abstraction is required. The primary purpose of such an abstraction will be to estimate the amount of inter-core interferences on shared caches in a *sound* manner and still avoid the exhaustive enumeration of access patterns.

Analysis of shared buses also faces complication due to the reason mentioned in the preceding paragraph. Since several cores may access the bus at the same time, a bus access might be delayed. This delay captures the time between a bus access request is made and the time when the bus access is granted. This waiting period for bus access highly depends on the bus arbitration policy and memory traffic generated by different cores. However, the time-predictability can be substantially improved if the arbitration policy is software controlled and it is available at compile time. Time-division-multiple-access (TDMA) is one such arbitration policy, where dedicated bus slots are available to each core for communication. Even in the presence of software-controlled arbitration policies, an accurate analysis of shared buses is complicated. This is due to the presence of variable bus delay. The same memory reference (*i.e.* a memory access instruction in a program) may experience highly different bus delay in different contexts, such as in different calling contexts and in different loop iterations. In an extreme case, each loop iteration may suffer different bus delays. Such a phenomenon makes WCET analysis extremely complicated due to a substantial increase in the number of micro-architectural contexts to consider. Of course, it is possible to consider the worst-case bus delay for each micro-architectural context and derive a safe upper bound on WCET. However, such a naive methodology will substantially increase the analysis overestimation. As a result, such an analysis methodology will not be very useful in practice.

In subsequent sections, we shall discuss several efforts in addressing the challenges mentioned in this section. Therefore, this monograph primarily focuses on the micro-architectural modeling stage of WCET analysis (*cf.* Figure 2.1), in the context of multi-core processors. We shall also perform a survey of other related techniques to address such challenges.

3

WCET analysis for multi-core platforms

In the preceding section, we have discussed the challenges in building time-predictable systems for multi-core platforms. In this section, we shall describe recent efforts in addressing such challenges. Specifically, we shall describe a comprehensive analysis methodology for predicting the worst-case execution time (WCET) of embedded software. We shall also perform a survey of related techniques proposed by different research groups along this line.

As mentioned in the previous section, resource sharing is a key feature in multi-core platforms. Such resource sharing in multi cores primarily happens via shared caches and shared buses. Therefore, in the following discussion, we shall mainly concentrate on the modeling of shared caches and shared buses. We shall also describe the interaction between the timing models of shared resources and the rest of the micro-architecture (*e.g.* pipeline and branch prediction).

3.1 Modeling shared caches

Shared cache modeling revolves around the modeling of caches on single core processors. Therefore, we shall start with a general background on cache analyses for single core platforms.

3.1.1 Background on cache modeling

Cache modeling has been an active topic of research for several decades. In modern embedded processors, caches are several order of magnitudes faster than the main memory. Therefore, to accurately analyze the timing behavior of an embedded software, it is crucial to know whether a particular memory reference can be serviced from cache. Existing research on cache modeling estimates the overall cache performance of a program via static analysis.

Earlier work on cache analysis [55] used integer linear programming (ILP) to analyze the cache behavior of a program. However, ILP-based modeling of caches faces scalability problems for large caches and programs with complex structures. Subsequently, a pioneering work [80] introduces the usage of abstract interpretation (AI) for cache analysis of embedded software. Analysis based AI has been shown to scale well and has also been adopted in industry-strength tool chain, such as aiT [1]. AI-based analysis categorizes each memory reference as *always hit* (AH), *always miss* (AM) or *unclassified* (NC). The memory block corresponding to an AH categorized memory reference is always in cache when accessed. On the contrary, the memory block corresponding to an AM categorized memory reference is never in cache when accessed. If a memory reference cannot be categorized as AH or AM, it is categorized as *unclassified* (NC). The precision of AI-based cache analysis can be improved significantly via *virtual inlining and virtual unrolling* (VIVU) [80]. Using *virtual inlining*, different calling contexts of a procedure are treated differently. Since the calling context may significantly affect the content in caches, *virtual inlining* plays a crucial role in improving the accuracy of static cache analysis. Besides, using *virtual unrolling*, each loop is unrolled once to distinguish the cold cache misses in the first iteration. The cache analysis proposed in [80] deals with single-level caches in single-core architectures. Based on abstract interpretation, several works have

subsequently extended the single-level cache analysis to multi-level caches [40, 19], data caches [45] and shared caches [39, 56, 85].

Analysis of shared caches is more complex due to the presence of inter-core cache conflicts. Such inter-core cache conflicts are generated by tasks running on different cores. Until now, only a few solutions have been proposed for analyzing timing behaviors of shared caches [56, 39, 85]. However, all of them suffer from overestimating the inter-core cache conflicts. In the subsequent section, we shall describe an analysis framework that systematically combines abstract interpretation and path-sensitive verification (*e.g.* model checking and symbolic execution) to improve the estimation of inter-core cache conflicts. Such a framework improves the accuracy of a baseline AI-based analysis via repeatedly using model checker calls. As a baseline, the framework uses the AI-based shared cache analysis presented in [56]. Recall that AI-based cache analysis categorizes memory references as *always hit* (AH), *always-miss* (AM) and *unclassified* (NC). The work proposed in [56] first analyses the shared cache in the absence of inter-core cache conflicts and derives the categorization (*i.e.* AH, AM or NC) of each memory reference. Subsequently, a separate inter-core conflict analysis phase is employed to refine the categorization of each memory reference. Such a refinement primarily takes into account the inter-core cache conflicts generated in the shared cache. To be more precise, inter-core conflict analysis may change the categorization of a memory block m from *always 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 the shared cache. More formally, cache hit/miss categorization (CHMC) of m is changed from *always hit* (AH) to *unclassified* (NC) if and only if the following condition holds:

$$N - age(m) < |\mathcal{M}_c(m)| \tag{3.1}$$

where $|\mathcal{M}_c(m)|$ captures the number of conflicting memory blocks from different cores which may potentially access the same set in the shared cache as m. N represents the associativity of the shared cache and $age(m)$ captures the *age* of memory block m in the shared-cache set in the absence of inter-core conflicts. Note that $age(m)$ captures the relative position of memory block m in the respective cache set, when memory blocks are ordered in

terms of eviction. Therefore, $1 \leq age(m) \leq N$ and $N - age(m)$ unique memory blocks are sufficient to evict out m from the cache. We call the term $N - age(m)$ as *residual age* of m[1].

3.1.2 A scalable approach for shared cache analysis

In the preceding section, we have argued that the modeling of shared caches is more challenging due to the presence of inter-core cache conflicts. In this section, we shall present a scalable solution [21] that significantly improves the analysis precision over the *state-of-the-art* shared cache analysis.

Basic idea

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 (AI), as discussed in Section 3.1.1. Abstract interpretation is usually efficient, but the results are often not precise. This is because the estimation of memory access behaviors are “joined" at the control flow merge points – resulting in an over-estimation of potential cache misses returned by the method.

In this section, we present a cache analysis framework which improves the precision of abstract interpretation, without appreciable loss of efficiency. This framework augments abstract interpretation with a gradual and controlled use of path sensitive program verification methods (*e.g.* model checking and symbolic execution). Because of path sensitivity in the search process, program verification methods are known to be of high complexity. Hence AI-based analysis cannot be naively replaced with standard program verification methods such as model checking or symbolic execution. Recent works [60] which have advocated combination of abstract interpretation and model checking for multicore software analysis – restrict the use of model checking to program path level; cache analysis is still accomplished only by

[1]This metric is also called *resilience* of m according to [7]

abstract interpretation. Indeed almost all current state-of-the-art WCET analyzers (such as Chronos [52], aiT [1]) 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 [82, 44].

The baseline analysis is abstract interpretation. Potential cache conflicts identified by abstract interpretation are then subjected to a path sensitive program verification method. The goal is to rule out "false" cache conflicts which can occur only on infeasible program paths. Such false conflicts are reported 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 program verification naturally rules out the infeasible program paths and the cache conflicts incurred therein.

One appealing nature of this analysis method is that the results are always safe. The analysis starts with the results from abstract interpretation and gradually refines the results with repeated runs of program verification. We show the instantiation of the framework with two different program verification methods – model checking and symbolic execution.

Model checking is a property verification method which takes in a system/program P and a temporal logic property φ, where φ [2] is interpreted over the execution traces of P. It checks whether all execution traces of P satisfy φ. Given a potentially conflicting pair of 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.

Symbolic execution refers to executing a program with symbolic or uninstantiated inputs - as opposed to concrete inputs. Symbolic execution may be static (by which we mean execution of all possible paths in a program) or dynamic (by which we mean execution of a specific program path). We show the use of static symbolic execution (as embodied in the KLEE toolkit [2]) for refining shared cache analysis.

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

Most often, a symbolic execution engine relies on the power of constraint solving. Constraint solving technology has made a significant progress with the advances in *satisfiability modulo theory* (SMT). As mentioned, in symbolic execution, a program is executed with *symbolic* input values (rather than concrete input values in normal execution). Since the input values are *symbolic*, a branch instruction in the program may lead to multiple execution scenarios, as both the *true* and *false* legs of the branch might be satisfiable. Such multiple execution scenarios are reasoned about independently by the symbolic execution engine. The feasibility of a path at a branch instruction is checked *on-the-fly* during the execution by sending a query to an SMT-based constraint solver. Given a formula φ to check at a particular program location, the constraint solver is also used to check the satisfiability of φ whenever the same program location is visited by any execution scenario during the symbolic execution.

Due to the inherent path sensitive nature of symbolic execution, spurious cache conflicts can be eliminated if 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 symbolic execution leads to an exciting opportunity for WCET analysis.

Recall that abstract interpretation merges the results from different paths, via the join function. Thus, abstract interpretation is not necessarily path-sensitive. On the other hand, the property checked in a single run of program verification (via model checking or symbolic execution) involves certain cache conflicts identified by abstract interpretation. The path sensitive search by program verification then detects whether these conflicts are indeed realizable. Overall, the scalability of such a framework is never in question. Given a time budget T, one can first employ abstract interpretation and then employ as many runs of program verification as (s)he can within time T. Of course, given more time, more precise analysis results (in the form of potential cache misses) are achieved.

General framework

Figure 3.1(a) demonstrates the general cache analysis framework. Specifically, Figure 3.1(a) highlights the relevant portion of micro-architectural modeling (*i.e.* the modeling of caches) in a typical WCET analysis frame-

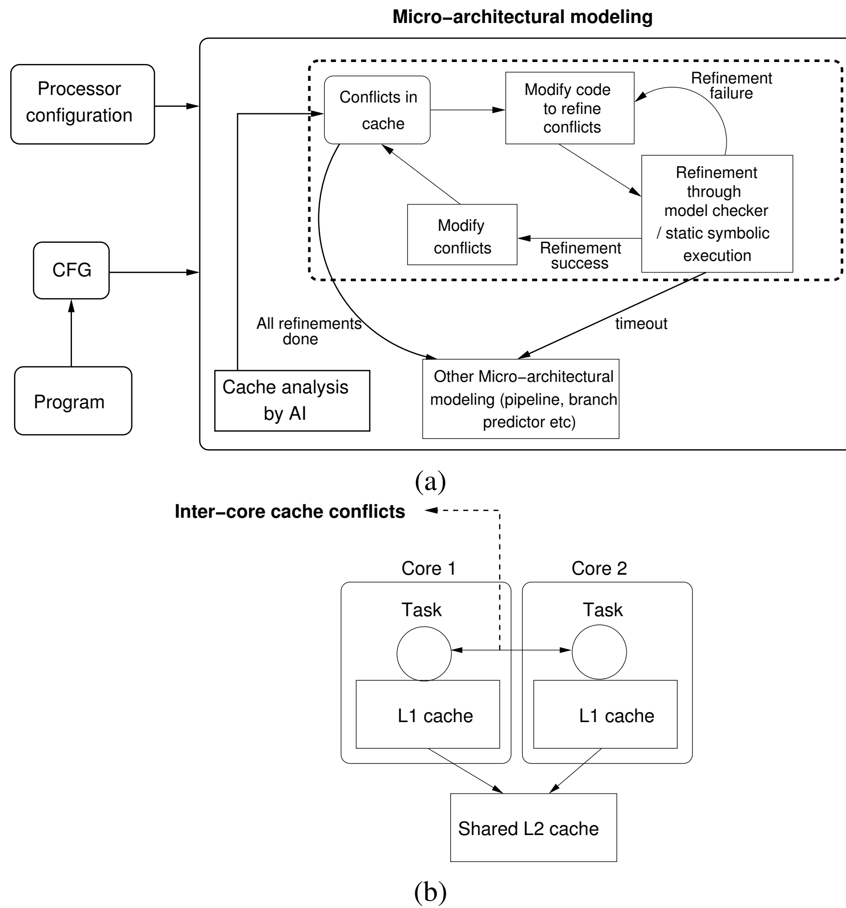


Figure 3.1: (a) General framework of cache modeling which combines abstract interpretation and path-sensitive verification, note that the block "Micro-architectural modeling" is connected with the rest of the WCET analysis framework as shown in Figure 2.1, (b) inter-core cache conflicts

work (*cf.* Figure 2.1). The goal is to refine the AI-based cache analysis via path-sensitive verification (*e.g.* model checking, symbolic execution). *Cold cache misses* are unavoidable and AI-based cache analysis can accurately predict the set of cold cache misses. With the advent of multi-core architectures, it has become important to precisely estimate the timing behavior of shared caches. AI-based shared cache analysis suffers from overestimating the inter-core cache conflicts, which is generated in the shared cache by a task running on a different core. Figure 3.1(b) pictorially represents the inter-core cache conflicts generated in the shared cache.

Even though the basic goal of this framework is cache conflict refinement, the notion of cache conflict may vary depending on the outcome of AI-based cache analysis. During 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.

In Figure 3.1(a), the dotted boxed portion captures the shared cache conflict refinement. The refinement of cache conflicts is iteratively performed via path-sensitive verification (*e.g.* model checking or symbolic execution) on a modified program. We rule out the cache accesses for which AI has generated precise information. Therefore, the refinement phase using model checking or symbolic execution works on a very small subset of all cache accesses. The iterative refinement through path-sensitive verification 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.

There are two important advantages of such analysis framework: first, the iterative refinement can be terminated at any point if the time budget is exceeded. The resulting *cache conflicts*, after a partial refinement, can *safely* be used for estimating the WCET. Secondly, such a framework can be composed with other micro-architectural features (*e.g.* pipeline and branch prediction) and thereby, not affecting the flexibility of AI-based cache analysis.

Code transformation to refine inter-core cache conflicts

The refinement of cache conflicts is performed by transforming the original program into an instrumented program. This typical transformation can be captured by a quintuple $\langle \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} \rangle$ 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 path-sensitive verification method. 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.

Any 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 3.1 for refining shared cache conflicts (as shown in Figure 3.1(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 3.2. Parameter z can be considered as an input to the program. The control flow graph (CFG) of the loop body and the accessed memory blocks are also shown in Figure 3.2.

A brief background on model checking

Model checking [26] is a state space exploration method for formal verification of program properties. The general formulation of the model checking

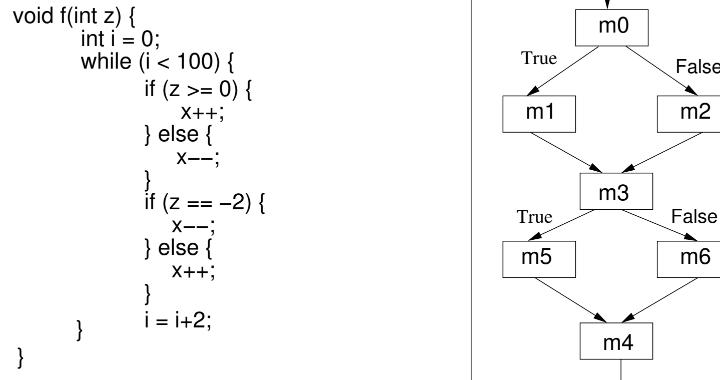


Figure 3.2: Example program and its corresponding control flow graph (CFG)

problem is simple, it checks whether a finite state machine M satisfies a property φ

$$M \models \varphi$$

To explain the use of model checking for program verification we need to explain how we get M, φ and what it means for M to satisfy φ.

The finite state machine M is automatically extracted from the program being verified. Such a finite state machine is formally described as a quadruple $\langle S, S_0, \rightarrow, L \rangle$ where S is the set of nodes (also called states) in the finite state machine, $S_0 \subseteq S$ is the set of initial states, $\rightarrow \subseteq S \times S$ is the set of edges (also called transitions) in the finite state machine, and $L : S \rightarrow 2^{AP}$ is a labeling function, which maps a given state s to the atomic propositions true in the state s. The atomic propositions true in a given state are drawn from AP, the set of all atomic propositions.

The properties verified are temporal logic properties, which constrain ordering of specific events in program executions. In this discussion, we are only concerned with Linear-time temporal logic (LTL). The syntax of LTL properties is recursively defined as follows

$$\varphi = true \mid false \mid AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid G\varphi \mid F\varphi \mid \varphi U \varphi \mid \varphi R \varphi$$

The formula $true$ is always true and the formula $false$ is never true. Further, the atomic propositions AP form the basic building blocks of the formula. A LTL property is constructed using the following

- Atomic propositions AP
- propositional logic operators
- temporal logic operators X (next), G (globally), F (finally), U (until), R (release).

For this framework, the only properties used are in the form of assertions which should hold in a control location of the program. For example consider the assertion $C_1 \leq 1$ which should hold in control location $P2$ in Figure 3.4. It corresponds to a linear time temporal logic property

$$G(pc == P2 \Rightarrow C_1 \leq 1)$$

meaning whenever the program counter variable (denoted pc in the above property) holds the value "$P2$" (*i.e.*, when control location $P2$ is reached during program execution), we must have $C_1 \leq 1$. Given an execution trace π of the program, we can check this property by looking for all the visits to control location $P2$ in the trace π, and then checking whether for each of these visits $C_1 \leq 1$ holds true in the corresponding program state.

Finally, we explain what it means for a finite state machine M to satisfy a given LTL property φ. The semantics of LTL dictates that M satisfies φ if and only if *all* the execution traces of M satisfy φ. In the context of our example property $G(pc == P2 \Rightarrow C_1 \leq 1)$ — even if one single trace of state machine M is such that it has a visit to control location $P2$ when $C_1 \leq 1$ does not hold – we will say that M does not satisfy the property $G(pc == P2 \Rightarrow C_1 \leq 1)$. Such an execution trace π will then be considered as a *counter-example* trace of the property.

Refinement of inter-core cache conflicts via model checking

We describe the refinement of inter-core conflicts generated in a shared cache (as shown in Figure 3.1(b)). Recall from Equation 3.1 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 the analysis framework refines the set $\mathcal{M}_c(m)$ by exploiting infeasible paths in the conflicting task.

Figure 3.3 demonstrates the instantiation of the general framework for inter-core conflict refinement. Specifically, the refinement phase considers

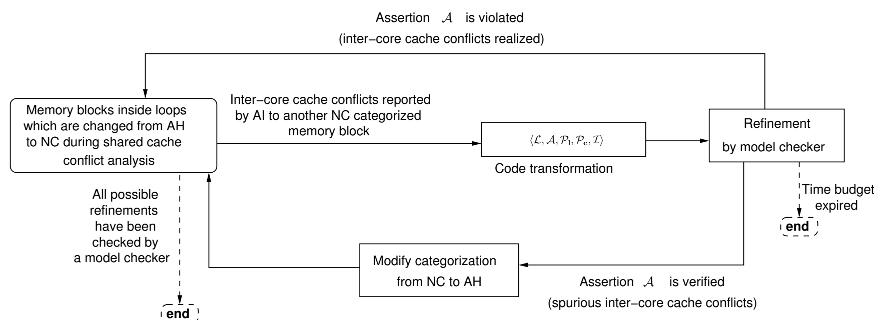


Figure 3.3: Refinement of shared cache conflict analysis

only 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 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, the model checker is used to verify an "assertion" property $C_i \leq N - age(m)$. We need to check the total amount of cache conflicts generated by task t_c. Therefore, in the transformed code, C_i is initialized only once, before any cache blocks accessed by t_c and the "assertion" property is checked just before the exit point of t_c.

For the example in Figure 3.2, we 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', which is running in parallel on a different core with the task in Figure 3.2. Consider t' accesses a memory block m', which maps 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

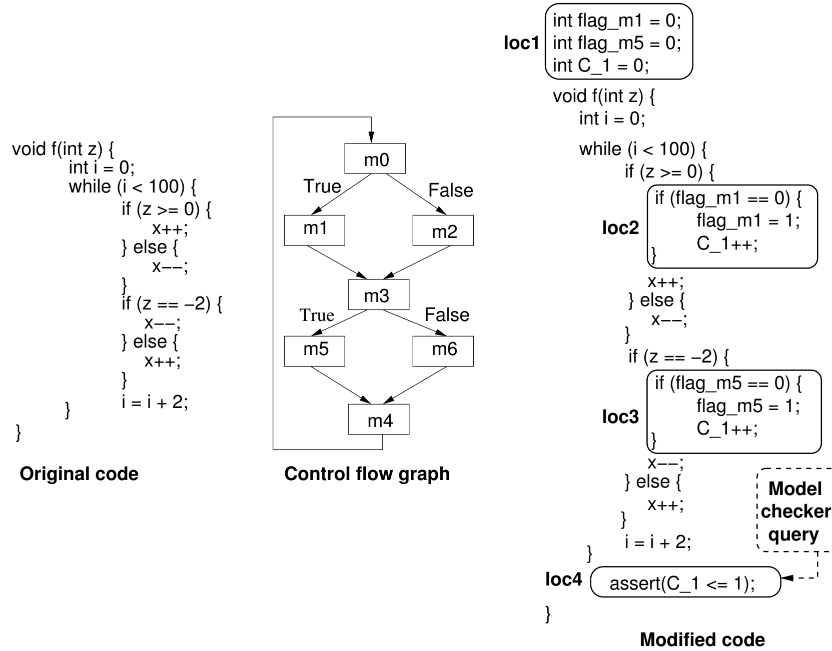


Figure 3.4: Inter-core cache conflict refinement

3.1), leading to unnecessary conflict misses. The code is modified 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 3.4 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 3.4 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 the code transformation framework $\langle \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} \rangle$ is 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, this 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, an additional *compose* phase in the transformation is applied first. 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, the code transformation technique can be applied to T in exactly the 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.

A brief background on symbolic execution

Symbolic execution [50] interprets a program with *symbolic input values* (rather than concrete input values). Any expression, whose value depends directly or indirectly on these symbolic input variables, are treated as symbolic expressions throughout the execution. At any point of interpreting the program, symbolic execution maintains a set of execution states. Each such execution state is associated with a *constraint store*. The *constraint store* is a symbolic formula capturing the set of inputs along which the respective execution state is reached. Let us consider an execution state which has to interpret a branch instruction. At a branch location, the symbolic execution must decide which branch to take. If the branch instruction contains a symbolic expression, such a decision making involves constraint solving. If the constraint solver can decide which branch to take, the execution state proceeds along the respective branch (without creating any additional execution state). Such an interpretation of branch instruction is usually called a "non-forking" execution. The more complex scenario appears when the outcome of a branch instruction cannot be decided – which means that there is at least one input which satisfy the *true* leg of the branch and there is also at least one input which satisfy the *false* leg of the branch. In such a scenario, symbolic execution creates two parallel execution states (called "forking" execution), one for the *true* leg of the branch (say *true state*) and the other for the *false* leg of the branch (say *false state*). Assuming that the branch instruction checks a condition θ and the constraint store of the execution state before branch was Φ, the constraint store of the *true state* is updated as $\Phi \wedge \theta$ and the constraint store of the *false state* is updated as $\Phi \wedge \neg\theta$. Both the *true state* and *false state*

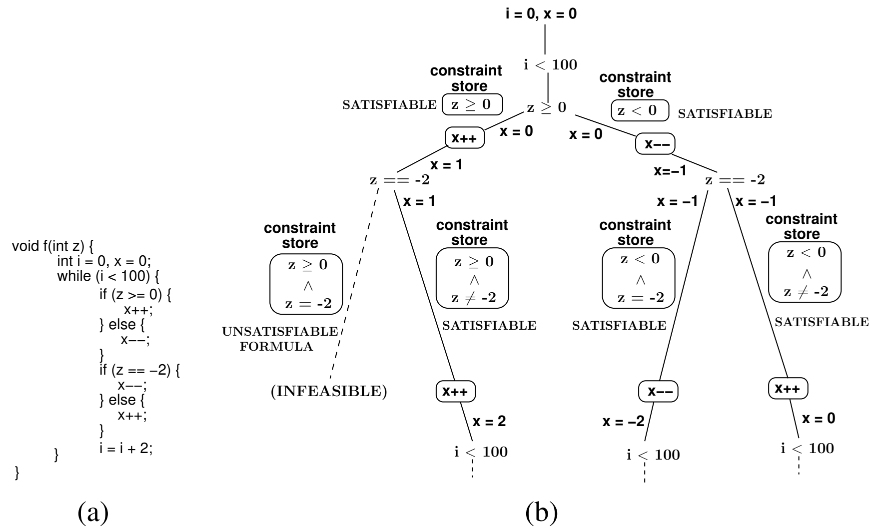


Figure 3.5: (a) Example program, which is the same as in Figure 3.2, (b) symbolic execution

inherit the same computation state before the branch location, but after the branch location, the two execution states proceed independently.

We shall illustrate the work flow of a symbolic execution engine with the example in Figure 3.5. Let us assume that z is an input to the program and therefore, z is marked as *symbolic*. 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 3.5, 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.

Recall that a constraint store is maintained for each execution state created during symbolic execution. The constraint store is a symbolic formula on the input variables which *must be satisfied to reach the respective execution state*. The constraint store is *the logical formula true* at the beginning of the program and is adjusted at each branch instruction. In Figure 3.5(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.

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 constraint solver to decide the condition outcome (*i.e.* true or false). The constraint 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, the symbolic execution 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 at the false leg as $z < 0$. The content of the constraint store is shown beside the control flow edges in Figure 3.5(b).

Now consider the execution state with constraint store $z \geq 0$. When this execution state hits the branch instruction $z == -2$, the constraint 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 quickly by an SMT solver with the theory of linear integer arithmetic. Therefore, the symbolic execution does not create any execution state which corresponds to the unsatisfiable constraint store $z \geq 0 \wedge z = -2$ (as marked “INFEASIBLE” in Figure 3.5(b)).

When the execution state with constraint store $z < 0$ hits the branch location $z == -2$, both the formulae $z < 0 \wedge z = -2$ and $z < 0 \wedge z \neq -2$ are *satisfiable* for some input. Therefore, the symbolic execution forks two execution states accordingly. As shown in Figure 3.5(b), both these execution states inherit the value of $x = -1$ before the branch location $z == -2$, however, proceeds independently thereafter to update $x = -2$ (for the execution state with constraint store $z < 0 \wedge z = -2$) and update $x = 0$ (for the execution state $z < 0 \wedge z \neq -2$), respectively.

Eventually, only three different execution states are created (as shown in Figure 3.5(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 is terminated when it finishes interpreting all the instructions in all the three execution states (as shown in the preceding).

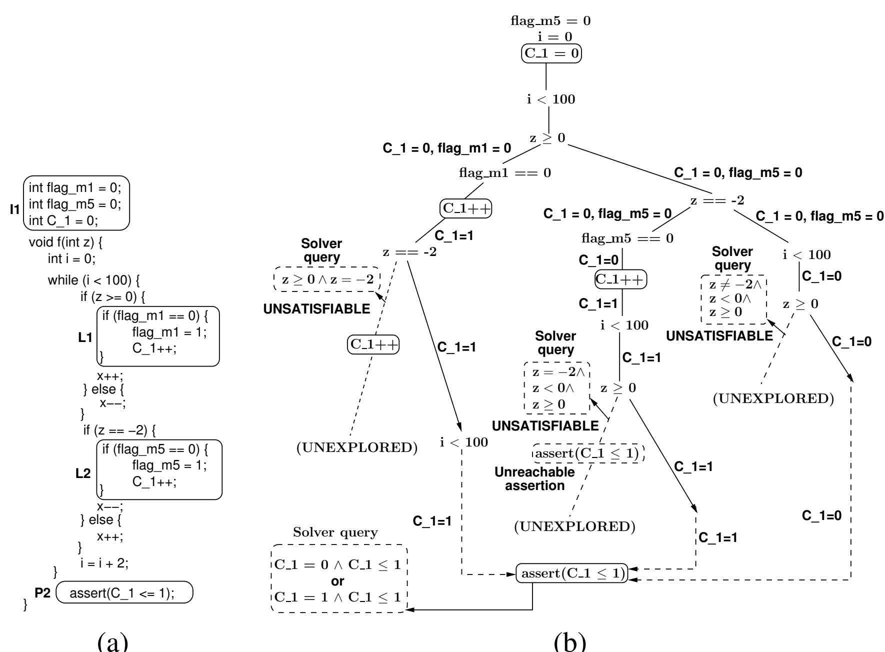


Figure 3.6: (a) Transformed code for checking cache conflict, (b) checking the assertion during static symbolic execution

Refinement of inter-core cache conflicts via symbolic execution

Symbolic execution has successfully been applied to discover many critical functionality bugs [17]. At a high level, the code transformation framework can be viewed as reducing the problem of cache timing checking to functionality checking. Recall that the 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 symbolic execution. If the assertion property $\mathcal{A}$ is violated at any execution state created by the symbolic execution, the entire symbolic execution 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 symbolic execution is not aborted, we can prove that the introduced as-

sertion 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 3.6. Figure 3.6(a) shows the instrumented code for inter-core cache conflict refinement (we use the same example from Figure 3.4). Figure 3.6(b) shows the cache conflict refinement process via symbolic execution. Figure 3.6(b) shows that only one execution state (among all three) can execute the assertion property involving the variable C_1. As evidenced by Figure 3.6(b), the execution state interpreting the assertion property captures an input condition $z \geq 0$. Since symbolic execution interprets the program, at each program point it holds the value of all the registers and memory locations. At the assertion location, the respective execution state checks whether the currently stored values satisfy the assertion. Since C_1 has a value of zero initially, a formula of the form $C_1 = 0 \wedge C_1 \leq 0$ is sent to the constraint solver as a query. If the constraint solver returns a *satisfiable* formula, we can conclude that the assertion property holds for the corresponding execution. Note that C_1 is incremented only for the execution states which satisfy input condition $z = -2$. On the other hand, the assertion property is reachable only if the input condition $z \geq 0$ is satisfied. As a result, none of the execution states which increment the variable C_1 can reach the assertion property (as marked "Unreachable assertion" in Figure 3.6(b)). Consequently, whenever the assertion property is reached, the same formula (*i.e.* $C_1 = 0 \wedge C_1 \leq 0$) is sent to the constraint solver. Therefore, symbolic execution is never aborted for the example and we can conclude that the cache conflicts captured by the instrumented code in Figure 3.6(b) cannot appear in *any real execution*.

Note that the symbolic execution engine tries to reason about a program *path-by-path*. Due to this *path sensitive* reasoning process, such a symbolic execution may generate very precise results compared to an equivalent abstract interpretation based analysis. Since the sole purpose of the refinement process is to check the inserted assertion property, the symbolic execution can be aborted as soon as a violation of the assertion property 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. This means that unsuccessful refinements of cache conflicts usually take less time to manifest, compared to the time taken to verify infeasible cache conflicts.

Optimization

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

The outcome of each refinement phase is stored as a triple $\langle set,\ result_{mc},\ conflicts\rangle$. The triple has the following meaning:

- set : Cache set for which the refinement is being made.
- $result_{mc}$: Returned result by the verifier. Assume $result_{mc}$ is one for a successful verification and zero otherwise.
- $conflicts$: Number of conflicts in the assertion property. For an assertion property $C_i \leq N$, value of $conflicts$ is N.

In Figure 3.4, we store $\langle 1, 1, 1\rangle$ 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 $\langle set,\ result_{mc},\ conflicts\rangle$ 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 smaller 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 verifier is made. Depending on the outcome, the new result is cached accordingly for future use.

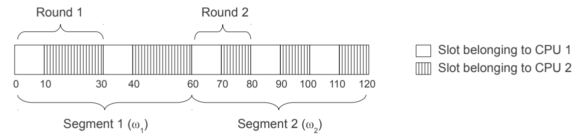


Figure 3.7: Example of a bus schedule

3.2 Modeling shared buses

In the previous section, we have discussed the challenges of modeling shared caches and an approach to address such challenges. In this section, we shall discuss the design and analysis methodologies of shared buses, which is another crucial component for building time-predictable multi-core systems. In the following, we shall start with an illustrative example and subsequently, we shall describe techniques to model the timing behavior of shared buses (some part of the content has previously been published in [69, 9, 22, 18]). Specifically, in Section 3.2.3 and in Section 3.2.4, we shall describe analysis of buses for *compositional* architectures (as explained in Section 2.1.2) and later in Section 3.3, we shall describe analysis methodologies for *non-compositional* architectures.

3.2.1 Bus model and an illustrative example

A precondition for achieving predictability is to use a predictable bus architecture. Therefore, it is useful to consider a TDMA-based bus arbitration policy, which is suitable for modern system-on-chip designs with QoS constraints [71, 64, 31].

The behavior of the bus arbiter is defined by the *bus schedule*, consisting of sequences of slots representing intervals of time. Each slot is owned by exactly one core, and has an associated start time and an end time. Between these two time instants, only the core owning the slot is allowed to use the bus. A bus schedule is divided into *segments*, and each segment contains a specific *round* (*i.e.* a sequence of slots) that is repeated periodically within the segment. See Figure 3.7 for an example.

The bus arbiter stores the bus schedule in a dedicated external memory, and grants access to the cores accordingly. If core CPU_i requests access to the bus in a time interval belonging to a slot owned by a different core, the

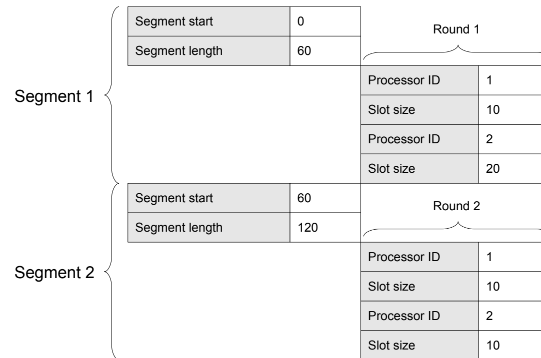


Figure 3.8: Bus schedule table representation

transfer will be delayed until the start of the next slot owned by CPU$_i$. A bus schedule is defined as a sequence of several segments and this sequence of segments is then repeated periodically. A table representation of the bus schedule in Figure 3.7 can be found in Figure 3.8.

To limit the required amount of memory on the bus controller needed to store the bus schedule, a TDMA round can be subject to various complexity constraints. A common restriction is to let every core own, at most, a specified number of slots per round. Also, one can let the sizes be the same for all slots of a certain round, or let the slot order be fixed.

A motivational example Consider two tasks running on a multi-core system with two cores and a shared communication infrastructure according to section 2. Each task has been analyzed with a traditional WCET tool, assuming a single core system, and the resulting Gantt chart of the worst-case scenario is illustrated in Figure 3.9a. The dashed intervals represent cache misses, each of them taking six time units to serve, and the white solid areas represent segments of code not using the bus. The task running on core 2 is also, at the end of its execution, transferring data to the shared memory, and this is represented by the black solid area.

Since the tasks are actually running on a multi-core system with a shared communication infrastructure, they do not have exclusive access to the bus handling the communication with the memories. Hence, some kind of arbitration policy must be applied to distribute the bus bandwidth among the

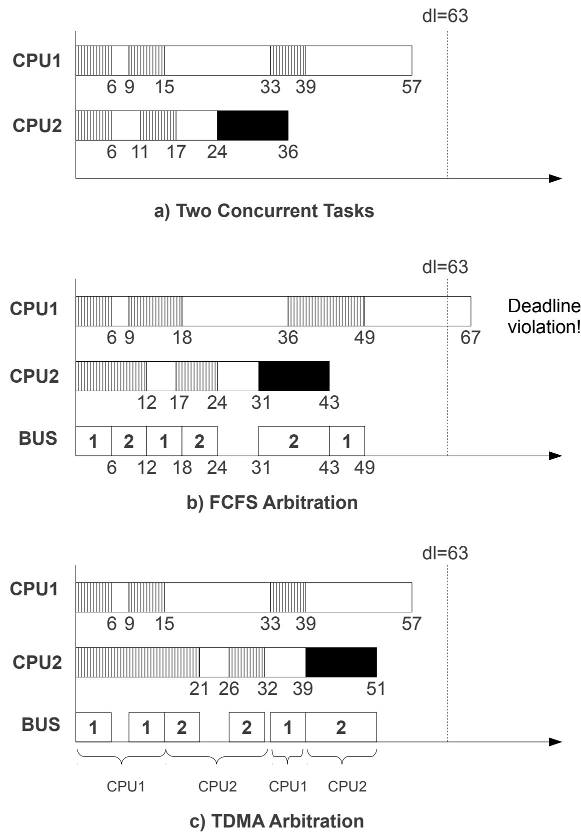


Figure 3.9: Motivational example

tasks. The result is that when two tasks request the bus simultaneously, one of them has to wait until the other has finished transferring. This means that transfer times are no longer constant. Instead, they now depend on the bus conflicts resulting from the execution load on the different cores. Figure 3.9b shows the corresponding Gantt chart when the commonly used *first-come-first-served* (FCFS) arbitration policy is applied.

The fundamental problem when performing worst-case execution time analysis on multi-core systems is that the load on the other cores is in general not known. For a task, the number of cache misses and their location in time depend on the program control flow path. This means that it is very hard to

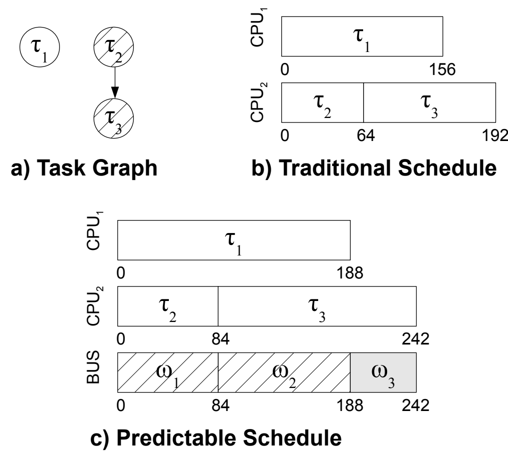


Figure 3.10: Overall approach example

foresee where there will be bus access collisions, since this will differ from execution to execution. To complicate things further, the worst-case control flow path of the task will change depending on the bus load originating from the other concurrent tasks. In order to solve this problem and introduce predictability, a TDMA bus schedule is used which, a priori, determines exactly when a core is granted the bus, regardless of what is executed on the other cores. Given a TDMA bus schedule, the WCET analysis tool calculates a corresponding worst-case execution time. Some bus schedules will result in relatively short worst-case execution times, whereas others will be very bad for the worst case. Figure 3.9c shows the same task configuration as previously, but now the memory accesses are arbitrated according to a TDMA bus schedule.

3.2.2 Overall approach

For a task running on a multi-core system, as described in section 2, the problem for achieving predictability is that the duration of a bus transfer depends on the bus congestion. Since bus conflicts depend on the task schedule, WCET analysis cannot be performed before that is known. However, task scheduling traditionally assumes that the worst-case execution times of the tasks to be scheduled are already calculated.

The solution of this circular dependency (as mentioned in the preceding paragraph) is based on the following principles:

1. A TDMA-based bus access policy, according to Section 3.2.1, is used for arbitration. The bus schedule, created at design time, is enforced during the execution of the application.

2. The worst-case execution time analysis is performed with respect to the bus schedule, and is integrated with the task scheduling process, as described in Algorithm 3.1.

We illustrate the overall approach with a simple example. Consider the application in Figure 3.10a. It consists of three tasks – τ_1, τ_2 and τ_3 – mapped on two cores. The static cyclic scheduling process is based on a list scheduling technique [27], and is performed in the outer loop described in Algorithm 3.1. Let us, as is done traditionally, assume that worst-case execution times have been obtained using techniques where each task is considered in isolation, ignoring conflicts on the bus. These calculated worst-case execution times are 156, 64, and 128 time units for τ_1, τ_2, and τ_3, respectively. The deadline is set to 192 time units, and would be considered as satisfied according to traditional list scheduling, using the already calculated worst-case execution times, as shown in Figure 3.10b. However, this assumes that no conflicts, extending the bus transfer durations (and implicitly the memory access times), will ever occur. This is, obviously, not the case in reality and thus results obtained with the previous assumption are wrong.

In this predictable approach, the list scheduler will start by scheduling the two tasks τ_1 and τ_2 in parallel, with start time 0, on their respective core (line 2 in Algorithm 3.1). However, we do not yet know the end times of the tasks, and to gain this knowledge, worst-case execution time analysis has to be performed. In order to do this, a bus schedule which the worst-case execution times will be calculated with respect to (line 6 in Algorithm 3.1) must be selected. This bus schedule is, at the moment, constituted by one bus segment ω, as described in Section 3.2.1. Given this bus schedule, worst-case execution times of tasks τ_1 and τ_2 will be computed (line 7 in Algorithm 3.1). Based on this output, new bus schedule candidates are generated and evaluated (lines 5-8 in Algorithm 3.1), with the goal of obtaining those worst-

case execution times that lead to the shortest possible worst-case response time (WCRT) of the application.

Assume that, after selecting the best bus schedule, the corresponding worst-case execution times of tasks τ_1 and τ_2 are 167 and 84 respectively. We can now say the following:

- Bus segment ω_1 is the first segment of the bus schedule, and will be used for the time interval 0 to 84.
- Both tasks τ_1 and τ_2 start at time 0.
- In the worst case, τ_2 ends at time 84 (the end time of τ_1 is still unknown, but it will end later than 84).

Now, we go back to step 3 in Algorithm 3.1 and schedule a new task, τ_3, on core CPU_2. According to the previous worst-case execution time analysis, task τ_3 will, in the worst case, be released at time 84 and scheduled in parallel with the remaining part of task τ_1. A new bus segment ω, starting at time 84, will be selected and used for analyzing task τ_3. For task τ_1, the already fixed bus segment ω_1 is used for the time interval between 0 and 84, after which the new segment ω is used. Once again, several bus schedule candidates are evaluated, and finally the best one, with respect to the worst-case response time, is selected. Assume that the segment ω_2 is finally selected, and that the worst-case execution times for tasks τ_1 and τ_3 are 188 and 192 respectively, making task τ_3 end at 276 (since τ_3 can start only after τ_2, which in turn ends at time 84). Now, ω_2 will become the second bus segment of the bus schedule, ranging from time 84 to 188, and this part of the bus schedule will be fixed. Now, we repeat the same procedure with the remaining part of τ_3 (which now ends at time 242 instead of 276, since ω_3 assigns all bus bandwidth to CPU_2). The final, predictable schedule is shown in Figure 3.10c, and leads to a *worst case response time* (WCRT) of 242.

An outline of the algorithm can be found in Algorithm 3.1. We define Ψ as the set of tasks active at the current time t, and this is updated in the outer loop. In the beginning of the loop, a new bus segment ω, starting at t, is generated and the resulting bus schedule candidate is evaluated with respect to each task in Ψ. Based on the outcome of the WCET analysis, the bus segment ω is improved for each iteration. The bus segments previously

Algorithm 3.1 Overall approach

1. **while** not all tasks scheduled **do**
2. schedule new task at $t \geq \theta$
3. Ψ=set of all tasks that are active at time t
4. **repeat**
5. select bus segment ω for the time interval starting at t
6. determine the WCET of all tasks in Ψ
7. **until** termination condition
8. θ=earliest time a task in Ψ finishes
9. **end while**

generated before time t remain unaffected. After selecting the best segment ω, θ is set to the end time of the task in Ψ that finished first. The time t is updated to θ and we continue with the next iteration of the outer loop.

Communication tasks (*e.g.* message passing between two different computation tasks) can be treated as a special class of computational tasks, which are generating a continuous flow of private cache misses (*i.e.* cache misses that lead to shared cache or main memory transactions) with no computational cycles in between. The number of private cache misses is specified such that the total amount of data transferred on the bus, due to these misses, equals the maximum length of the explicit message. Therefore, from an analysis point of view, no special treatment is needed for explicit communication. In the rest of the section, when we talk about private cache misses (typically L1 cache misses), it applies to both explicit and implicit communications.

3.2.3 TDMA-based WCET analysis

Performing worst-case execution time analysis with respect to a TDMA bus schedule requires not only the knowledge about the number of cache misses for a certain program path, but also their locations with respect to time. Hence, each memory access needs to be considered with respect to the bus schedule, granting access to the bus only during the slots belonging to the requesting core. Calculating the worst-case execution time has to be done with respect to the particular hardware architecture on which the task being analyzed is going to be executed. Factors such as the instruction set, pipelining complexity and caches must be taken into account by the analysis. For an application running on a compositional architecture (as described in Sec-

tion 2.1.2), the analysis can be divided into subproblems processed in a local fashion, for instance, computing the worst case latency from each microarchitectural component. Besides, we can be sure that the local worst-case always contributes to the worst-case globally.

For a predictable multi-core system with a shared communication structure, it is necessary to search through all feasible program paths and match each possible bus transfer to slots in the actual bus schedule, keeping track of exactly when a bus transfer is granted the bus in the worst case. This means that the execution time of a basic block will vary depending on when it is executed. Fortunately, for an application running on a compositional architecture, efficient search-tree pruning techniques dramatically reduce the search space, allowing for local analysis, just as for traditional WCET techniques.

For compositional architectures, the computation of bus delay can be attributed at WCET calculation phase, as shown in Figure 2.6 at section 2. Specifically, while computing the worst-case path, we can compute the additional communication delay for each potential cache miss (*i.e.* memory accesses that may potentially access the shared bus) and accumulate the overall delay in the final WCET calculation. In the subsequent section, we shall describe a simplified technique to integrate the bus delay into WCET calculation. This simplified technique is based on traversing the control flow graph of the program, along with loop unrolling. It is, however, important to note that implicit path enumeration (IPET) via ILP solver is generally used for WCET calculation. In Section 3.3.7, we shall describe the integration of bus modeling with the traditional IPET-based WCET calculation.

Multi-core WCET example

Consider a task τ executing on a system with two cores (core 1 and core 2). The task is being mapped on core 1, and has start time 0. First, an annotated control flow graph, as illustrated in Figure 3.11, is constructed. The rectangular elements $\mathbf{B}$, $\mathbf{C}$, $\mathbf{H}$, $\mathbf{E}$, $\mathbf{F}$ in the graph represent basic blocks, and the circles $\mathbf{A}$, $\mathbf{D}$, $\mathbf{G}$, $\mathbf{I}$ represent control nodes gluing them together. The loop starting at control node $\mathbf{G}$ will run at most three times, so the loop bound is consequently set to 3. The annotated numbers in the basic blocks represent consecutive cycles of execution, in the worst case, not accessing the bus. For instance, basic block $\mathbf{B}$ will, when executed, immediately – after 0 clock

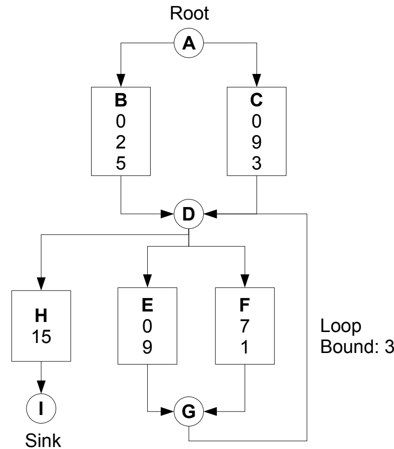


Figure 3.11: CFG example, the annotated numbers in each basic block capture consecutive cycles of execution, in the worst case, not accessing the bus

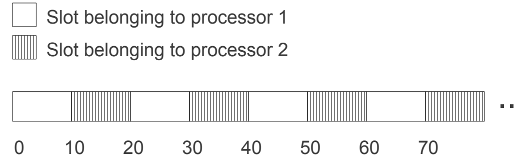


Figure 3.12: TDMA bus schedule example

cycles – issue a cache miss. After this, 2 cycles will be spent without bus accesses before the next (and last) cache miss occurs. Finally, 5 bus access-free cycles will be executed before the basic block ends. Hence, the execution time of basic block **B** will be $(0+k_1+2+k_2+5)$ where k_1 and k_2 represent the transfer times of the first and second cache miss respectively. Note that usually, loop unrolling is performed in order to decrease the pessimism of the analysis. This example is, however, purposely kept as simple as possible, and therefore the loop has not been unrolled.

For a typical single-core system, all cache misses take the same constant amount of time to process, and the execution time of basic block $\mathbf{B}$ would be known immediately. However, for multi-core architectures such as the one described in section 2, we must calculate the individual transfer times with respect to a given TDMA schedule.

Instead of a single core system, assume a multi-core system, as described in Section 3.2.1, using the bus schedule in Figure 3.12. Core 1, on which the task is running, gets a bus slot of size 10 processor cycles periodically assigned to it every 20th cycle. In this particular example, a cache miss takes 10 cycles for the bus to transfer, resulting in the bus being granted to Core 1 *only* at times t satisfying $t \equiv 0 \pmod{20}$, where $\equiv$ is the congruence operator.

Let us denote the worst-case start time of a basic block $\mathbf{Z}$ by $s(\mathbf{Z})$, and the end time in the worst case by $e(\mathbf{Z})$. The execution time of a basic block $\mathbf{Z}$, with respect to the worst-case start time, is then defined as $w(\mathbf{Z}) = e(\mathbf{Z}) - s(\mathbf{Z})$. Without considering bus conflicts, as in traditional methods, the worst-case execution time of the basic blocks would be $w_{\text{trad}}(\mathbf{B}) = 27, w_{\text{trad}}(\mathbf{C}) = 32, w_{\text{trad}}(\mathbf{E}) = 19, w_{\text{trad}}(\mathbf{F}) = 18$ and $w_{\text{trad}}(\mathbf{H}) = 15$. The corresponding worst-case program path becomes $\mathbf{C}, \mathbf{E}, \mathbf{E}, \mathbf{E}, \mathbf{H}$ resulting in a worst-case execution time of $27+19\cdot3+15 = 104$ clock cycles. However, this assumes that all cache misses take the same amount of time to transfer, and this is false in a multi-core system with a shared communication structure. In a TDMA-based approach, the execution time of a basic block depends on its start time in relation to the bus schedule. We can start from the root node and successively calculate the execution time of each basic block with respect to the worst-case start time. At the same time, the worst-case path is calculated.

With respect to the TDMA schedule in Figure 3.12, the worst-case start times of the basic blocks connected directly to the root node is 0, since they will never execute at any other time instant. The execution time of block $\mathbf{B}$, in the worst case, is $w(\mathbf{B}) = 0+10+2+18+5 = 35$ whereas the corresponding execution time of block $\mathbf{C}$ is $w(\mathbf{C}) = 0 + 10 + 9 + 11 + 3 = 33$. Note that $w(\mathbf{B}) > w(\mathbf{C})$, even though the relation is the opposite in the traditional case above where $w_{\text{trad}}(\mathbf{B}) < w_{\text{trad}}(\mathbf{C})$. In order to decide which one of these two basic blocks is on the critical path, two very important observations must be made based on the predictable nature of the TDMA bus (and the compositionality considered in this section).

1. *The absolute end time of a basic block can never increase by letting it start earlier*. That is, considering a basic block $\mathbf{Z}$ with $s(\mathbf{Z}) = x$ and $e(\mathbf{Z}) = y$, any start time $x' < x$ will result in an end time $y' \leq y$. The execution time of the particular basic block can increase, but the increment can never exceed the difference $x - x'$ in start time.

This means that a basic block $\mathbf{Z}$ will never end later than $e(\mathbf{Z})$, as long as it start before (or at) $s(\mathbf{Z})$. This guarantees that the worst-case calculations will never be violated, no matter what program path is taken. Note that $w(\mathbf{Z})$ is the execution time in the worst case, with respect to $e(\mathbf{Z})$, and that the time spent by executing $\mathbf{Z}$ can be greater than $w(\mathbf{Z})$ for an earlier start time than $s(\mathbf{Z})$.

2. Consider a basic block $\mathbf{Z}$ with worst-case start time $s(\mathbf{Z}) = x$ and worst-case end time $e(\mathbf{Z}) = y$. If we, instead, assume a worst-case start time of $s(\mathbf{Z}) = x''$ where $x'' > x$, the corresponding resulting absolute end time $e(\mathbf{Z}) = y''$ will always satisfy the relation $y'' \geq y$. This means that the greatest assumed worst-case start time $s(\mathbf{Z})$ will also result in the greatest absolute end time $e(\mathbf{Z})$.

Based on the second observation, we can be sure that the maximum absolute end time for the basic block ($\mathbf{E}$, $\mathbf{F}$ or $\mathbf{H}$) succeeding $\mathbf{B}$ and $\mathbf{C}$ will be found when the worst-case start time is set to 35 rather than 33. Therefore, we conclude that $\mathbf{B}$ is on the worst-case program path and, since they are not part of a loop, $\mathbf{B}$ and $\mathbf{C}$ do not have to be investigated again.

Next follow three choices. We can enter the loop by executing either $\mathbf{E}$ or $\mathbf{F}$, or we can go directly to $\mathbf{H}$ and end the task immediately. Due to observation 2 above, we can conclude that the worst-case absolute end time of $\mathbf{H}$, and thus the entire task, will be achieved when the loop iterates the maximum possible number of times, which is 3 iterations, since that will maximize $s(\mathbf{H})$. Therefore, the next step is to calculate the worst-case execution time for basic blocks $\mathbf{E}$ and $\mathbf{F}$ respectively for each of the three iterations, before finally calculating the worst-case execution time of $\mathbf{H}$. In the first iteration, the worst-case start time is $s(\mathbf{E_1}) = s(\mathbf{F_1}) = 35$ and the execution times become $w(\mathbf{E_1}) = 0 + 15 + 9 = 24$ and $w(\mathbf{F_1}) = 7 + 28 + 1 = 36$ for $\mathbf{E}$ and $\mathbf{F}$ respectively. We conclude that the worst-case program path so far is $\mathbf{B}$, $\mathbf{F}$ and the new start time is set to $s(\mathbf{E_2}) = s(\mathbf{F_2}) = 35{+}36 = 71$. In the second loop iteration, we get $w(\mathbf{E_2}) = 0 + 19 + 9 = 28$ and $w(\mathbf{F_2}) = 7 + 12 + 1 = 20$. Hence, in this iteration, $\mathbf{E}$ contributes to the worst-case program path and the new worst-case start time becomes $s(\mathbf{E_3}) = s(\mathbf{F_3}) = 99$. In the final iteration, the execution times are $w(\mathbf{E_3}) = 0 + 11 + 9 = 20$ and $w(\mathbf{F_3}) = 7 + 24 + 1 = 32$ respectively, resulting in the new worst-case start time $s(\mathbf{H}) = 131$. We now know that the worst-case program path is

$\mathbf{B}, \mathbf{F}, \mathbf{E}, \mathbf{F}, \mathbf{H}$, and since $\mathbf{H}$ contains no cache misses, and therefore always takes 15 cycles to execute, the WCET of the entire task is $e(\mathbf{H}) = 146$.

As shown in this example, in a loop-free control flow graph, each basic block has to be visited once. For control flow graphs containing loops, the number of investigations will be the same as for the case where all loops are unrolled according to their respective loop bounds.

3.2.4 Modeling both shared caches and shared buses

In the previous section, we have described the modeling of shared buses in isolation. In general, there might exist complex timing interactions between shared caches and shared buses. Such interactions may affect the overall schedulability analysis. Therefore, to accurately model the timing behavior in a multi-core system, it is critical to consider both shared caches and shared buses. In the following, we shall first give an overview of an integrated analysis framework which considers both shared caches and shared buses (previously proposed in [22]). We shall then illustrate the workflow of this analysis framework through an example. Also, in this section, we shall assume a *compositional* architecture (as explained in Section 2.1.2). As a result, the worst case memory latency (including bus delay) for each memory access instruction can simply be added to obtain the overall worst case memory latency.

Integrated analysis framework

The analysis framework in the presence of shared cache and bus in multi-core platforms appears in Figure 3.13. Such a framework estimates the worst case response time (WCRT) of an application containing several tasks. The application is captured by a set of *task graphs*. Each task graph is a directed acyclic graph containing a number of tasks. Each node in a task graph captures a specific task. Besides, a directed edge between task T_i and task T_j captures that task T_j can start only after task T_i finishes execution. In our discussion, task graphs are only used to show the dependency between shared cache and shared bus analyses. We only describe non-preemptive system. For preemptive systems, additional challenges, such as estimating the cache related preemption delay (CRPD) [6], need to be handled. Analysis of preemptive multi-tasking systems is also an active research topic, however, it is

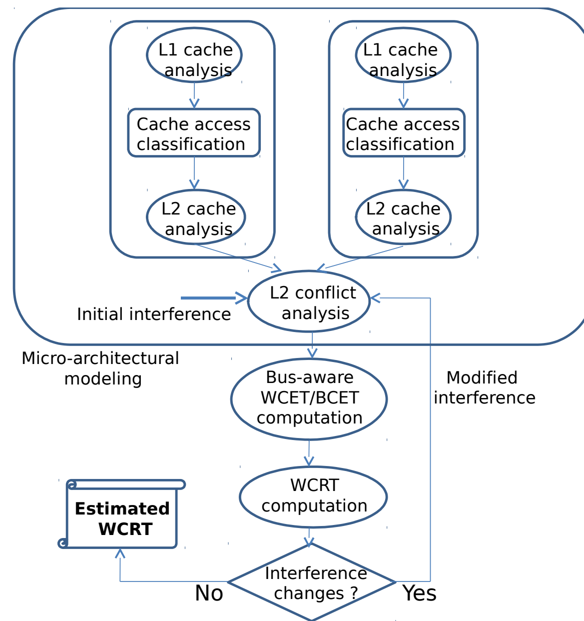


Figure 3.13: Iterative analysis framework

somewhat outside the scope of this monograph. Therefore, interested readers are referred to [6] and related literature. 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. Note that all cache analyses are performed as part of micro-architectural modeling (*cf.* Figure 2.1). For compositional architectures, worst case memory latency (including bus delay) can be computed during the WCET calculation phase (by traversing the program's control flow graph, as also explained in Section 3.2.3).

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 3.13. In the first iteration, the analysis of shared L2 cache assumes that a task on one core can conflict with all the tasks in other cores. Based on this pessimistic L2 cache analysis results, we can 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. It can be formally proved that such an analysis monotonically reduces the task interferences across iterations, and hence is guaranteed to terminate.

Illustrative Example We now show the working of the analysis using the example in Figure 3.14(a). We assume a 2-core system where the task graph containing tasks $T1$ and $T2$ is running on core 0 and task graph containing tasks $T3$ and $T4$ is running on core 1. For simplicity of exposition, we shall assume in this example that the *best-case* and the *worst-case* execution times of any task are the same. $T1.1, T2.1, \ldots, T4.2$ represent the memory blocks within the tasks. Each memory block is annotated with the required computation cycles excluding the memory/bus latency. 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. An initial L2 cache analysis is performed for each core individually that ignores conflicts from other cores. This per-core L2 cache analysis determines all the memory blocks ($T1.2, T2.2$, and $T4.2$) as guaranteed L2 cache hits. Let us also assume that the L2 cache hit latency is 10 cycles, whereas L2 cache miss latency is 20 cycles. Further,

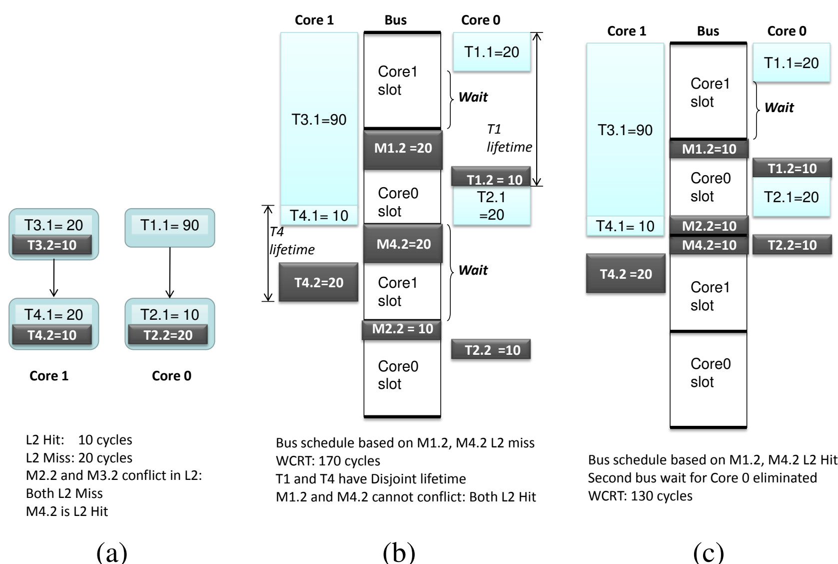


Figure 3.14: Example to show dependency between cache and bus analysis, annotations of the form $Tx.y$ capture different memory blocks accessed by task Tx. The quantitative value beside each annotation $Tx.y$ captures the computation cycle excluding the memory/bus latency. A bus transaction (if required) for a memory block $Tx.y$ is captured by the annotation $Mx.y$. The memory blocks colored in black (*e.g.* $T2.2$) capture potential L1 cache misses and hence, accessing the shared bus

the 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, the analysis does not rely on this assumption and it can accurately model 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 the analysis of the shared L2 cache. 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 $T1.2$ and $T4.2$ 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 $T2.2$ 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 $T2.2$ may not conflict with $T4.2$ since it maps to a different cache block in the shared L2 cache.

After shared L2 cache analysis, we proceed to shared bus analysis. The result of the analysis can be visualized in Figure 3.14(b). In Figure 3.14, 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. An L2 cache access from core i has to wait for core i to get access to the bus. The L1 cache miss $M4.2$ in core 1 occurs at time 100. From the bus schedule, we can observe that the slot beginning at time 100 belongs to core 1. Thus $M4.2$ does not encounter any additional waiting time to acquire the shared bus and is completed by time 120. Thus, $T4$ finishes at time 140. However, the L2 cache miss $M1.2$ in core 0 happens at time 20 and the bus slot from time 0 to time 50 is allotted to core 1. Hence, $M1.2$ encounters an additional 30 cycles waiting time to acquire the bus and eventually the memory transaction corresponding to $M1.2$ completes at time 70. This makes task $T1$ to finish at time 80. Similarly, the L2 cache hit $M2.2$ in core 0 occurs at time 100 and the bus slot from time 100 to time 150 is allotted to core 1. Thus $M2.2$ encounters an additional 50 cycles waiting time and eventually the task graph running on core 0 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 $T1$ and $T4$ have disjoint lifetimes. So memory blocks $T1.2$ and $T4.2$ 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 3.14(c). Task graph running on core 1 finishes at time 130 because $M4.2$ is now an L2 cache hit. Due to the earlier completion of $M1.2$ (because of L2 hit), L2 cache hit $M2.2$ occurs at time 90. Since L2 cache hit latency is 10 cycles, $M2.2$ can be serviced in the remaining bus slot belonging to core 0 (i.e., the bus slot from time 90 to time 100) and therefore making $T2$ finish by time 110. Hence, this new analysis results in a much tighter WCRT estimate as the second wait time for the bus in core 0 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.

Bus-aware WCET analysis without loop unrolling

In this section, we shall describe a different TDMA bus analysis technique which avoids the problem of loop unrolling for the analysis proposed in Section 3.2.3. Such an analysis is very efficient, however, the efficiency comes at a cost of analysis precision.

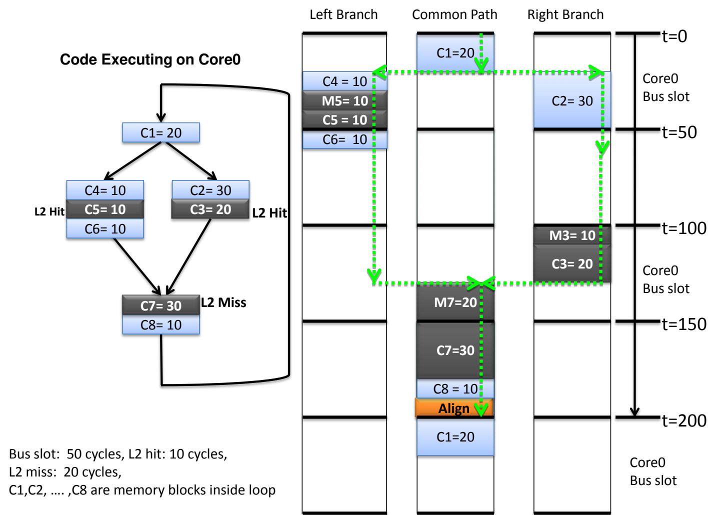
Recall that L1 cache misses are transmitted via the bus to access the shared L2 cache. Classical WCET analysis can compute the WCET of a program by taking into account only the number of worst case cache misses. The exact time-stamps of the cache misses (the time at which the cache misses occur) are not required for WCET computation. In presence of a shared 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, the effect of shared bus in WCET analysis can be ignored at the cost of obtaining highly over-estimated WCET values.

For the sake of simplicity in the following discussion, we shall assume that each core has been assigned the same slot length in a given bus schedule. For variable length slots, the following analysis methodology remains unchanged. However, all the equations in the rest of the section will become substantially more complex. Therefore, to convey the overall idea, we shall restrict our following description for a simplified bus schedule which assigns bus slots of the same length to each core. We shall discuss the optimization of more complex TDMA schedules (with variable slot lengths) in section 4.

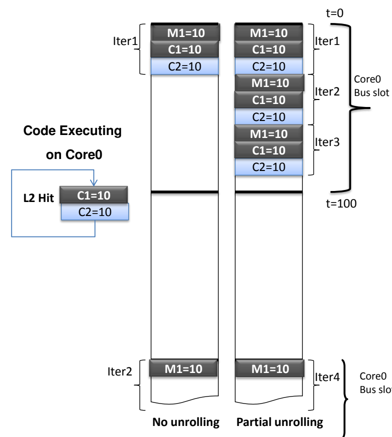
Formally, one can capture the bus schedule by the following mathematical relation:

$$CS_k^{(i)} = A_k + B \cdot i \tag{3.2}$$

where $CS_k^{(i)}$ is the starting time of the bus schedule assigned to $k\text{-}th$ core in $i\text{-}th$ round, $B = \mathcal{C} \times S_l$, $\mathcal{C}$ 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 $k\text{-}th$ core.



(a)



(b)

Figure 3.15: (a) An example of loop analysis, annotations of the form Cx capture different memory blocks accessed by the running task. The quantitative value beside each annotation Cx captures the required computation cycles excluding the memory/bus latency. A bus transaction (if required) for a memory block Cx is captured by the annotation Mx. The memory blocks colored in black (*e.g.* $C3$) capture potential L1 cache misses and hence, accessing the shared bus (b) Limited loop unrolling with low cost

At first we discuss the WCET computation of a single loop (no nesting) and later we extend it to a full program. Analysis of loops is depicted by an example in Figure 3.15(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 the 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 (as also explained in Section 3.2.3). This nice property is crucial for *compositional* architectures (*cf.* Section 2.1.2). Due to this property, the worst case starting time of each L1 cache miss can be taken into account for a *sound* WCET computation.

Formally, WCET computation of a loop is described in Algorithm 3.2. $start_{b_i}$ 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 25). The 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 8). For

Algorithm 3.2 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.   **else**
7.   find the predecessor p_{max} of b_i having maximum finish time ($finish_{p_{max}}$);
8. $start_{b_i} := finish_{p_{max}}$;
9.   **end if**
10.   *inst* := first instruction in basic block b_i;
11.   **repeat**
12.   **if** (*inst* is an L1 cache hit) **then**
13.   *cost* := *cost* + $L1_{lat}$; /* $L1_{lat}$: L1 cache hit latency */
14.   **else**
15. Δ := ($start_{b_i}$ + *cost*);
16.   *cost* := *cost* + Wait(Δ) + LAT;
17.   **end if**
18.   *inst* := next instruction in basic block b_i;
19.   **until** (all instructions in basic block b_i finish)
20.   **if** (b_i is the sink node of loop lp) **then**
21. Δ := ($start_{b_i}$ + *cost*);
22.   *cost* := *cost* + AlignCost(Δ);
23. $cost_{iter}$:= ($start_{b_i}$ + *cost*);
24.   **end if**
25. $finish_{b_i} := start_{b_i}$ + *cost*; /* finish time of b_i */
26. **end for**
27. **return** $cost_{iter} \times N + B$;

an L1 cache miss, function *Wait* computes the worst-case additional delay for accessing the shared bus (line 16).

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

Here Δ is the timepoint where the shared bus is accessed. S_l is the bus slot length assigned to each core. LAT is equal to the fixed L2 cache hit latency in case of an L2 cache hit and it is equal to main memory latency in case of

an L2 cache miss. The term $\lfloor \frac{\Delta}{B} \rfloor$ represents 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 within time Δ. The end time of this *latest* slot is at time $\lfloor \frac{\Delta}{B} \rfloor \times B + S_l$. To complete the L1 cache miss occurred at time Δ, 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 $(\lfloor \frac{\Delta}{B} \rfloor + 1) \times B$. Thus $(\lfloor \frac{\Delta}{B} \rfloor + 1) \times B - \Delta$ precisely captures the waiting time to acquire this next bus slot. 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 22). *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 time-line 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, the 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. Such loops can be partially unrolled so that worst-case cost of a single iteration of the unrolled loop exceeds one single bus slot. This situation is illustrated in Figure 3.15(b). The loop is unrolled three times as L1 cache misses (M1) 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 the analysis to whole programs, the program's control flow graph is transformed 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 3.2 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. This analysis is also context sensitive, i.e., procedure calls at different call sites are analyzed separately. Specifically, the cache analysis module can handle different contexts of a loop (i.e., *Virtual Inlining and Virtual Unrolling* (VIVU) approach [80]) and thus the shared bus analysis indeed can model different contexts of a loop. However, for the sake of simplicity, we restrict our description for the WCET analysis of a loop in a single context.

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).

In this discussion, we consider a non-preemptive system. Let us assume that *WCET(t)* and *BCET(t)* denote the worst-case execution time (WCET) and best-case execution time (BCET) of task t, respectively. 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 execu-

tion. Therefore, the following two equations can capture the computation of *Earliestfinish*(t) and *EarliestReady*(t):

$$EarliestFinish(t) = EarliestReady(t) + BCET(t)$$
$$EarliestReady(t) = \max_{u \in Pred(t)} EarliestFinish(u)$$

For a task t without any predecessor *EarliestReady(t)=0*. However, the latest finish time of a task is not only affected by its predecessors but also by the set of tasks running on the same core whose execution intervals may overlap (called *peers*) [56]. Let us call the set of tasks overlapping with t, and running on the same core by $\Re^t_{peers}$. Since the 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 the maximum alignment cost (equals B). Thus the *LatestFinish* time is defined as follows:

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

Here $|\Re^t_{peers}|$ captures the number of peers of task t. Intuitively, the term $(|\Re^t_{peers}| + 1) \times B$ captures the *worst-case cost* for a task t to be aligned to the beginning of a bus slot assigned to it. However, it is worthwhile to note that the WCRT computation is *sound* even in scenarios where tasks do not start at the boundary of a bus slot. The additional cost $(|\Re^t_{peers}| + 1) \times B$ is added only to perform the underlying WCET analysis in a simplified fashion (as described in Section 3.2.4).

It is important to note that the computed WCET and BCET already takes into account the shared cache interferences. These shared cache interferences are iteratively refined, as shown in Figure 3.13. Finally, for a given iteration in Figure 3.13, WCRT of an application is defined as follows:

$$\begin{aligned} WCRT = & \; max_t(LatestFinish(t)) \\ - & \; min_t(EarliestReady(t)) \end{aligned}$$

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

The iterative refinement of WCRT, as shown in Figure 3.13 works as follows. Initially a task t' cannot overlap (that is, *interfere*) with a task t if and only if *(i)* task t' depends on t and vice versa by the partial order imposed from the task graph or *(ii)* 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. The following two properties ensure that this WCRT analysis always terminates.

Property 3.2.4.1. For any task t, its EarliestReady(t) and BCET do not change across different iterations of L2 cache conflict and WCRT analysis.

Property 3.2.4.2. Task interferences monotonically decrease (strictly decrease or remain the same) across different iterations of the analysis framework (Figure 3.13).

3.2.5 Discussion

The analysis of buses described in 3.2.3 might be expensive in the presence of nested loops. This is due to virtual loop unrolling. The bus analysis described in 3.2.4 solves this problem by aligning loop iterations with bus schedules. However, such an efficiency in the analysis comes at a cost of reduced analysis precision. The work in [48] proposes an efficient TDMA-based bus analysis technique which avoids the problem of full loop unrolling, but it is almost as precise as the analysis described in Section 3.2.3. The analysis time in [48] significantly improves over the analysis time with full loop unrolling. For details, readers are referred to [48].

3.3 Modeling timing interactions

In the preceding discussion, we have described recent efforts in building timing models for shared caches and shared buses. We have also shown the existence of complex timing interactions between shared caches and buses and

effective ways to model such interactions. However, such solutions, as described in the preceding sections, have two important shortcomings as follows:

- They do not model timing interactions between shared resources and the rest of the micro-architecture (*e.g.* pipeline and branch prediction).
- They do not provide a full-fledged solution for architectures that may exhibit *timing anomaly* [59] (*i.e.* non-compositional architectures).

In this section, we shall describe a unified WCET analysis framework (the content has partially been published in [18] before) that solves the problems mentioned above.

3.3.1 Background

In this section, we introduce the background behind this unified WCET analysis framework. This WCET analysis framework for multi-core is based on the pipeline modeling of [54].

Pipeline modeling through execution graphs The central idea of pipeline modeling revolves around the concept of the *execution graph* [54]. 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 3.16 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.

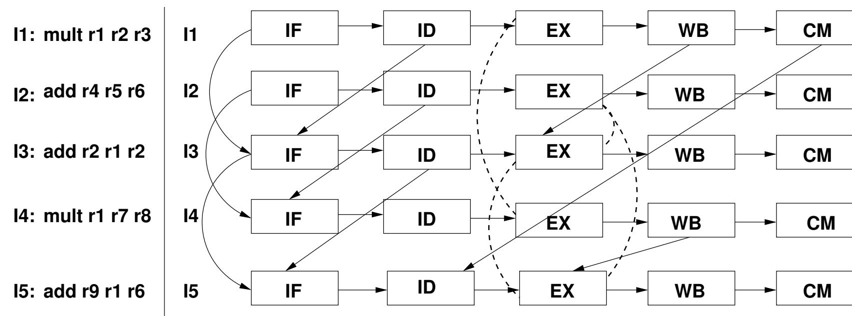


Figure 3.16: Execution graph for the example program in a 2-way superscalar processor with 2-entry instruction fetch queue and 4-entry reorder buffer. Solid edges show the dependency between pipeline stages, whereas the dotted edges show the contention relation

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 {(I2,I3), (I2,I5), (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. The WCET of the code snippet is then given by the worst case completion time of the CM node for I5.

3.3.2 Overview of the WCET analysis framework

Figure 3.17 gives an overview of the WCET analysis framework. Each processor core is analyzed, at a time, by taking care of the *inter-core conflicts* generated by all other cores. Figure 3.17 shows the analysis flow for some program A running on a dedicated processor core. Specifically, Figure 3.17 captures the instantiation of the general WCET analysis framework (discussed in Figure 2.1) for multi-core platforms. Basic analysis of caches is performed using the technique described in Section 3.1.2. However, to model the interaction between caches and branch predictor, L1 and L2 cache analysis has to consider the effect of *speculative execution* when a branch instruction is mispredicted (refer to Section 3.3.9 for details). Similarly, the timing effects generated by the mispredicted instructions are also taken into account during the iterative pipeline modeling (refer to [54] 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 3.3.4 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), microarchitectural 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 [53]). 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 3.3.7). *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 deriving the whole program's WCET via IPET.

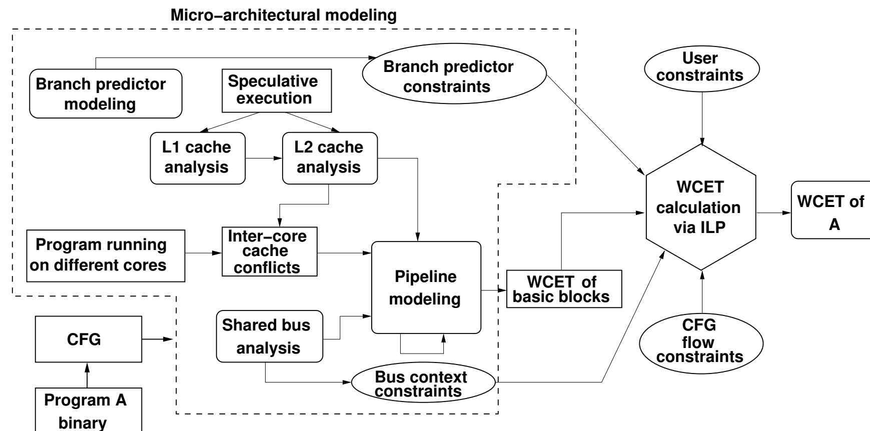


Figure 3.17: Instantiation of the WCET analysis framework shown in Figure 2.1 for multi-core platforms

System and application model We assume a multi-core processor as described in section 2. Therefore, each core has a private L1 cache. Additionally, multiple cores share an L2 cache. The extension of this 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 arbitration policy* (similar to the discussion in section 3), 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. Besides, we primarily discuss the effect of instruction caches. The extension for data caches are discussed in Section 3.6. Since we discuss only instruction caches, the cache miss penalty (computed from cache analysis) directly affects the *instruction fetch* (IF) stage of the pipeline. We also do not discuss the modeling of data prefetching units, which are usually available in modern processors. Therefore, we assume that the data prefetching units are disabled to improve timing predictability. We do not discuss *self modifying* code and therefore, we do not discuss the modeling of coherence traffic. Finally, we begin our discussion with *least-recently-used* (LRU) cache replacement policy and *non-inclusive* caches only. Later in Section 3.3.11, we

Algorithm 3.3 Outline of WCET analysis

1. Unroll each loop once to distinguish the execution context at first iteration
2. Perform analysis of caches via abstract interpretation
3. **for** (each basic block B in topological order) **do**
4. Model interaction of caches with pipeline stages (*cf.* Section 3.3.5)
5. Model interaction of shared buses with pipeline stages (*cf.* Section 3.3.6)
6. Model pipeline delays using iterative analysis (*cf.* Section 3.3.1)
7. **if** (B is the header node of loop L) **then**
8. Update flow graph G_L to add the bus context at the entry of L (*cf.* Section 3.3.4)
9. **end if**
10. **if** (B is the sink node of some loop L) **then**
11. Reevaluate the bus context O_L at the entry of L
12. **if** (O_L does not exist in G_L) **then**
13. Repeat from 5 in topological order for all basic blocks enclosed by L
14. **else**
15. update G_L to reflect the transition to bus context O_L
16. **end if**
17. **end if**
18. **end for**
19. Generate ILP constraints to predict the number of mispredicted branches
20. Generate ILP constraints from all flow graphs G_L for all loops L (*cf.* Section 3.3.4)
21. Generate CFG flow constraints to capture the structure of CFG
22. Solve the ILP problem to get the WCET (*cf.* Section 3.3.10)

discuss the extension of the framework for other cache replacement policies (*e.g.* FIFO and PLRU) and other cache hierarchies (*e.g.* inclusive).

In the following section, we shall first give an outline of the WCET analysis process. Subsequently, we shall describe the timing interaction of shared resources with pipeline. Specifically, we shall first describe such timing interactions within a single basic block. In next sections, this basic-block level analysis will be lifted to handle complex program flow structures, such as branches and loops.

3.3.3 Outline of WCET analysis algorithm

Algorithm 3.3 outlines the overall WCET analysis process. As discussed before, WCET analysis primarily works on the control flow graph (CFG) of a

given program. Each loop in the CFG is unrolled once to distinguish the first loop iteration from all other loop iterations. This limited unrolling helps both cache analysis (to distinguish cold cache misses) and the bus analysis (to distinguish the bus contexts outside of or within the loop). For each basic block, the pipeline analysis is carried out via the execution graph modeling (*cf.* Section 3.3.1). During the pipeline modeling, the outcome of cache analyses is used to compute the delay of different pipeline stages, such as instruction fetch (IF). Besides, the pipeline modeling also computes the set of bus contexts that may appear at the entry and the exit of each pipeline stage. These bus contexts are then used to accurately compute the bus delay suffered by each potential memory access.

During the pipeline modeling, basic blocks are traversed in topological order of the acyclic CFG (*i.e.* ignoring the backedges of the CFG). If a basic block is enclosed by loop(s), the pipeline modeling needs to be carried out repeatedly to distinguish different bus contexts within the loop. In particular, basic blocks within a loop have different WCETs for each bus context entering the loop. Potentially, there might be a huge number of bus contexts that may enter a loop. This, in turn, may lead to a full-fledged loop unrolling. To avoid this, the discussed analysis methodology approximates the potential bus contexts via limited loop unrolling. Specifically, the bus contexts that may enter a loop L, is approximated via a flow graph G_L. The flow graph G_L contains N nodes if the loop L is unrolled $N - 1$ times. Besides, G_L contains a backedge to approximate the bus contexts for all iterations beyond N. The flow graph G_L is also used to bound the execution count of different bus contexts with which the loop L might be executed. These bounds on execution counts are specified as ILP constraints. These ILP constraints are linked up with the CFG structural constraints and ILP constraints to bound mispredicted branches. Finally, all such ILP constraints are used to formulate the WCET analysis as maximizing the objective function of an ILP problem.

3.3.4 Iterative modeling of pipeline latency

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 performed in an iterative fashion. 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 iterative 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. These timing intervals are used for computing the WCET of basic blocks. 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.

3.3.5 Interaction of shared caches with pipeline

Let us assume $CHMC_i^{L1}$ denotes the AH/AM/NC cache *hit-miss* classification of an IF node i in L1 cache. Similarly, $CHMC_i^{L2}$ captures the AH/AM/NC cache *hit-miss* classification of an IF node i in the shared L2 cache. Further assume that E_i denotes the possible latencies of an *IF* node i

without considering any shared bus delay. Using the preceding notations, E_i can be defined as follows:

$$E_i = \begin{cases} 1, \ \text{if } CHMC_i^{L1} = AH; \\ LAT^{L1} + 1, \quad \text{if } CHMC_i^{L1} = AM \wedge CHMC_i^{L2} = AH; \\ LAT^{L1} + LAT^{L2} + 1, \quad \text{if } CHMC_i^{L1} = AM \wedge CHMC_i^{L2} = AM; \\ [LAT^{L1} + 1, LAT^{L1} + LAT^{L2} + 1], \quad \text{if } CHMC_i^{L1} = AM \\ \qquad \wedge CHMC_i^{L2} = NC; \\ [1, LAT^{L1} + 1], \quad \text{if } CHMC_i^{L1} = NC \wedge CHMC_i^{L2} = AH; \\ [1, LAT^{L1} + LAT^{L2} + 1], \ \text{otherwise.} \end{cases} \tag{3.3}$$

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. In the next section, we show the interaction of shared buses with the pipeline.

3.3.6 Interaction of shared buses with pipeline

For the sake of clarity, we shall assume that each core has been assigned the same slot length for a given TDMA bus schedule. For variable length slots, the analysis methodology does not change, however, the equations described in the following become more complex in terms of readability. Therefore, to give the general idea, we shall discuss bus slots (assigned to each core) are the same in terms of length. We shall discuss the optimization of more complex TDMA schedules (with variable slot lengths) in section 4.

Let us assume that we have a total of $\mathcal{C}$ cores and the TDMA-based 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 section:

Definition 3.1. (*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 3.2. (*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 can 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:

$$latest[t_i^{finish}] = latest[t_i^{start}] + max_lat_p(O_i^{in}, E_i) \tag{3.4}$$

$$earliest[t_i^{finish}] = earliest[t_i^{start}] + min_lat_p(O_i^{in}, E_i) \tag{3.5}$$

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:

$$max_lat_p(O_i^{in}, E_i) = \begin{cases} 1, & \text{if } CHMC_i^{L1} = AH; \\ \max\limits_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t), & \text{otherwise.} \end{cases} \tag{3.6}$$

$$min_lat_p(O_i^{in}, E_i) = \begin{cases} 1, & \text{if } CHMC_i^{L1} \neq AM; \\ \min\limits_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t), & \text{otherwise.} \end{cases} \tag{3.7}$$

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 3.3). 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 [22] or [48]):

$$\Delta_p(o,t) = \begin{cases} t, & \text{if } pS_l \leq o + t \leq (p+1)S_l; \\ t + pS_l - o, & \text{if } o < pS_l; \\ t + (\mathcal{C}+p)S_l - o, & \text{otherwise.} \end{cases} \tag{3.8}$$

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 can conservatively assume the absence of contention. Therefore, for a particular core p ($0 \leq p < \mathcal{C}$), we can compute O_i^{out} from the value of O_i^{in} as follows:

$$O_i^{out} = \begin{cases} u(O_i^{in}, E_i + [0, latest[t_i^{start}] - latest[t_i^{ready}]]), & \text{if } i = EX; \\ u(O_i^{in}, \bigcup_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t)), & \text{if } i = IF; \\ u(O_i^{in}, E_i), & \text{otherwise.} \end{cases} \tag{3.9}$$

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

$$u(O, X) = \bigcup_{o \in O, t \in X} \{(o + t) \ mod \ S_l \mathcal{C}\} \tag{3.10}$$

The iterative pipeline modeling refines the worst-case contention suffered by node i. This refinement approximates the overlap between EX stages using the overlap in timing interval $[earliest[t_i^{ready}], latest[t_i^{finish}]]$. Finally, the worst-case contention suffered by node i is captured in the quantity $latest[t_i^{start}] - latest[t_i^{ready}]]$. Therefore, $E_i + [0, latest[t_i^{start}] - latest[t_i^{ready}]]$ captures all possible latencies suffered by the execution graph node i, taking care of contentions as well. Consequently, 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 3.9 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 $pred(i)$ de-

notes 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 dependencies). 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, a 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:

$$O_i^{in} = \bigcup \{ O_j^{out} \mid j \in pred(i) \wedge earliest[t_{pmax}^{finish}] \leq latest[t_j^{finish}] \} \quad (3.11)$$

where $pmax$ is any predecessor of i that satisfies the condition $latest[t_{pmax}^{finish}] = \max_{j \in pred(i)} latest[t_j^{finish}]$. 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 3.11 gives an over-approximation of the incoming bus context at node i. Finally, Equation 3.9 and Equation 3.11 together ensure a sound computation of the bus contexts at the entry and exit of each execution graph node.

3.3.7 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. The set of instructions before (after) B which directly affect the execution time of B is called the *prologue* (*epilogue*) of B [54]. B may have multiple prologues and epilogues 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. In particular, the number of instructions

which can be in the pipeline, when B enters the pipeline, is bounded by the total size of IFQ and ROB. Similarly, the number of instructions after B, which can contend with instructions in B, is bounded by the size of 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 3.3.6). After the analysis of B is completed for all possible combinations of prologues and epilogues, we can compute an *over-approximation* of O_i^{in} (O_i^{out}) by *merge* operation as follows:

$$O_i^{in} = \bigcup_{p,e} O_i^{in}(p,e) \tag{3.12}$$

$$O_i^{out} = \bigcup_{p,e} O_i^{out}(p,e) \tag{3.13}$$

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.

To effectively compute the TDMA offsets, basic blocks are analyzed in *topological order* (not accounting the back edges). As a result, before computing the bus contexts of a basic block, the bus contexts of its prologues are computed at least once. Therefore, the set of bus contexts within a basic block can be computed *precisely* by propagating the bus contexts computed at its prologues.

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, two sets of nodes is computed — 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

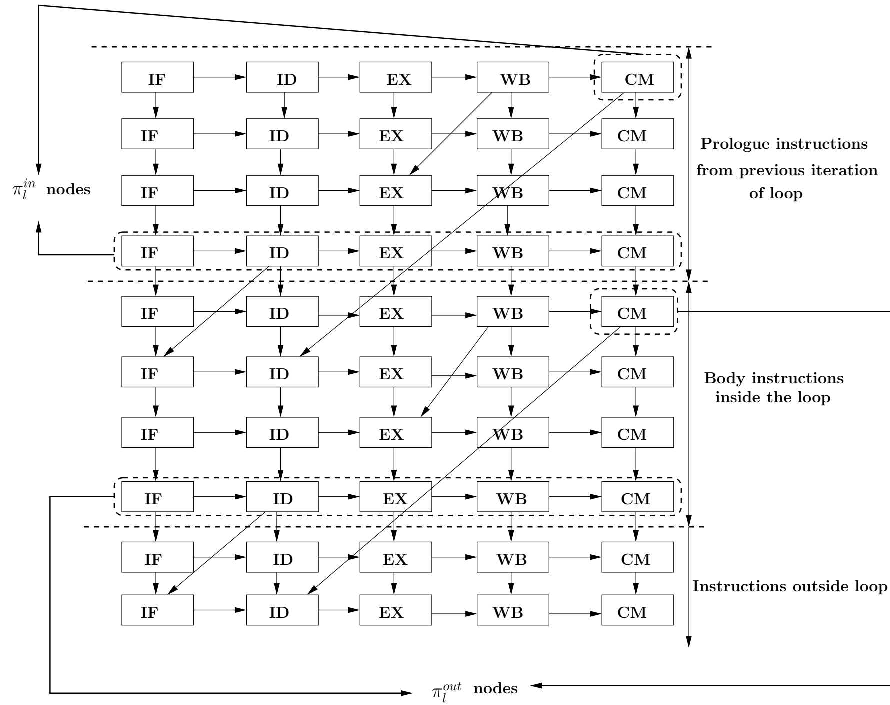


Figure 3.18: π_l^{in} and π_l^{out} nodes shown with the example of a sample execution graph. π_l^{in} nodes propagate bus contexts across iterations, whereas, π_l^{out} nodes propagate bus contexts outside of loop.

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 3.18 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 3.3.6). 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.

3.3.8 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 3.18. 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 3.3.6.

In this section, we 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:

$\mathbf{\Omega_l}$: The set of all bus contexts that may reach loop l in any iteration.

$\mathbf{\Omega_l^s}$: 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 3.3.6. If l is an inner loop, an element of Ω_l^s is computed (as described in Section 3.3.6) for each analysis invocation of the loop immediately enclosing l.

$\mathbf{G_l^s}$: The flow graph capturing the transition of different bus contexts. For each $s_0 \in \Omega_l^s$, a *flow graph* $G_l^s = (V_l^s, F_l^s)$ is constructed, where $V_l^s \subseteq \Omega_l$. The graph G_l^s captures the transitions among different bus contexts across loop iterations. An edge $f_{w_1 \to w_2} = (w_1, w_2) \in F_l^s$ 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 many bus contexts that are the nodes of G_l^s.

$\mathbf{M_l^w}$: The number of times the body of loop l is entered with bus context $w \in \Omega_l$ in any iteration.

$\mathbf{M_l^{w_1 \to w_2}}$: The 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 $f_{w_1 \to w_2} \notin F_l^s$ for any flow graph G_l^s, $M_l^{w_1 \to w_2} = 0$.

Construction of G_l^s For each loop l and for each $s_0 \in \Omega_l^s$, a flow graph G_l^s is constructed. 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 3.3.6), 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 can add an edge from s_0 to s_1 in G_l^s. We can 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, the construction of G_l^s can be terminated by adding a backedge from s_{n-1} to s_k. We can 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 experiments, it was observed 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 can avoid the computationally intensive unrolling of every loop.

Generating separate ILP constraints Using each flow graph G_l^s for loop l, ILP constraints are generated 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:

$$\forall w \in \Omega_l : \sum_{x \in \Omega_l} M_l^{x \to w} = M_l^w \tag{3.14}$$

$$\forall w \in \Omega_l : M_l^w - 1 \leq \sum_{x \in \Omega_l} M_l^{w \to x} \leq M_l^w \tag{3.15}$$

$$\sum_{w \in \Omega_l} M_l^w = N_{l.h} \tag{3.16}$$

where $N_{l.h}$ denotes the number of times the header of loop l is executed. Equations 3.14-3.15 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 3.15 takes this special case into consideration. On the other hand, Equation 3.16 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 [80]. Equations 3.14-3.16 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 can add the following ILP constraints to ensure this:

$$\forall w \in \Omega_l^s : M_l^w \geq N_{l.h}^{w.h} \tag{3.17}$$

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 3.14-3.17 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:

$$\forall G_l^s = (V_l^s, E_l^s) : \sum_{w \in V_l^s} M_l^w \leq bound_l * (\sum_{w' \in parent(G_l^s)} M_{l'}^{w'}) \tag{3.18}$$

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 3.18 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 3.18.

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

$$\sum_{w \in \Omega_l} N_i^{w.i} = N_i \tag{3.19}$$

$$\forall w \in \Omega_l : \; N_i^{w.i} \leq M_l^w \tag{3.20}$$

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 3.20 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 ([48]) has shown the computation of O_l^{exit} without a full loop unrolling. We can use a similar technique as in [48] with one important difference: In [48], 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 3.18). 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 [48], 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 [48].

3.3.9 Effect of branch prediction

The presence of branch prediction introduces additional complexity in the WCET computation. If a conditional branch is mispredicted, the timing due to the mispredicted instructions needs 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 the 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 the instruction fetch queue and reorder buffer.

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 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 the instruction fetch queue and reorder buffer. To maintain a *safe* cache state at the entry of each basic block i, we can join the two cache states arising due to the *correct* and *incorrect* predictions of conditional edge $j \rightarrow i$. We demon-

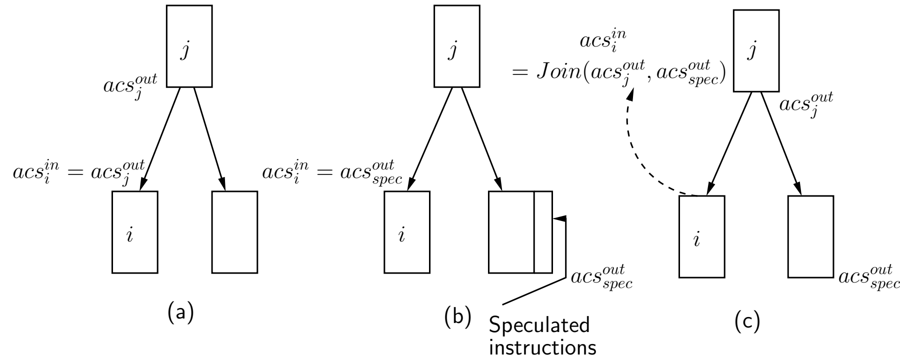


Figure 3.19: (a) Computation of acs_i^{in} when the edge $j \to i$ is correctly predicted, (b) Computation of acs_i^{in} when the edge $j \to i$ is mispredicted, (c) A *safe* approximation of acs_i^{in} by considering both correct and incorrect prediction of edge $j \to i$.

strate the entire scenario through an example in Figure 3.19. In Figure 3.19, 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 can combine two different possibilities —- one when the respective branch is *correctly* predicted (Figure 3.19(a)) and the other when the respective branch is *incorrectly* predicted (Figure 3.19(b)). The combination is performed through an abstract *join* operation (as shown in Figure 3.19(c)), 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.

Effect on bus for speculative execution

Due to branch misprediction, some additional instructions might be fetched from the mispredicted path. As described in Section 3.3.7, 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 ([54]). Since the propagation of *bus context* is entirely performed on the execution graph (as shown in Sec-

tion 3.3.6), the 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.

Computing the number of mispredicted branches

In the presence of a branch predictor, each conditional edge $j \to i$ in the program CFG can be correctly or incorrectly predicted. Let us assume $E_{j \to i}$ denotes the total number of times control flow edge $j \to i$ is executed and $E^c_{j \to i}$ ($E^m_{j \to i}$) denotes the number of times the control flow edge $j \to i$ is executed due to correct (incorrect) branch prediction. Clearly, $E_{j \to i} = E^c_{j \to i} + E^m_{j \to i}$. The value of $E_{j \to i}$ is further bounded by CFG structural constraints. On the other hand, values of $E^c_{j \to i}$ and $E^m_{j \to i}$ depend on the type of branch predictor. ILP-based techniques, such as the one proposed in [53] can be used to predict the bound on $E^c_{j \to i}$ and $E^m_{j \to i}$. The ILP constraints generated on $E^c_{j \to i}$ and $E^m_{j \to i}$ are as well captured in the global ILP formulation to compute the whole program WCET. We exclude here the details of branch predictor modeling — interested readers are referred to [53].

3.3.10 WCET computation of an entire program

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

$$Maximize\, T = \sum_{i=1}^{N} \sum_{j \to i} \sum_{w \in \Omega_i} t^{c,w}_{j \to i} * E^{c,w}_{j \to i} + t^{m,w}_{j \to i} * E^{m,w}_{j \to i} \qquad (3.21)$$

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 3.3.7). $t^{c,w}_{j \to i}$ denotes the WCET of basic block i when the basic block i is reached from basic block j, the control flow edge $j \to i$ is correctly predicted and i is reached with bus context $w \in \Omega_i$. Similarly, $t^{m,w}_{j \to i}$ denotes the WCET of basic block i under the same bus context but when the control flow edge $j \to i$ was mispredicted. Note that both $t^{c,w}_{j \to i}$ and $t^{m,w}_{j \to i}$ are computed during the iterative pipeline modeling (with the modifications proposed in Section 3.3.4).

$E^{c,w}_{j \to i}$ ($E^{m,w}_{j \to i}$) denotes the number of times basic block i is reached from basic block j with bus context w and when the control flow edge $j \to i$ is correctly (incorrectly) predicted. Therefore, we have the following two constraints:

$$E^{c}_{j \to i} = \sum_{w \in \Omega_i} E^{c,w}_{j \to i}, \;\; E^{m}_{j \to i} = \sum_{w \in \Omega_i} E^{m,w}_{j \to i} \tag{3.22}$$

Constraints on $E^{c}_{j \to i}$ and $E^{m}_{j \to i}$ are proposed by the ILP-based formulation in [53]. On the other hand, $E^{c,w}_{j \to i}$ and $E^{m,w}_{j \to i}$ are bounded by the CFG structural constraints ([80]) and the constraints proposed by Equations 3.14-3.20 in Section 3.3.7. Note that in Equations 3.14-3.20, 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 the analysis framework for an IPET implementation.

Finally, the WCET of the program maximizes the objective function in Equation 3.21. Any ILP solver (*e.g.* CPLEX) can be used for maximizing the objective function in Equation 3.21.

3.3.11 Extension of shared cache modeling

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 [67]. Recently, abstract interpretation based analysis of FIFO replacement policy has been proposed in [33, 34] for single level caches and for multi-level caches in [41]. In this section, we shall discuss the extension of the 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).

Review of cache analysis for FIFO replacement

We can use the *must cache analysis* for FIFO replacement as proposed in [33]. 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 ([33]) as follows:

$$\mathcal{A}_s = [T_1, T_2, \ldots, T_k] \tag{3.23}$$

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:

$$\tau([T_1, T_2, \ldots, T_k], tag_i) = \begin{cases} [T_1, T_2, \ldots, T_k], & \text{if } tag_i \in \bigcup_i T_i; \\ [\{tag_i\}, T_1, \ldots, T_{k-1}], & \text{if } tag_i \notin \bigcup_i T_i \\ & \quad \wedge |\bigcup_i T_i| = k; \\ [\phi, T_1, \ldots, T_{k-1} \cup \{tag_i\}], & \text{otherwise.} \end{cases} \tag{3.24}$$

The first scenario captures a *cache hit* and the second scenario captures a *cache miss*. The third scenario appears when the static analysis cannot accurately determine the *hit-miss* categorization of the memory reference. It is worthwhile to mention that the analysis of FIFO caches can be greatly improved using the information from *may analysis*, as also shown in [33]. Precise analysis of different cache replacement policies (including FIFO caches), although interesting and challenging, is outside the scope of this monograph. Therefore, interested readers are referred to [33] and related literature for further details.

The *abstract join* function for the FIFO must cache analysis is exactly the 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 abstract cache sets, the *right most* relative position of the cache tag is captured after the join operation.

Analysis of shared cache with FIFO replacement

To analyze the shared cache, we can use the technique described in Section 3.1.2. Recall that 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 the shared cache. More precisely, for an N-way set-associative shared cache, m might be replaced due to inter-core conflicts if the following condition holds:

$$N - AGE_{fifo}(m) < |\mathcal{M}_c(m)| \tag{3.25}$$

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. Using this notion, the shared cache update function can now be defined as follows.

$$\tau([T_1, T_2, \ldots, T_k], tag_i) = \begin{cases} [T_1, T_2, \ldots, T_k], \text{ if } tag_i \in \bigcup_i T_i \\ \wedge N - AGE_{fifo}(tag_i) \geq |\mathcal{M}_c(tag_i)| \\ \\ [\{tag_i\}, T_1, T_2 \ldots, T_{k-1}], \\ \text{if } tag_i \notin \bigcup_i T_i \wedge |\bigcup_i T_i| = k; \\ \\ [\phi, T_1 \setminus \{tag_i\}, T_2 \setminus \{tag_i\}, \ldots, T_{k-1} \cup \{tag_i\}], \\ \text{otherwise.} \end{cases} \tag{3.26}$$

Timing interaction with FIFO caches with pipeline and branch predictor

As described before, 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 adds computation cycles with the instruction fetch (IF) stage. Therefore, we use Equation 3.3 to compute the latency suffered by cache hit/miss and propagate the latency through different pipeline stages.

Recall from Section 3.3.9 that speculative execution may introduce additional cache conflicts. In Section 3.3.9, we show the modification of abstract interpretation based cache analysis to handle the effect of speculative execution on cache. From Figure 3.19, we observe that the solution is *independent* of the cache replacement policies concerned. Therefore, the modification due to speculative execution for FIFO replacement policy is exactly the same. We can perform an *abstract join* operation on the cache states along the *correct* and *mispredicted* path (as shown in Figure 3.19). However, for FIFO replacement polices, the abstract join operation is performed according to the FIFO replacement analysis (instead of the LRU join operation we performed in case of LRU caches).

Other cache organizations

In the preceding, we have discussed the extension of the 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, the shared cache conflict analysis can be integrated. As a result, the 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 [35]). Cache analysis based on *relative competitiveness* [67] 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 the WCET analysis framework. Nevertheless, more precise analysis than the ones based on relative competitiveness can be designed, as shown in [35] for PLRU policy. However, description of such precise cache analysis is outside the scope of this monograph. The purpose of this section is to describe a unified WCET analysis framework for multi-core processors and any precision gain in the

existing cache analysis technique will directly benefit the framework by improving the precision of WCET prediction.

In this section, 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 size of the smaller caches. Therefore, processor architects sometimes resort to non-inclusive cache hierarchies [88]. 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 [41] for multi-level private cache hierarchies). The analysis in [41] 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). The shared cache conflict analysis phase can be applied on this reduced set of AH categorized memory references for inclusive caches, keeping the rest of the 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 the WCET analysis framework unchanged.

3.4 Discussion about analysis complexity

In the preceding sections, we have discussed some comprehensive proposals for analyzing the WCET on multi-core platforms. In the following discussion, we shall consider the complexity of such analysis techniques.

Pipeline modeling

Pipeline modeling revolves around the traversal of the *execution graph*. This execution graph is constructed for each basic block. The timing information of each node in the execution graph is computed iteratively. For each such iteration, the complexity of traversing the execution graph is $O(|V| + |E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in the execution graph. The number of nodes $|V|$ is at least $|S| \cdot |I|$, where $|S|$ is the number of pipeline stages and $|I|$ is the number of instructions in the

basic block. Besides, the size of the execution graph depends on the following factors:

- The sizes of instruction fetch queue (IFQ) and reorder buffer (ROB) increase the number of nodes in the execution graph. In particular, the size of IFQ and ROB increases the execution context of a basic block. As a result, the complexity of pipeline modeling also increases.

- The size of the execution graph may increase with the amount of data dependencies in the code. In particular, each data dependency corresponds to an edge in the execution graph. Therefore, the complexity of pipeline modeling also depends on the amount of data dependencies within a basic block.

- Finally, the size of the execution graph might be increased due to several factors influencing instruction-level parallelism. Such factors include out-of-order and superscalar processors and speculative execution, among others.

Cache modeling

The complexity of cache modeling directly depends on the number of cache lines tracked for analysis. The number of cache lines increases with the size of caches. As a result, the complexity of cache analysis also increases with the size of caches in the system. Besides, the abstract join operations at control flow merge points perform either *union* or *intersection* of different sets. As a result, the complexity of the abstract join operation is proportional to the complexity of *set union* and *set intersection*.

Shared bus modeling

The shared bus modeling revolves around tracking different TDMA offsets. These TDMA offsets are tracked to accurately compute the memory latency. Besides, ILP constraints are generated to accurately compute the set of TDMA offsets at different program locations. Since the goal is to compute the maximum memory latency, the complexity of computing memory latency is proportional to the number of TDMA offsets. Moreover, the number of ILP constraints (as formulated in Section 3.3.8) may increase proportionally with

the number of TDMA offsets. Therefore, the complexity of shared bus analysis heavily depends on the number of tracked TDMA offsets. The number of TDMA offsets is proportional to the bus slot length allocated to each core and the total number of cores. The bus slot length allocated to each core is typically small and it does not substantially influence the complexity of shared bus analysis. In contrast, the complexity of shared bus analysis may significantly increase with increasing number of cores, due to a substantial number of TDMA offsets to be tracked. However, this increase in the analysis complexity has a trade-off with decreased analysis pessimism. For instance, an analyzer can capture a set of TDMA offsets via an *interval*, instead of tracking each individual TDMA offset in a *set*. Such an abstraction will greatly simplify the analysis complexity. However, note that abstractions via *intervals* may potentially capture *spurious* TDMA offsets, which can never appear in any real execution. This, in turn, will increase the pessimism in the overall WCET analysis.

Modeling branch prediction and speculation

The presence of speculative execution may increase the size of the execution graph, which, in turn increases the complexity of pipeline modeling. This increase in the complexity highly depends on the *depth of speculation*. The *depth of speculation* is defined as the number of instructions that can be issued in the pipeline in the presence of an unresolved branch instruction. Apart from the speculation-depth, the size and type of branch predictor may substantially increase the number of ILP constraints generated by the discussed analysis techniques. For instance, the presence of complex branch predictors (*e.g.* `gshare`) may generate more ILP constraints than simple two-bit branch predictors. As a result, the complexity of the WCET computation may increase. However, the complexity of ILP-based branch predictor modeling can be reduced by using less precise but more efficient analysis techniques, such as *abstract interpretation*.

To empirically understand the impact of different analysis complexities, this monograph includes a discussion in Section 3.5.5, where we provide empirical comparisons to discuss the analysis scalability with respect to different micro-architectural parameters. In particular, we show the analysis complex-

Table 3.1: Salient features of the benchmarks used in evaluation

Benchmark	*Lines of code*	*Code size (in bytes)*
matmult	163	968
cnt	133	840
fir	275	584
fdct	238	2232
expint	168	824
qurt	158	1368
nsichneu	4266	38344
bs	114	408
crc	128	1936
fibcall	72	288
janne_complex	64	264
lcdnum	64	272
minver	201	1592
prime	47	208
select	114	3120
sqrt	77	336
fft	210	576
edn	283	4392
ludcmp	147	1592
ns	531	392
ndes	238	3816
bsort100	127	440
adpcm	828	6664
st	157	1880
jfdctint	374	2856
statemate	1273	9464

ity with respect to different pipeline structures, cache sizes and the presence of speculative execution.

3.5 Experimental evaluation

3.5.1 Experimental setup

The evaluation of the WCET analysis framework in this section uses benchmarks from [37], which are generally used for timing analysis. Some salient features of these benchmarks are listed in Table 3.1.

Individual benchmarks are compiled for simplescalar PISA (Portable Instruction Set Architecture) [10] — a MIPS like instruction set architecture. Simplescalar gcc cross compiler is used 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 the analysis framework. In the current implementation of the framework, the analysis frontend (CFG extractor) and the modeling of pipeline do not appropriately handle `recursions`, `switch` cases and unstructured `goto`, `break` statements inside loops. Such programs from [37] are therefore not included in this evaluation.

To validate the analysis framework, the simplescalar toolset [10] 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, it is worthwhile to point out that the presence of a shared cache and a shared bus makes the realization of the worst-case scenario extremely 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 the following experiments may highly under-approximate the *actual WCET*.

For all the experiments, the WCET overestimation ratio is presented, 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, the analysis uses the default system configuration in Table 3.2 (as shown by the column “Default settings“). Since the data cache modeling is not yet included in the current implementation, all data accesses are assumed to be *L1 cache hits* (for analysis and measurement both). Besides, cache sizes in the default setting are chosen in a fashion to be comparable with the code size mentioned in Table 3.1.

Two different tasks are used 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 all experiments (Figures 3.21-3.22), the task `jfdctint` is used to generate inter-core conflicts to the first half of the tasks

Table 3.2: 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*

(*i.e.* `matmult` to `lcdnum`). On the other hand, the task `statemate` is used to generate inter-core conflicts to the second half of the tasks (*i.e.* `minver` 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 can expect a better WCET overestimation ratio for the first half of the benchmarks (*i.e.* `matmult` to `lcdnum`) compared to the second half (*i.e.* `minver` to `st`).

To measure the WCET overestimation due to *cache sharing*, we can 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 the 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, one can selectively turn off the analysis of different micro-architectural components. We say that an analysis of a micro-architectural component is *turned off*, if the

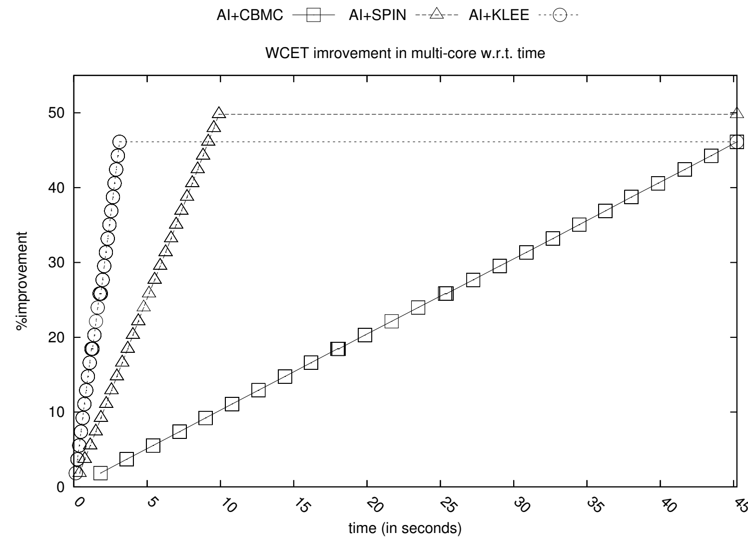


Figure 3.20: WCET improvement w.r.t. time using `statemate` as the conflicting task

respective micro-architectural component has *perfect setting* (refer to the column "Perfect settings" in Table 3.2).

3.5.2 Basic analysis result

Improvement of precision in shared cache modeling Figure 3.20 captures the evaluation of the shared cache modeling presented in Section 3.1.2. Specifically, it shows the improvement in WCET precision (with respect to analysis time) over the baseline abstract interpretation. Two different model checkers are used for experiment – SPIN [75] and CBMC [25]. SPIN is an linear time temporal logic (LTL) based model checker. SPIN can be used as an exhaustive verifier to check the assertion properties introduced by the technique discussed in Section 3.1.2. CBMC formally verifies program through bounded model checking [24]. In the WCET analysis framework, CBMC is used to check the assertion properties. For symbolic execution, KLEE [2] toolkit is used to explore the assertions.

Figure 3.20 shows the average improvement in WCET when task `statemate` is used to generate inter-core conflicts. Due to the *anytime* na-

ture, a provably correct WCET can be obtained from any vertical cut along the time axis in Figure 3.20. Nevertheless, if the refinement process is allowed more time to run, better precision in WCET can be obtained. We exclude the detailed evaluation of the shared cache modeling here. Interested readers are referred to [21] for detailed evaluation.

Effect of caches Figure 3.21(a) 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 the WCET overestimation ratio has reasonable bound except for a few benchmarks (*e.g.* `qurt`, `nsichneu`, `lcdnum`, `select`). The major source of this overestimation is the presence of many *infeasible* paths in such programs, which may lead to infeasible micro-architectural states and WCET overestimations. These infeasible paths can be eliminated by providing additional user constraints into the analysis framework and hence improving the ILP-based WCET calculation. We can 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 programs such as `adpcm`, `ndes` and `edn`, where the overestimations in the presence of shared L2 caches are higher than the same using partitioned L2 caches. This is due to the difficulty in modeling the inter-core cache conflicts from programs being run in parallel (*i.e.* `jfdctint` and `statemate`).

Effect of speculative execution As we explained in Section 3.3.9, 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 3.21(b) and Figure 3.22(a) show the effect of speculation in L1 and L2 cache, respectively. `qurt` and `ndes` show reasonable increases in the WCET overestimations in the presence of speculation (Figure 3.21(b) and Figure 3.22(a)). A similar increase in the WCET overestimation is also observed with `bs` and `sqrt` in the presence of L1 caches and speculation (Figure 3.21(b)). Such an increase in the overestimation ratio can be explained from the overestimation arising in the modeling of the ef-

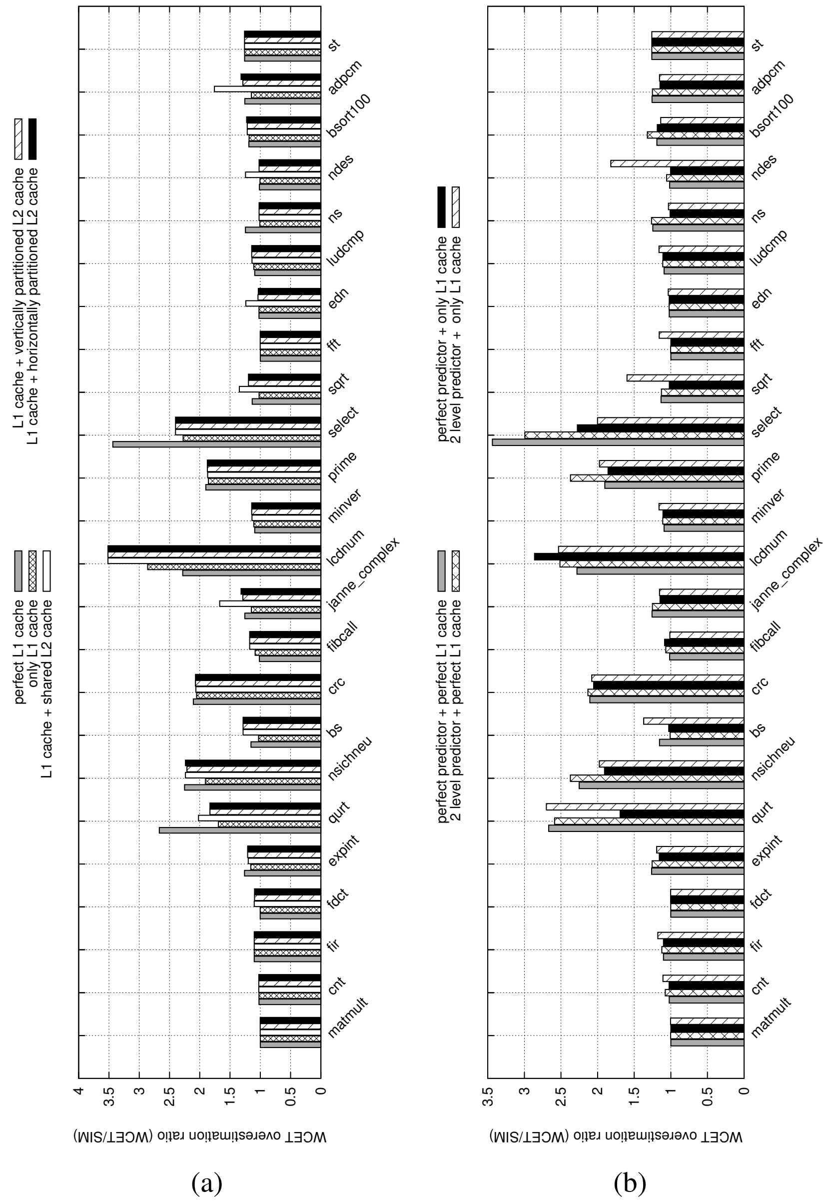


Figure 3.21: (a) Effect of shared and partitioned L2 cache on WCET overestimation, (b) effect of speculation on L1 cache

fect of speculation on caches (refer to Section 3.3.9). Due to the abstract *join* operation to combine the cache states in correct and mispredicted path, some *spurious* cache conflicts might be introduced. Nevertheless, the 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 3.22(b) shows the WCET overestimation in the presence of a shared cache and a shared bus. We observe that the shared bus analysis can reasonably control the overestimation due to the shared bus. Except for a few benchmarks (*e.g.* `edn`, `nsichneu`, `ndes`, `qurt`), the overestimation in the presence of a shared cache and a shared bus is mostly equal to the overestimation when the shared bus analysis is turned off (*i.e.* using 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 include 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 the 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 case 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`, the execution time is dominated by shared bus delay, which is most accurately computed by the shared bus analysis for this benchmark. On the other hand, we observed in Figure 3.21(a) 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 a high WCET overestimation ratio is obtained. Average WCET overestimation in the presence of both a shared cache and a shared bus is around 50%.

3.5.3 WCET analysis results for FIFO replacement policy

Figure 3.23 demonstrates the WCET analysis results with FIFO replacement policy. The experimental setup is exactly the same as mentioned in Section

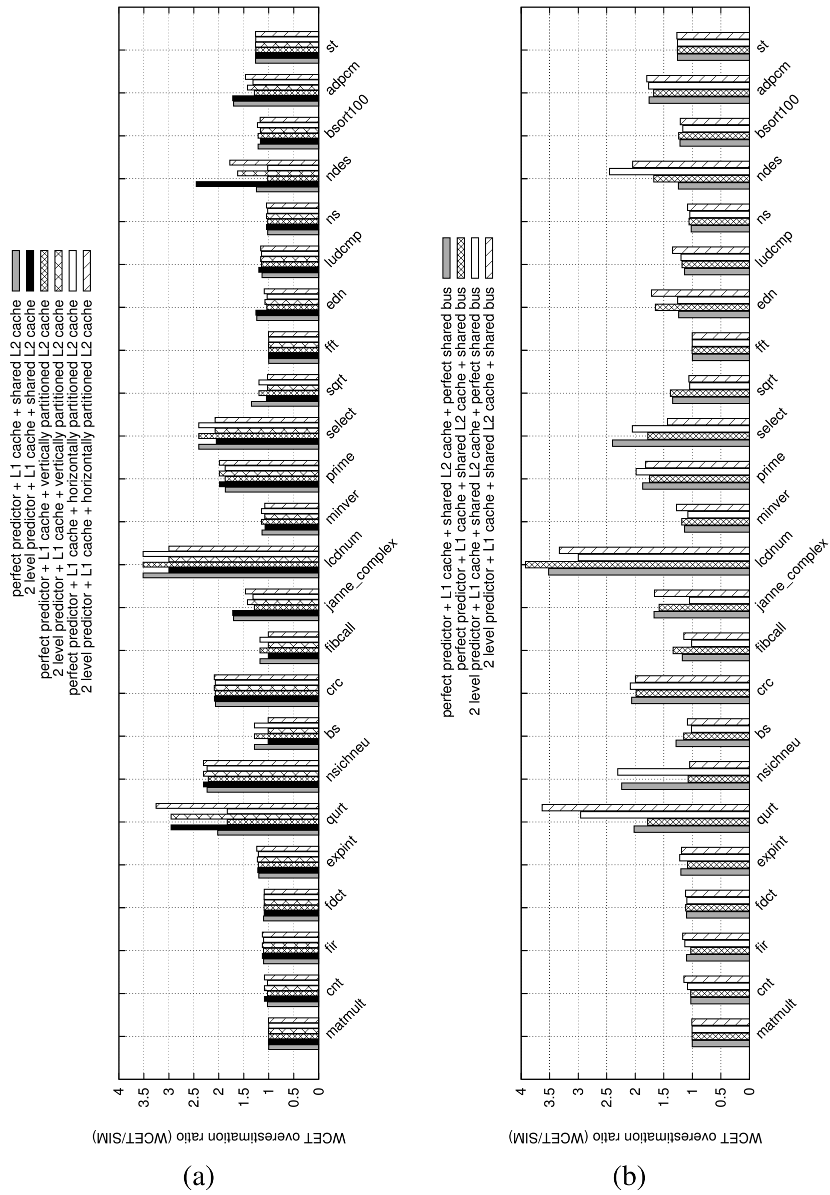


Figure 3.22: (a) effect of speculation on partitioned and shared L2 caches, (b) effect of shared bus on WCET overestimation

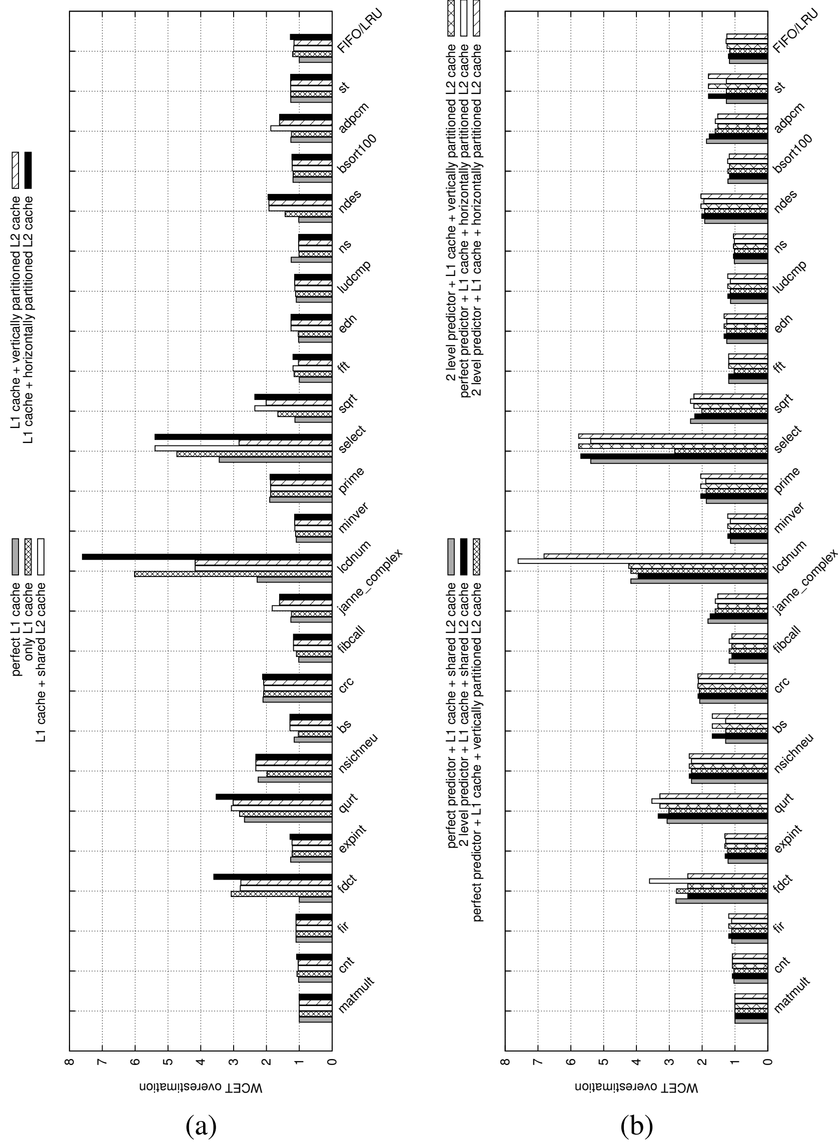


Figure 3.23: Analysis of cache in the presence of FIFO replacement policy (a) WCET overestimation w.r.t. different L2 cache architectures, (b) WCET overestimation in the presence of FIFO cache and speculative execution

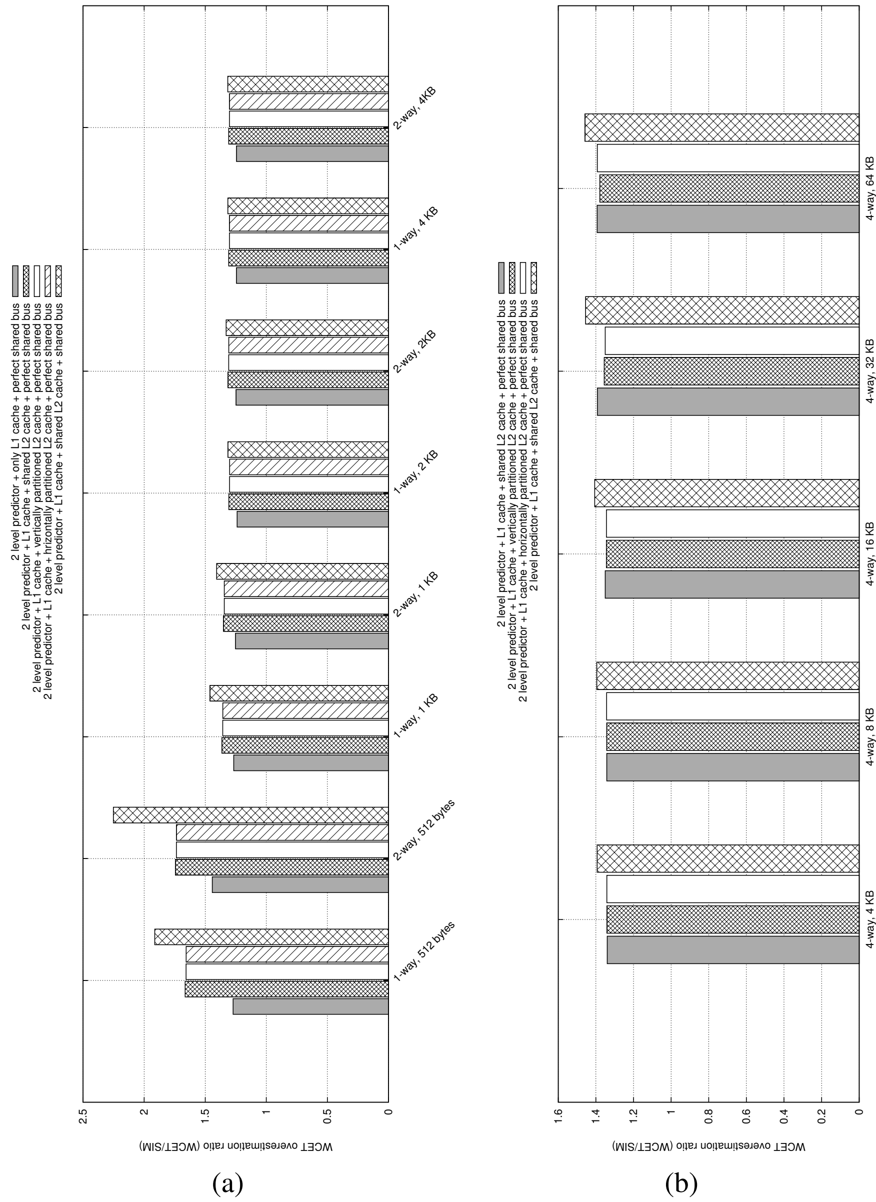


Figure 3.24: WCET overestimation sensitivity w.r.t. (a) L1 cache sizes and configurations; (b) L2 cache sizes and configurations

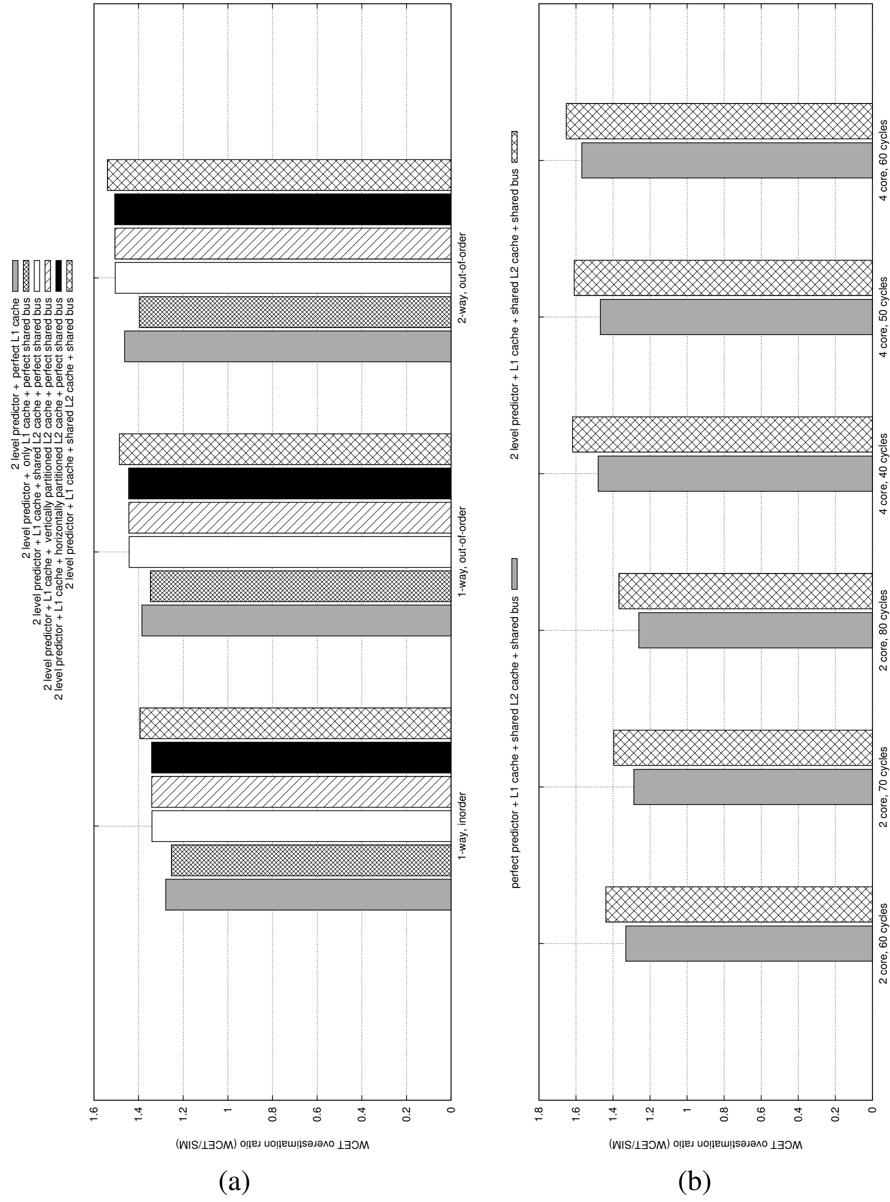


Figure 3.25: WCET overestimation sensitivity w.r.t. (a) pipeline configurations; (b) number of cores and different bus slot lengths

3.5.1. Figure 3.23(a) shows the WCET overestimation ratio in the absence of speculative execution and Figure 3.23(a) shows the same in the presence of branch predictor. In general, the 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 3.23(a). However, as mentioned in [14], the observed worst-case for FIFO replacement may highly under-approximate the *true worst case* due to the *domino effect*. Otherwise, results in Figure 3.23(a) show that FIFO is a reasonably good alternative of LRU replacement even in the context of shared caches.

Figure 3.23(b) shows that the modeling of the interaction between FIFO cache and the branch predictor does not much affect the WCET overestimation. As evidenced by Figure 3.23(b), the increase in the WCET overestimation is *minimal* due to the speculation.

It is worthwhile to mention that the analysis of FIFO caches in a precise manner is more challenging compared to the analysis of LRU caches. In Figures 3.23(a)-(b), the less pessimism is potentially attributed to regular code access patterns and small working sets within loops. For larger working sets and irregular memory access patterns (*e.g.* in `fdct`, `select`), more sophisticated FIFO analyses exist (*e.g.* [33, 34]), which can further improve the analysis precision.

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

In this section, we report the WCET overestimation sensitivity with respect to different micro-architectural parameters. For all the experiments (Figures 3.24-3.25), the reported WCET overestimation denotes the *geometric mean* of the term $\frac{Estimated\ WCET}{Observed\ WCET}$ over all the different benchmarks.

The analysis framework is evaluated for different L1 and L2 cache sizes and configurations (Figure 3.24(a) and Figure 3.24(b), respectively). We observe that the average WCET overestimation is around 40% (50%) with respect to different L1 (L2) cache configurations. Figure 3.25(a) presents the WCET overestimation for different pipeline configurations. Superscalar pipelines increase the instruction level parallelism and therefore, it also in-

creases the performance of the entire program. However, it also becomes difficult to model the inherent instruction level parallelism in the presence of superscalar pipelines. Therefore, Figure 3.25(a) shows an increase in the WCET overestimation with superscalar pipelines. Finally, Figure 3.25(b) shows the WCET overestimation sensitivity with respect to the number of cores and different bus slot lengths. For four core experiments, four adjacent programs are taken (from left to right as shown in Figure 3.21) to run on four different cores. Figure 3.25(b) reports the geometric mean of WCET overestimation over all the benchmarks. With very high length of the TDMA round (*i.e.* number of cores multiplied by TDMA bus slot length), WCET overestimation normally increases (as shown in Figure 3.25(b)). This is due to the fact that with higher TDMA round 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.

3.5.5 Analysis time

All the experiments have been performed on an 8 core, 2.83 GHz Intel Xeon machine having 4 GB of RAM and running Fedora Core 4 operating system. Tables 3.3-3.4 report the *maximum* analysis time when the shared bus analysis is *disabled* and Tables 3.5-3.6 report the *maximum* analysis time when all the analyses are enabled (*i.e.* cache, shared bus and pipeline). Recall from Section 3.3.2 that the 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, the number of mispredicted branches is modeled by integer linear programming [53]. 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 Tables 3.3 and 3.6).

Table 3.3: Analysis time [of `nsichneu`] in *seconds* w.r.t. size of shared L2 cache. The first row represents the analysis time when speculative execution was *disabled*. The second row represents the analysis time when speculation was *enabled*

Shared L2 cache									
4 KB		8 KB		16 KB		32 KB		64 KB	
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
2.6	240	2.9	240	3.5	238	4.6	238	7	239

Table 3.4: Analysis time [of `nsichneu`] in *seconds* w.r.t. pipeline structures. The first row represents the analysis time when speculative execution was *disabled*. The second row represents the analysis time when speculation was *enabled*

Pipeline					
inorder		out-of-order		superscalar	
μ arch	ILP	μ arch	ILP	μ arch	ILP
1.3	1.3	1.2	1.3	1.3	1.4
2.6	238	2.4	239	2.8	254

Shared bus analysis increases the micro-architectural modeling time (as evidenced by Tables 3.5-3.6) 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.

Tables 3.3-3.6 only present the analysis time for the longest running benchmark (`nsichneu`) from the test-suite. For any other program used in the experiment, the entire analysis (micro-architectural modeling and ILP solving time) takes around 20-30 seconds on average to finish.

The results reported in Tables 3.3-3.4 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,

Table 3.5: Analysis time [of `nsichneu`] in *seconds* (two-core systems). The first row shows the analysis time when speculation was *disabled*. The second row shows the analysis time when speculation was *enabled*

Number of cores, TDMA bus slot length					
2 core, 60 cycles		2 core, 70 cycles		2 core, 80 cycles	
μ arch	ILP	μ arch	ILP	μ arch	ILP
128	4	160	4.2	198	5.1
205	158	261	181	363	148

Table 3.6: Analysis time [of `nsichneu`] in *seconds* (four-core systems). The first row shows the analysis time when speculation was *disabled*. The second row shows the analysis time when speculation was *enabled*

Number of cores, TDMA bus slot length					
4 core, 40 cycles		4 core, 50 cycles		4 core, 60 cycles	
μ arch	ILP	μ arch	ILP	μ arch	ILP
199	7.1	228	9.3	257	12.5
373	148	441	165	521	154

such as *interval* instead of an *offset set*. Nevertheless, the appropriate choice of analysis method and abstraction depends on the precision-scalability trade-off required by the user.

3.6 Data caches and branch target buffers

The modeling of data caches is usually more complicated than instruction caches. This is due to the fact that different instances of the same instruction may access different data memory blocks (*e.g.* array accesses inside a loop, pointer aliasing). Therefore, the modeling of data caches usually involves an address analysis phase (*e.g.* similar to the analysis proposed in [12]). The output of address analysis is an over-approximation of the set of addresses accessed by each load/store instruction. Using the results of address analysis, the modeling of data caches has been proposed in [74]. The data cache modeling proposed in [74] is a *must analysis*. Therefore, each load/store instruction is classified as *all-hit* (AH) or *unclassified* (NC). The extension of

the basic data cache modeling for multi-level data caches (as well as for unified caches) has been discussed in [19]. Since the basic technique applied for such data cache modeling is abstract interpretation, the modeling of data caches can easily be integrated into the framework (*e.g.* refer to Equation 3.3 for integration with pipeline and Figure 3.19 for integration with branch prediction). Therefore, the integration of such data cache modeling into the framework does not pose any additional challenge. However, a recent approach ([45]) has shown that a data cache modeling based on address analysis (*e.g.* using [12]) may highly overestimate the WCET. To overcome the imprecision caused due to address analysis, we can compute the set of loop iterations in which a particular data memory block could be accessed [45]. Such a computation strategy is useful for data accesses, as the data memory blocks accessed in disjoint loop iterations can never conflict with each other in the data cache.

Besides accurately estimating the set of data addresses, modeling of data caches might be challenging for specific write policies, such as *write-back*. For *write-through* caches, the write latency involves accessing the main memory. Therefore, apart from address analysis, *write-through* caches do not pose any additional challenge in WCET analysis. However, for *write-back* caches, evicting a memory block might involve variable latencies, depending on whether the specific block has been modified in the cache. Thus, analysis of *write-back* policy needs to precisely estimate the modification history of different memory blocks in the data cache.

Many modern embedded processors also employ a branch target buffer (BTB) to cache the target address of a branch. The BTB analysis proposed in [36] is a combined *must* and *may* analysis. Given any branch instruction address, the analysis proposed in [36] classifies a branch instruction as t (*i.e.* the branch instruction must be in the BTB), f (*i.e.* the branch instruction must not be in the BTB) or $\top$ (*i.e.* static analysis cannot determine the inclusion of the branch instruction in the BTB). Such a classification is analogous to the classification in the instruction cache analysis. Therefore, given an upper bound on BTB miss penalty, such a classification can be integrated into the framework using the technique similar to Equation 3.3. Moreover, the static analysis of BTB content (as proposed in [36]) can be used in the framework

to determine the speculative instructions and their effects on caches (exactly in the same fashion as shown in Figure 3.19).

3.7 A survey of related techniques

Performance analysis and predictability of embedded software has been an active topic of research for several decades. A recent survey article [11] has discussed several research efforts in the past to build time-predictable embedded systems. Since the inception of multi-core architectures, the research on performance analysis is gradually focusing its attention towards multi-core platforms. Analysis of shared caches has been proposed in [85, 39]. These works on shared caches use some variant of abstract interpretation (AI) based analysis [80] and extend the AI-based analysis for private caches [80] to shared caches. The work proposed in [85] has two limitations. First, this work does not exploit task dependencies. Such dependencies may eliminate some spurious cache conflicts, which might be created between tasks that can never coexist. This limitation can be solved by techniques discussed in Section 3.1.2 and Section 3.2.4. Secondly, it does not include any additional optimization for set-associative caches. This limitation can also be handled by the methodologies described earlier in this monograph (Section 3.1.2 and Section 3.2.4).

The impact of shared buses on WCET analysis has also been investigated by several research communities. The work in [60] proposes a framework based on abstract interpretation and model checking to analyze embedded software on multi-core platforms. Timed-automata model checking has been used to analyze TDMA and *first-in-first-out* (FIFO) buses. A subsequent work [30] proposes to use model checking with abstractions. Due to the heavy complexity of model checking, techniques such as [60] cannot scale beyond two cores [30]. Therefore, several abstractions have been proposed. These abstractions aim to reduce the complexity of model checking for bus analysis on many-core systems. The work in [73] proposes system-level analysis of embedded software in the presence of TDMA arbitration policy. A different work [15] analyzes the timing effects in a ring bus architecture, where multiple in-flight bus transactions may coexist. However, it is worthwhile to note that the works proposed in [73, 15] are targeted towards schedulability anal-

ysis of embedded software, which is performed, in general, after low-level WCET analysis. WCET analysis in the presence of complex bus topologies (*e.g.* mesh, ring and torus) still poses significant research challenges. Finally, the work in [66] discusses the impact of resource interferences on WCET for embedded systems using *commercial off-the-shelf* (COTS) components. Specifically, the impact of peripheral activities have been investigated. Such peripheral activities may substantially delay the execution time of a program by occupying the shared bus.

In summary, WCET analysis in the presence of multi-core platforms (and hence in the presence of resource sharing) is currently an active area of research. Deriving a safe as well as a precise upper bound on the WCET for multi-core platforms is challenging. We believe that works presented in this section will give valuable insights and it will lead to substantial research activities in future.

4

WCET optimization for multi-core platforms

In the preceding section, we have described several analysis methodologies to predict the execution time of embedded software on multi-core platforms. In this section, we shall explore an orthogonal direction to improve the time-predictability on multi-core platforms. In the following, we shall primarily describe a compiler-directed optimization to improve the WCET on multi-core platforms.

Given any TDMA bus schedule, we have seen the computation of a safe WCET estimate in the preceding section. This means that the WCET of a task is directly dependent on the bus schedule. In the subsequent sections, we shall describe how to generate a bus schedule, while satisfying various efficiency requirements (previously proposed in [69] and the part of the content in this section has previously been published in [69, 70]).

4.1 Optimization of worst-case response time

Since the bus schedule is directly affecting the worst-case execution time of the tasks, and consequently also the worst-case response time (WCRT) of the application, it is important that it is chosen carefully. Ideally, when constructing the bus schedule, we would like to allocate a time slot for each

individual cache miss on the worst-case control flow path, granting access to the bus immediately when it is requested. There are, however, two significant problems preventing us from doing this. The first one is that several cores can issue a cache miss at the same time instant, creating conflicts on the bus. The second problem is that allocating bus slots for each individual memory transfer would create a very irregular bus schedule, requiring an unfeasible amount of memory space on the bus controller.

In order to solve the problem of irregular and memory-consuming bus schedules, some restrictions on the TDMA round complexity need to be imposed. For instance, an efficient strategy is to allow each core to own the maximum number of slots per round. Other limitations can be to let each round have the same slot order, or to force the slots in a specific round to have the same size. In this section, we assume that every core can own at most one bus slot per round. The slots in a round can have different sizes, and the order can be set without restrictions. However, it is straight-forward to adapt this algorithm to more (or less) flexible bus schedule design rules. In addition to the main algorithm, we discuss a simplified algorithm for the special case where all slots in a round must be of the same size.

The problem of handling cache miss conflicts is handled using the technique described in Section 3.2.2. This is done in the inner loop of the overall approach outlined in Algorithm 3.1. For the optimization process, we shall first give an outline of the overall approach. A detailed description will follow in subsequent sections.

4.2 WCRT optimization approach

Algorithm 4.1 outlines the overall approach for bus schedule optimization. In general, the approach revolves around minimizing a *cost function* (see Section 4.3 for details). This cost function, in turn, captures the *worst-case response time* (WCRT) of the application as a function of the bus bandwidth distribution. As far as the bus schedule is concerned, there are primarily two factors that influence the WCRT of an application:

- The ordering of TDMA slots assigned to each task
- The size of each TDMA slot

Algorithm 4.1 The optimization approach

1. Define cost function (*cf.* Section 4.3)
2. Calculate initial slot sizes (*cf.* Section 4.4.2)
3. Calculate an initial slot order
4. Analyze the WCET of each task $\tau \in \Psi$ and evaluate the result according to the cost function
5. Generate a new slot order candidate and repeat from 4 until all candidates are evaluated (*cf.* Section 4.4.1)
6. Select the best slot order candidate according to the cost function
7. Generate a new slot size candidate and repeat from 3 until the exit condition is met (*cf.* Section 4.4.3)
8. The best configuration according to the cost function is then used

Therefore, in a broader perspective, the optimization approach can be viewed as searching through the space of different possible slot orderings and slot sizes. At first, an initial slot size is assigned to each core. This initial selection is based on an estimation on how the slot size assigned to each core affects the overall response time (*cf.* Section 4.4.2). Once an initial selection of slot sizes is made, the optimization procedure aims to find the best slot ordering for the given slot size selection. To accomplish this, the process first selects a default slot ordering and estimates the task τ_i which maximizes the cost function. Subsequently, the optimization procedure attempts to find a different slot ordering based on the following intuition: since τ_i is responsible for maximizing the cost function, it is likely that the cost will reduce if we change the relative position of the slot assigned to τ_i in the TDMA round. Based on this intuition, different slot orderings are generated (*cf.* Section 4.4.1) by swapping the relative position of the slot assigned to τ_i with a different position in the TDMA round. Finally, the best slot ordering is chosen according to the cost function.

The preceding paragraph outlines the inner optimization loop (lines 4-5 of Algorithm 4.1). The outer optimization loop (lines 3-7 of Algorithm 4.1) attempts to find the best slot size assigned to each core (*cf.* Section 4.4.3). In particular, very large slots assigned to a task may substantially delay other tasks to access the shared bus. Therefore, it is important to appropriately adjust the slot sizes. The slot size selection revolves around the estimation

of bus bandwidth distribution. Specifically, two bandwidth distributions are computed:

- From the worst-case execution time (WCET) analysis. This captures the bus bandwidth that belongs to each core along the worst-case path of the task running on it. Let us call this bus bandwidth p_i for a specific task τ_i. Intuitively, p_i can be viewed as the *demand* of bus bandwidth for τ_i.

- From the slot size distribution. Let us call this bus bandwidth p'_i for a specific task τ_i. Intuitively, p'_i can be viewed as the *provision* of bus bandwidth to τ_i.

For instance, if the slot sizes are distributed as $\langle k, 2k, 3k \rangle$ (k being the minimum slot size) to three different cores, the bus bandwidth distribution from the slot size is $\langle 0.50, 0.33, 0.17 \rangle$. The purpose of slot-size selection is to primarily reduce the deviation of bus bandwidth, quantified by $p'_i - p_i$. In other words, the primary goal of slot-size selection is to *balance* the *provision* and the *demand* of bus bandwidth. The optimization loop first tries to find the task having the maximum deviation in bus bandwidth and reduces the slot-size assigned to the task, one minimum slot-size at a time. For these new slot sizes, the inner optimization loop attempts to find the best slot ordering, as described in the preceding paragraph. Finally, this selection of slot ordering and slot-size continues as long as the time budget for optimization permits or the resulting solution no longer leads to any improvement in the overall WCRT.

4.3 Cost function

Recall that in Sections 3.2-3.3, we had discussed analyses of TDMA-based bus arbitration. For the sake of simplicity, we had restricted our discussion for a simple TDMA schedule, where each core is assigned a bus slot of the same length. However the analysis methodologies, as described in the previous section, also hold for more complex TDMA schedules (*e.g.* TDMA schedules described in the following discussion). In the following, we shall discuss bus schedule optimization for a more general TDMA-based arbitra-

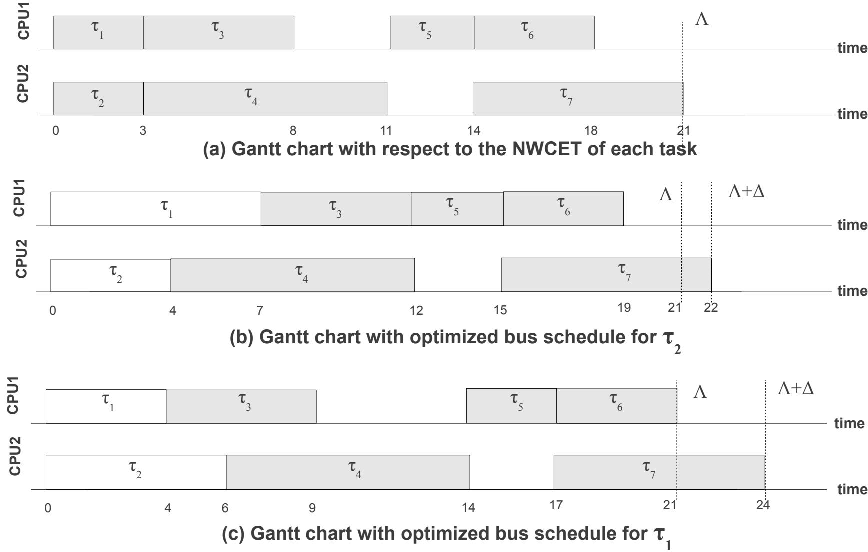


Figure 4.1: Estimating the response time

tion method. Nevertheless, the WCET analysis can be carried out analogous to the approach discussed in the previous section.

Given a set of active tasks $\tau_i \in \Psi$, the goal is now to generate a close to optimal bus segment schedule with respect to Ψ. The optimal bus schedule is a bus schedule taking into account the global context, minimizing the response time of the application. This response time includes tasks not yet considered and for which no bus schedule has been defined. This requires knowledge about future tasks, not yet analyzed, and, therefore, we must find ways to approximate their influence on the response time.

In order to estimate the response time, we need to build a schedule S^λ of the tasks not yet analyzed, using a list scheduling technique. When building S^λ the WCET of each task is approximated by its respective worst-case execution time in the naive case, where no conflicts occur on the bus and any task can access the bus at any time. From now on we refer to this conflict-free WCET as NWCET (Naive Worst-Case Execution Time).

When optimizing the bus schedule for the tasks $\tau \in \Psi$, we need an approximation of how the WCET of one task $\tau_i \in \Psi$ affects the response time.

Let D_i be the union of the set of all tasks depending directly on τ_i in the process graph, and the singleton set containing the first task in S^λ that is scheduled on the same core as τ_i. We now define the tail λ_i of a task τ_i recursively as:

- $\lambda_i = 0$, if $D_i = \emptyset$
- $\lambda_i = \max_{\tau_j \in D_i} (x_j + \lambda_j)$, otherwise.

where $x_j = \text{NWCET}_j$ if τ_j is a computation task. For communication tasks (*i.e.* message passing between two computation tasks), x_j is an estimation of the communication time, depending on the length of the message. Intuitively, λ_i can be seen as the length of the longest (with respect to the NWCET) chain of tasks that are affected by the execution time of τ_i. Without any loss of generality, in order to simplify the presentation, only computation tasks are considered in the examples of this section. Consider Figure 4.1a, illustrating a Gantt chart of tasks scheduled according to their NWCETs. Direct data dependencies exist between tasks τ_4 & τ_5, τ_5 & τ_6, and τ_5 & τ_7; hence, for instance, $D_3 = \{\tau_5\}$ and $D_4 = \{\tau_5, \tau_7\}$. The tails of the tasks are: $\lambda_7 = \lambda_6 = 0$ (since $D_7 = D_6 = \emptyset$), $\lambda_5 = 7$, $\lambda_4 = \lambda_3 = 10$, $\lambda_2 = 18$ and $\lambda_1 = 15$.

Since the main concern when optimizing the bus schedule for the tasks in Ψ is to minimize the response time, a cost function taking λ_i into account can be formulated as follows:

$$C_{\Psi,\theta} = \max_{\tau_i \in \Psi} (\theta + \text{WCET}_i^\theta + \lambda_i) \tag{4.1}$$

where WCET_i^θ is defined as the length of that portion of the worst case execution path of task τ_i which is executed after time θ.

4.4 Optimization algorithm

In this Section, we shall describe the different components of Algorithm 4.1 in detail.

4.4.1 Slot order selection

At step 4 of Algorithm 4.1, a default initial order is set. When step 5 is reached for the first time, after calculating a cost for the current slot configuration, the

task $\tau_i \in \Psi$ that is maximizing the cost function in Equation 4.1 is identified. Subsequently, $n-1$ new bus schedule candidates are constructed, n being the number of tasks in the set Ψ, by moving the slot corresponding to this task τ_i, one position at a time, within the TDMA round. It is important to note that only the relative position of the bus slot corresponding to τ_i is *swapped* with a different position. Therefore, if τ_i is assigned to core c_i and the initial slot ordering belongs to three cores in the order $c_{i-1} \rightarrow c_i \rightarrow c_{i+1}$, two different slot orderings are generated, namely $c_{i-1} \rightarrow c_{i+1} \rightarrow c_i$ and $c_i \rightarrow c_{i-1} \rightarrow c_{i+1}$. Since τ_i is maximizing the cost function defined in Equation 4.1, the intuition is that a different relative position of the slot corresponding to τ_i will most likely reduce the cost of τ_i and hence will reduce the overall cost. Once all $n-1$ slot orderings are generated, the best slot ordering with respect to the cost function is selected. Next, we need to check if any new task τ_j, different from τ_i, now has taken over the role of maximizing the cost function. If so, the procedure of slot ordering selection is repeated, otherwise it is terminated.

4.4.2 Determination of initial slot sizes

At step 2 of Algorithm 4.1, the initial slot sizes are dimensioned based on an estimation of how the slot size of an individual task $\tau_i \in \Psi$ affects the response time.

Consider λ_i, as defined in Section 4.3. Since it is a sum of the NWCETs of the tasks forming the tail of τ_i, it will never exceed the sum of WCETs of the same sequence of tasks. Consequently, for all $\tau_i \in \Psi$ define

$$\Lambda = \max_{\tau_i \in \Psi}(\text{NWCET}_i^{\theta} + \lambda_i) \tag{4.2}$$

where NWCET_i^{θ} is the NWCET of task $\tau_i \in \Psi$ counting from time θ, a lower limit of the response time can be calculated by $\theta + \Lambda$. This is illustrated in Figure 4.1a, for $\theta = 0$. Furthermore, let us define Δ as the amount by which the estimated response time increases due to the time each task $\tau_i \in \Psi$ has to wait for the bus.

See Figure 4.1c for an example. Contrary to Figure 4.1a, τ_1 and τ_2 are now considered using their real WCETs, calculated according to a particular bus schedule ($\Psi = \{\tau_1, \tau_2\}$). The corresponding expansion Δ is 3 time units. Now, in order to minimize Δ, we want to express a relation between the

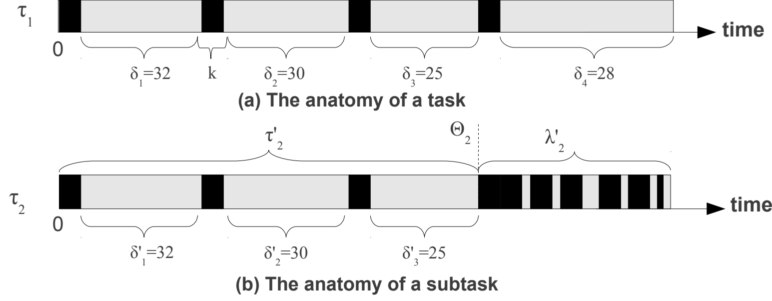


Figure 4.2: Close-up of two tasks

response time and the bus schedule. For task $\tau_i \in \Psi$, we define m_i as the number of remaining cache misses on the worst case path, counting from time θ. Similarly, also counting from θ, l_i is defined as the execution time on processor and can thus be seen as the length (in terms of execution time) of the task minus the time it spends using the bus or waiting for it (both m_i and l_i are determined by the WCET analysis). Hence, if we define the constant k as the time it takes to process a cache miss when ignoring bus conflicts, we get

$$\text{NWCET}_i^{\theta} = l_i + m_i k \tag{4.3}$$

As an example, consider Figure 4.2a showing a task execution trace, in the case where no other tasks are competing for the bus. A black box represents the idle time, waiting for the transfer, due to a cache miss, to complete. In this example $m_1 = 4$ and $l_1 = \delta_1 + \delta_2 + \delta_3 + \delta_4 = 115$.

Let us now, with respect to the particular bus schedule, denote the average waiting time of task τ_i by d_i. That is, d_i is the average time task τ_i spends waiting, due to other cores owning the bus and the actual time of the transfer itself, every time a cache miss has to be transferred on the bus. Then, analogously to Equation 4.3, the WCET of task τ_i, counting from time θ, can be calculated as

$$\text{WCET}_i^{\theta} = l_i + m_i d_i \tag{4.4}$$

The dependency between a set of average waiting times d_i and a bus schedule can be modeled as follows. Consider the distribution P, defined as the set $p_1, \ldots, p_n$, where $\sum p_i = 1$. The value of p_i represents the fraction of bus bandwidth that, according to a particular bus schedule, belongs to the core running task $\tau_i \in \Psi$. Given this model, the average waiting times can be rewritten as

$$d_i = \frac{1}{p_i} k \tag{4.5}$$

If $p_i = 1$, the core running task $\tau_i \in \Psi$ has the full bus bandwidth and therefore, it will not suffer any additional bus delay. As observed from Equation 4.5, every cache miss will suffer the constant cache miss delay k, if full bus bandwidth is allocated to the respective core. If $p_i < 1$, we get $d_i > k$, capturing the additional waiting time to access the shared bus.

Putting Equations 4.2, 4.4, and 4.5 together and noting that Λ has been calculated as the maximum over all $\tau_i \in \Psi$, we can formulate the following system of inequalities:

$$\begin{gathered}
\theta + l_1 + m_1 \frac{1}{p_1} k + \lambda_1 \leq \theta + \Lambda + \Delta \\
\vdots \\
\theta + l_n + m_n \frac{1}{p_n} k + \lambda_n \leq \theta + \Lambda + \Delta \\
p_1 + \cdots + p_n = 1
\end{gathered}$$

What we want is to find the bus bandwidth distribution P that results in the minimum Δ satisfying the above system. Unfortunately, solving this system is difficult due to its enormous solution space. However, an important observation that simplifies the process can be made, based on the fact that the slot distribution is represented by continuous variables p. Consider a configuration of $p_1, \ldots, p_n, \Delta$ satisfying the above system, and where at least one of the inequalities is not satisfied by equality. We say that the corresponding task τ_i is not on the critical path with respect to the schedule, meaning that its corresponding p_i can be decreased, causing τ_i to expand over time without affecting the response time. Since the values of p must sum to 1, decreasing p_i, allows for increasing the percentage of the bus given to the tasks τ that are on the critical path. Even though the decrease might be infinitesimal, this makes

the critical path shorter, and thus Δ is reduced. Consequently the smallest Δ that satisfies the system of inequalities is achieved when every inequality is satisfied by equality. As an example, consider Figure 4.1c and note that τ_5 is an element in both sets D_3 and D_4 according to the definition in Section 4.3. This means that τ_5 is allowed to start first when both τ_3 and τ_4 have finished executing. Secondly, observe that τ_5 is on the critical path, thus being a direct contributor to the response time. Therefore, to minimize the response time, we must make τ_5 start as early as possible. In Figure 4.1c, the start time of τ_5 is defined by the finishing time of τ_4, which also is on the critical path. However, since there is a block of slack space between τ_3 and τ_5, we can reduce the execution time of τ_2 and thus make τ_4 finish earlier, by distributing more bus bandwidth to the corresponding core. This will make the execution time of τ_1 longer (since it receives less bus bandwidth), but as long as τ_3 ends before τ_4, the response time will decrease. However, if τ_3 expands beyond the finishing point of τ_4, the former will now be on the critical path instead. Consequently, making task τ_3 and τ_4 end at the same time, by distributing the bus bandwidth such that the sizes of τ_1 and τ_2 are adjusted properly, will result in the earliest possible start time of τ_5, minimizing Δ. In this case the inequalities corresponding to both τ_1 and τ_2 are satisfied by equality. Such a distribution is illustrated in Figure 4.1b.

The resulting system consists of $n + 1$ equations and $n + 1$ variables (p_1, $\ldots$, p_n and Δ), meaning that it has exactly one solution, and even though it is nonlinear, it is simple to solve. Using the resulting distribution, a corresponding initial TDMA bus schedule is calculated by setting the slot sizes to values proportional to P.

4.4.3 Generation of new slot-size candidates

One of the possible problems with the slot sizes defined as in Section 4.4.2 is the following: if one core gets a very small share of the bus bandwidth, the slot sizes assigned to the other cores can become very large, possibly resulting in long wait times. By reducing the sizes of the larger slots while trying to keep their mutual proportions, this problem can be avoided.

Let us first consider an example where a round, consisting of three slots, are ordered as in Figure 4.3a. The slot sizes have been dimensioned according to a bus distribution $P = \{0.49, 0.33, 0.18\}$, calculated using the method

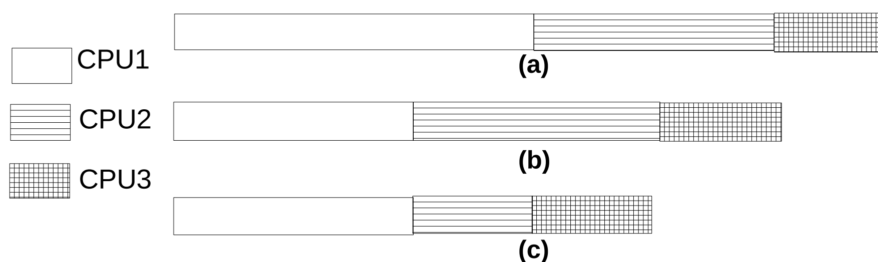


Figure 4.3: Calculation of new slot sizes

in Section 4.4.2. The smallest slot, belonging to CPU 3, has been set to the minimum slot size k, and the remaining slot sizes are dimensioned proportionally [1] as multiples of k. Consequently, the initial slot sizes become $3k$, $2k$ and k. In order to generate the next set of candidate slot sizes, we define P' as the actual bus distribution of the generated round. Considering the actual slot sizes, the bus distribution becomes $P' = \{0.50, 0.33, 0.17\}$. Since very large slots assigned to a certain core can introduce long wait times for tasks running on other cores, we want to decrease the size of slots, but still keep close to the proportions defined by the bus distribution P. Consider once again Figure 4.3a. Since, $p'_1 - p_1 > p'_2 - p_2 > p'_3 - p_3$, we conclude that slot 1 has the maximum deviation from its supposed value. Hence, as illustrated in Figure 4.3b, the size of slot 1 is decreased one unit. This slot size configuration corresponds to a new actual distribution $P' = \{0.40, 0.40, 0.20\}$. Now $p'_2 - p_2 > p'_3 - p_3 > p'_1 - p_1$, hence the size of slot 2 is decreased one unit and the result is shown in Figure 4.3c. Note that in the next iteration, $p'_3 - p_3 > p'_1 - p_1 > p'_2 - p_2$, but since slot 3 cannot be further decreased, we recalculate both P and P', now excluding this slot. The resulting sets are $P = \{0.60, 0.40\}$ and $P' = \{0.67, 0.33\}$, and hence slot 1 is decreased one unit. From now on, only slots 1 and 2 are considered, and the remaining procedure is carried out in exactly the same way as before. When this procedure is continued as above, all slot sizes will converge towards k which, of course, is not the desired result. Hence, after each iteration, the cost function (Equa-

[1] In practice, slot sizes are usually multiples of the minimum slot size k to avoid unnecessary slack on the bus.

tion 4.1) is evaluated and the process is continued only until no improvement is registered for a specified number π of iterations. The best slot sizes (with respect to the cost function) are, finally, selected. Accepting a number of steps without improvement makes it possible to escape certain local minima[2].

4.4.4 Density regions

A problem with the technique presented above is that it assumes that the cache misses are evenly distributed throughout the task. We call this distribution of cache misses along the task execution as the *cache miss structure*. For most tasks, the cache misses are not evenly distributed throughout the task execution. A solution to this problem is to analyze the internal cache miss structure of the actual task and, accordingly, divide the worst-case path into disjoint intervals, so called *densityregions*. A *density region* is defined as an interval of the path where the distance between consecutive cache misses (δ in Figure 4.2) does not differ more than a specified number. In this context, if we denote by α the average time between two consecutive cache misses (inside a region), the density of a region is defined as $\frac{1}{\alpha+1}$. A region with high density, close to 1, has very frequent cache misses, while the opposite holds for a low-density region.

Consequently, in the beginning of the optimization loop, we identify the next density region for each task $\tau_i \in \Psi$. Now, instead of constructing a bus schedule with respect to each entire task $\tau_i \in \Psi$, only the interval $[\theta..\Theta_i)$ is considered, with Θ_i representing the end of the density region. We call this interval of the task a subtask since it will be treated as a task of its own. Figure 4.2b shows a task τ_2 with two density regions, the first one corresponding to the subtask τ_2'. The tail of τ_2' is calculated as $\lambda_2' = \lambda_2'' + \lambda_2$, with λ_2'' being defined as the NWCET of τ_2 counting from Θ_2. Furthermore, in this particular example $m_2' = 3$ and $l_2' = \delta_1' + \delta_2' + \delta_3' = 87$.

Consider Algorithm 3.1 illustrating the overall approach. Analogous to the case where the entire tasks are analyzed, when a bus schedule for the current bus segment has been decided, θ' will be set to the finish time of the first subtask. Just as before, the entire procedure is then repeated for $\theta = \theta'$.

However, modifying the bus schedule can cause the worst-case control flow path to change. Therefore, the entire cache miss structure can also

[2]In experiments the range $8 < \pi < 40$ is used, depending on the number of cores

change during the optimization procedure (lines 4 and 5 in Algorithm 4.1), resulting in possible changes with respect to both subtask density and size. This problem is solved by using an iterative approach, adapting the bus schedule to possible changes of the subtask structure while making sure that the total cost is decreasing. This procedure will be described in the following paragraphs.

Subtask evaluation

First, let us in this context define two different cost functions, both based on Equation 4.1. Let ${\tau_i'}^{\text{end}}$ be the end time of subtask τ_i', and define ${\tau'}^{\text{end}}$ as:

$${\tau'}^{\text{end}} = \min_{\tau_i \in \Psi}({\tau_i'}^{\text{end}}) \tag{4.6}$$

Furthermore, let $\text{NWCET}_i^{{\tau'}^{\text{end}}}$ be the NWCET of the task τ_i, counting from ${\tau'}^{\text{end}}$ to the end of the task. The *subtask cost* $C'_{\Psi,\theta}$ can now be defined as:

$$C'_{\Psi,\theta} = \max_{\tau_i \in \Psi}({\tau'}^{\text{end}} + \text{NWCET}_i^{{\tau'}^{\text{end}}} + \lambda_i) \tag{4.7}$$

Hence, the subtask cost is a straight-forward adaption of the cost function in Equation 4.1 to the concept of subtasks. Instead of using the worst-case execution time of the entire task, only the part corresponding to the first density region after time θ is considered. The rest of the task, from the end of the first density region to the end of the entire task, is accounted for in the tail, with respect to its corresponding NWCET.

In order to more accurately approximate how the subtask affects the worst-case response time, its *complementary task cost* $C''_{\Psi,\theta}$ is introduced in addition to the subtask cost. Let $\text{WCET}_i^{{\tau'}^{\text{end}}}$ be the worst-case execution time of task τ_i starting from time ${\tau'}^{\text{end}}$. We here assume that $\text{WCET}_i^{{\tau'}^{\text{end}}}$ has been calculated with respect to a tailored bus segment, starting after ${\tau'}^{\text{end}}$. The bus schedule representing this bus segment is calculated considering the cache miss structure of the corresponding part of the task, for instance by using the algorithm described in Section 4.4.2 for calculating initial slot sizes. This way we can approximate the transfer delays of the cache misses between ${\tau'}^{\text{end}}$ and the end of the task, instead of using the corresponding NWCET (as

is done when calculating the subtask cost). The complementary task cost can be defined as:

$$C''_{\Psi,\theta} = \max_{\tau_i \in \Psi} (\tau'^{\text{end}} + \text{WCET}_i^{\tau'^{\text{end}}} + \lambda_i) \tag{4.8}$$

Note that the only difference between this cost function and the previous one in Equation 4.7 is that we now use a calculated WCET for the remaining part of the task, instead of the NWCET. Consequently, the complementary task cost is always greater than or equal to the subtask cost. The problem with using the NWCET, as done when calculating the subtask cost, is that small subtasks tend to be favored. The complementary cost is more precise, but also more time-consuming to calculate. Therefore the idea is to use it only when necessary.

With the two cost functions defined, we can now formulate an algorithm for subtask evaluation, as presented in Algorithm 4.2. In step 2, the tasks $\tau_i \in \Psi$ are analyzed, in their entirety, in order to achieve an initial cache miss structure. This structure is then used to identify the first subtask τ'_i of each task (step 3), and to calculate an initial bus schedule (step 4). In order to evaluate the bus schedule, the complementary cost is evaluated in step 5. In step 8, the bus schedule is modified with respect to the subtasks τ_i. The algorithms to change the slot sizes and order of the current TDMA round, used for these modifications, can be found in Section 4.4.3 and Section 4.4.1. In step 9, the first corresponding subtask τ'_i of each task $\tau_i \in \Psi$ is reidentified with respect to the new cache miss structure, and an updated cost is calculated (by using the less expensive subtask cost function). If this cost is an improvement of the previous cost[3], we can also evaluate the complementary cost $C''_{\Psi,\theta}$. If the new complementary cost is lower than the best cost $C^{\text{best}}_{\text{inner}}$ found so far in the inner loop, we can update $C^{\text{best}}_{\text{inner}}$ to this new lowest cost.

We then try to modify the bus schedule further until no more improvements are found (steps 8-12). Consequently, reaching step 13 means two things. Either we have found the best bus schedule, or the worst-case control flow path has changed during the iterations, resulting in a different cache miss structure, not suitable for the generated bus schedule (again, note that

[3]In the opposite case, for which no improvement of the cost was made, there is no need to calculate $C''_{\Psi,\theta}$ since $C'_{\Psi,\theta} < C''_{\Psi,\theta}$.

Algorithm 4.2 Subtask evaluation algorithm

1. Set $C_{\text{outer}}^{\text{best}} = \infty$
2. Calculate initial slot sizes with respect to all tasks $\tau_i \in \Psi$
3. For each task $\tau_i \in \Psi$, calculate the WCET and identify the corresponding first subtask τ_i'
4. Calculate the initial slot sizes with respect to the subtasks τ_i'
5. Calculate the complementary task cost $C_{\Psi,\theta}^{''}$
6. If $C_{\Psi,\theta}^{''} < C_{\text{outer}}^{\text{best}}$, set $C_{\text{outer}}^{\text{best}} = C_{\Psi,\theta}^{''}$
7. Set $C_{\text{inner}}^{\text{best}} = C_{\text{outer}}^{\text{best}}$
8. Modify the bus schedule with respect to the cache miss structure of τ_i'
9. Once again, for each task $\tau_i \in \Psi$, calculate the WCET and identify the first corresponding subtask τ_i'
10. Calculate the subtask cost $C_{\Psi,\theta}^{'}$
11. If $C_{\Psi,\theta}^{'} < C_{\text{inner}}^{\text{best}}$, calculate the complementary task cost $C_{\Psi,\theta}^{''}$ and, if $C_{\Psi,\theta}^{''} < C_{\text{inner}}^{\text{best}}$, set $C_{\text{inner}}^{\text{best}} = C_{\Psi,\theta}^{''}$
12. Repeat from 9 until no improvements have been made for N iterations
13. If $C_{\text{inner}}^{\text{best}} < C_{\text{outer}}^{\text{best}}$ then set $C_{\text{outer}}^{\text{best}} = C_{\text{inner}}^{\text{best}}$ and goto 4
14. Use the bus schedule corresponding to $C_{\text{outer}}^{\text{best}}$ for the interval between θ and the end time of the subtask that finished first, and update θ to this end time

the steps 8-12 try to improve the initial sizes calculated, with respect to a specific density, in step 4). If $C_{\text{inner}}^{\text{best}} = C_{\text{outer}}^{\text{best}}$, we did not manage to improve the existing best cost from the last time the inner loop was visited, and the algorithm is halted. If $C_{\text{inner}}^{\text{best}} < C_{\text{outer}}^{\text{best}}$, on the other hand, we identify new subtasks with respect to the improved bus schedule (step 3), and repeat the procedure. It is worthwhile to note that Algorithm 4.2 will always converge towards a solution. This is because the algorithm never accepts solutions that lead to higher costs.

4.5 Simplified algorithm

For the case where all slots of a round have to be of the same, round-specific size, calculating the distribution P makes little sense. Therefore, we discuss a simpler, but quality-wise equally efficient algorithm, tailor-made for this class of more limited bus schedules. The slot ordering mechanisms are still the same as for the main algorithm, but the procedures for calculating the slot sizes are now vastly simplified. Algorithm 4.3 shows the overall process.

Algorithm 4.3 The simplified optimization approach

1. Initialize the slot sizes to the minimum size k
2. Calculate an initial slot order
3. Analyze the WCET of each task $\tau \in \Psi$ and evaluate the result according to the cost function
4. Generate a new slot order candidate and repeat from 3 until all candidates are evaluated
5. Increase the slot sizes one step
6. If no improvements were achieved during a specified number of iterations then exit. Otherwise repeat from 2
7. The best configuration according to the cost function is then used

In step 1 of this algorithm, we can start by using the smallest possible slot size, since this will minimize the maximum transfer delay. Next, an initial slot order, chosen arbitrarily, is specified in step 2. The slot order candidates are then generated just as in the general algorithm, by changing the position of the slot belonging to the core on the critical path. After finding the best order for a particular slot size, the latter is modified by, for instance, increasing it k steps. After an appropriate slot size is found, it can also be "fine tuned" by increasing or decreasing the size by a very small amount, less than k. Since all cores get the same amount of bus bandwidth, the concept of density regions is not useful in this simplified approach.

4.6 Memory consumption

A TDMA bus schedule is usually composed of segments. Therefore, the amount of memory space needed to store the bus schedule is defined by the number of segments and the complexity restrictions imposed, by the system designer, on the underlying TDMA rounds. In order to calculate an upper bound on the number of segments needed, we make the observation that a new segment is created at every time t when at least one task starts or finishes. For the case when density regions are not used, these are also the only times when a new segment will be created. Hence, an upper bound on the number of segments is $2|\Pi|$, where Π is the set of all tasks.

When using density regions, the start and finish of every region can result in a new segment each. Therefore, tasks divided into very many small density

regions will result in bus schedules consuming a lot of memory. A straightforward solution is to limit, according to the available controller memory, the minimum size of a density region. For instance, if the minimum density region size for a task τ_i is $x\%$ of the task length l_i as defined above, the number of generated segments becomes at most $2|\Pi|\frac{100}{x}$.

4.7 Experimental results

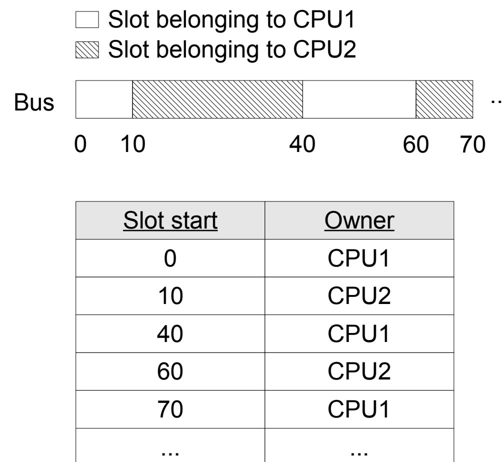
The complete flow illustrated in Figure 3.1 has been implemented and used as a platform for the experiments presented in this section. The bus schedule synthesis was carried out on a general purpose PC with a dual core Pentium 4 processor, running at 2.8 GHz. A system-on-chip design, consisting of several ARM7 cores, is assumed for the worst-case execution time analysis. For these examples, we have assumed that cache miss penalty is 12 clock cycles.

4.7.1 Bus schedule approaches

To evaluate the optimization algorithms, four bus schedule approaches of varying complexity are defined. The least restrictive approach, BSA1, imposes no restrictions at all and is therefore mostly of interest for comparisons with the other approaches. Since there is no requirement for regularity, a BSA1 schedule is composed of only one segment, consisting of a (very complex) round having the same size as the segment itself. Each core can own any number of slots of different sizes, and the order of the slots is arbitrary. An example of a BSA1 bus schedule and its table representation can be found in Figure 4.4.

With the more restrictive BSA2, each core can own at most one bus slot per round. However, the slots in a round can still have different sizes, and the order can be set arbitrarily. Imposing this restriction on the round dramatically decreases the memory needed to store the bus schedule, since the regularity can be used to store it in an efficient fashion. An example of a BSA2 bus schedule is depicted in Figure 4.5. The first segment starts at time unit 0 and ends at time unit 60, immediately followed by the second segment. The main algorithm in this section assumes that a BSA2 bus schedule is used.

BSA3 is as BSA2 but with the additional restriction that all slots in a round must be of the same size, regardless of owner. This further decreases



Slot start	Owner
0	CPU1
10	CPU2
40	CPU1
60	CPU2
70	CPU1
...	...

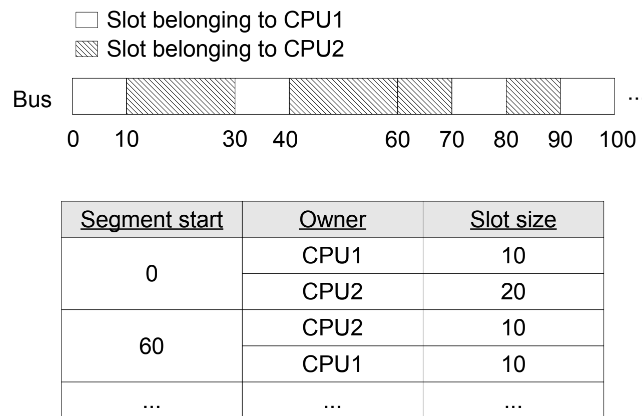
Figure 4.4: BSA1 bus schedule

the amount of memory required on the bus arbiter, since only one size has to be stored for each round, regardless of the number of slots. The order is, however, still arbitrary, just as for BSA2. An example is illustrated in Figure 4.6. The simplified algorithm explained in this section operates on BSA3 bus schedules.

Let us define a fourth approach, BSA4, which is as BSA3 but with the very strong restriction of allowing only bus schedules constituted by one segment (and thus one round). This requires almost no memory at all on the bus arbiter. Since this approach is extremely limited, it is interesting mostly for comparisons with the other approaches, just as for BSA1. An example is shown in Figure 4.7

4.7.2 Synthetic benchmarks

The first set of experiments was done using benchmarks consisting of randomly generated task graphs with 50 to 200 tasks. The individual tasks were generated according to control flow graphs extracted from various C programs, such as algorithms for sorting, searching, matrix multiplication and DSP processing. Experiments were run for configurations consisting of 2 to



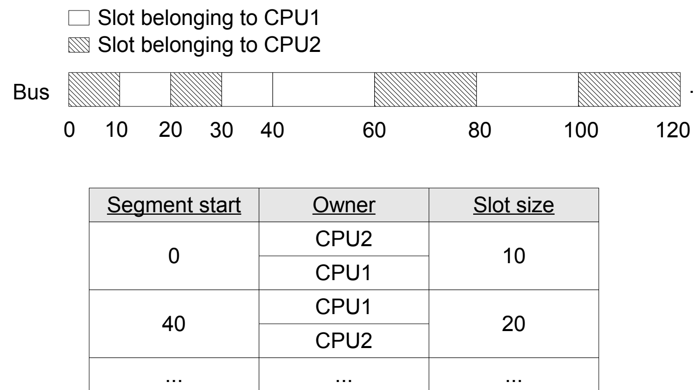
Segment start	Owner	Slot size
0	CPU1	10
	CPU2	20
60	CPU2	10
	CPU1	10
...	...	...

Figure 4.5: BSA2 bus schedule

10 cores, and for each configuration, 50 randomly generated task graphs were used and an average WCRT was calculated.

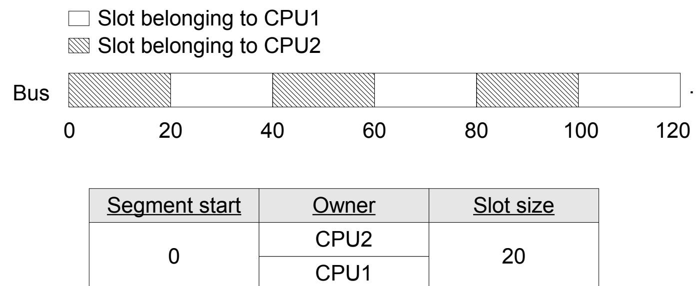
For comparison, this set of experiments was carried out using each of the four bus scheduling approaches defined in Section 4.7.1. In addition, to use as a baseline for evaluating the optimization algorithms, the WCRT was also calculated assuming immediate access to the bus for all cores, resulting in no memory access being delayed. Note that this is an unrealistic assumption, even for a hypothetical optimal bus schedule, resulting in optimistic and unsafe results. This would also be the result obtained from traditional WCET analysis techniques which do not model shared buses.

The result of experiments is depicted in Figure 4.8. The diagram corresponding to each bus scheduling approach represents how many times larger the respective average WCRT is, in relation to the baseline. As can be seen, not surprisingly, BSA1 produces the shortest worst-case response times. This is expected, since the corresponding bus schedules have no restrictions with respect to flexibility. The results produced by BSA2 and BSA3 are, however, not at all far behind. This shows that the price for obtaining regular bus schedules, which can be fitted into memories with a relatively small capacity, is very low. The poor flexibility provided by BSA4, on the other hand, is not enough, and for large bus schedules, the results become inferior.



Segment start	Owner	Slot size
0	CPU2	10
	CPU1	
40	CPU1	20
	CPU2	
...	...	...

Figure 4.6: BSA3 bus schedule



Segment start	Owner	Slot size
0	CPU2	20
	CPU1	

Figure 4.7: BSA4 bus schedule

In a second set of experiments, BSA2 and BSA3 bus scheduling approaches are compared, since they are the important alternatives from a practical viewpoint. In particular, it is interesting to see the efficiency of these policies for applications with different cache miss patterns. A cache miss pattern of a particular task is, in this context, characterized by the standard deviation of the set of time-intervals between all consecutive cache misses. Three classes of applications, each one representing a different level of cache miss irregularity, were created. Every application was composed, according to a randomized task graph, by 20 randomly generated tasks, and each class

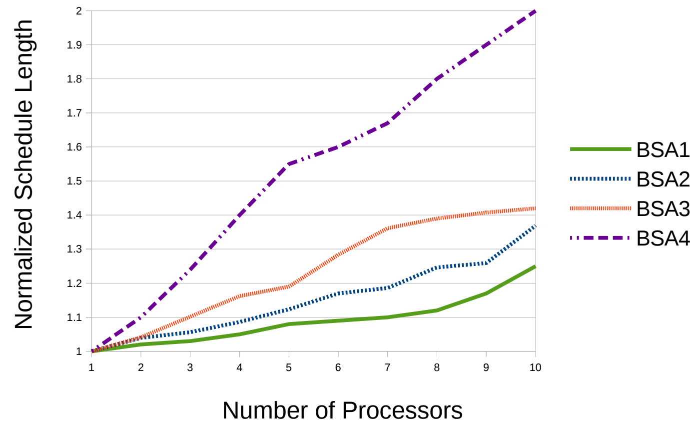


Figure 4.8: Comparison of the four bus access policies

contained 30 applications. For all tasks, the average distance between consecutive caches misses was 73 clock cycles.

The first class of applications was generated with a uniformly distributed cache miss pattern, corresponding to a standard deviation of 0 clock cycles. The other two classes had a more irregular cache miss structure, corresponding to standard deviations of 50 and 150 clock cycles, respectively. Just as for the previous set of examples, the unsafe traditional case, where no core ever has to wait for the bus, is used as a baseline. A comparison of the resulting average worst-case response times is shown in Figure 4.9. It is expected that the two approaches produce the same worst-case response times for very regular cache miss structures since, most of the time, all cores will demand an equal amount of bus bandwidth. However, as the irregularity of the cache miss structure increases, the ability of BSA2 to distribute the bandwidth more freely becomes more and more of an advantage.

A third set of experiments were carried out, demonstrating the efficiency of the successive steps of the main bus access optimization algorithm. The same three classes of applications were used as for the previous set, as well as the same baseline. The results are presented in Figure 4.10. The ISS bar

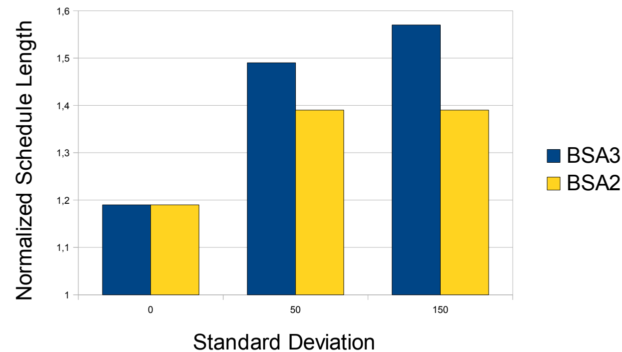


Figure 4.9: Comparison between BSA2 and BSA3

represents the average worst-case response time obtained using the initial slot sizes, calculated as described in Section 4.4.2. The SSA bar corresponds to the average WCRT after slot size adjustments, as described in Section 4.4.3, have been performed as well. Finally, the DS bar shows the result of also applying the concept of density regions, according to Section 4.4.4, in addition to the previous two steps. As expected, density regions are efficient for irregular cache miss patterns, but do not help if the structure is uniformly distributed.

The execution time, for the whole flow, of an example consisting of 100 tasks on 10 cores is 120 minutes for the BSA2 algorithm and 5 minutes for the simplified BSA3 version.

In order to validate the real-world applicability of this approach, a smart phone design has been analyzed. It consists of a GSM encoder, GSM decoder and an MP3 decoder, mapped on four ARM7 cores. The GSM encoder and decoder are mapped on one core each, whereas the MP3 decoder is mapped on two cores. The software applications have been partitioned into 64 tasks, and the size of one such task is between 70 and 1304 lines of C code for the GSM codec, and 200 and 2035 lines for the MP3 decoder. We have assumed a 4-way set associative instruction cache with a size of 4 kilobytes and a direct mapped data cache of the same size. The worst-case response time was

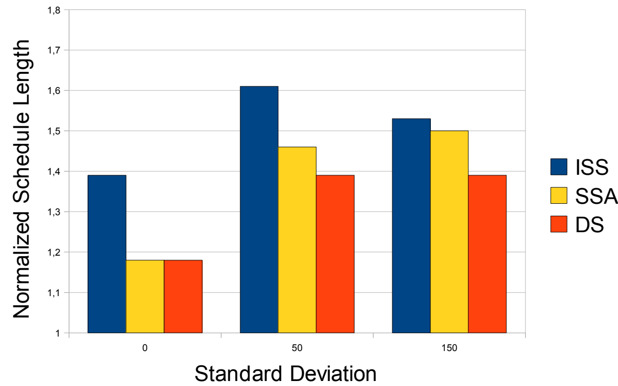


Figure 4.10: BSA2 optimization steps

BSA1	BSA2	BSA3	BSA4
1.17	1.31	1.33	1.62

Table 4.1: Results for the smart phone

calculated using the four bus scheduling approaches defined in Section 4.7.1. For comparison, WCRT is also calculated assuming, unrealistically, immediate access to the bus for each core, as done by traditional WCET analysis techniques. Table 4.1 shows, for each of the four bus scheduling approaches, how many times larger the obtained safe worst-case response time is compared to the unrealistic counterpart. As can be seen, the results are coherent with the experiments in Section 4.7.2.

4.8 A survey of related techniques

In this section, we shall review related literature that aims to accomplish time-predictability on multi-core via compile-time optimization.

The work in [39] performs a compile-time analysis to find memory blocks that are used at most once in the program. Accessing such memory blocks does not heavily affect performance, as they are used only once in the program. However, accessing such memory blocks might affect the content of

shared cache by evicting some other memory blocks. Therefore, the central idea was to bypass the shared cache for all such memory blocks (*i.e.* the set of memory blocks used only once). Another work [56] describes a technique that reduces shared cache conflict by selectively locking memory blocks into L1 caches. The work on software-controlled scratchpad memory [20] aims to reduce shared bus traffic by allocating appropriate blocks into scratch-pad memory. The final goal of such a compile-time strategy is to improve the time-predictability of embedded software on multi-core. A different work [65] 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 bus transaction scheduled during the execution phases of all other tasks, does not suffer any additional delay due to the bus contention. A recent work [62] has proposed a cache management framework for multi-core systems. Such a framework leverages program profiling to analyze the memory access behavior. Once such analysis is performed, frequently accessed memory pages are packed to be allocated in the cache and such memory pages are also locked-down to guarantee deterministic access time. Another approach [81] proposes a shared cache management framework via a systematic combination of cache locking and cache scheduling. Specifically, for unlocked cache lines, cache accesses are scheduled in a fashion to minimize the overall WCET. The work in [87] describes a memory-bandwidth reservation framework to provide predictable worst-case behavior without an appreciable loss of performance. Memory bandwidth is partitioned between guaranteed-service and best-effort service. While a predictable worst-case behavior is possible for guaranteed-service, best-effort service is primarily used to increase the overall throughput of the application. Along the same line, the work proposed in [70] attempts to generate a bus schedule to improve the average-case execution time (ACET) and the WCET of an application *simultaneously*. This technique allows to improve the ACET of an application while keeping its WCET as small as possible.

To summarize, it is evident from the current trend of research that bringing predictability via careful compiler optimizations is also an important as-

pect of multi-core embedded system design. Some of these optimizations take an orthogonal approach to the analysis discussed in section 3, such as the work proposed in [65]. However, there also exist optimization techniques which greatly benefit from the analyses discussed in section 3, such as the bus schedule optimization discussed in this section and the scratchpad allocation discussed in [20]. Moreover, the analysis of WCET may be greatly simplified due to predictable software design. Therefore, we believe that the ideas described in section 3 and in this section are equally important for building fully time-predictable system on multi-core platforms.

5

Time-predictable multi-core architecture for WCET analysis

In section 3, we have discussed WCET analysis methodologies for multi-core platforms. We have seen that resource sharing (*e.g.* shared caches and shared buses) makes WCET analysis for multi-core platforms far more complex than WCET analysis for single core processors. In this section, we describe a number of approaches which aim to accomplish time-predictability via architectural transformations. The primary goal of these approaches is to eliminate (or at least reduce) features that make WCET analysis difficult and, potentially excessively pessimistic. On one hand, such approaches attempt to reduce the pessimism in WCET analysis for multi-core platforms. On the other hand, designing such architectural transformations may lead to the loss of average-case performance. Therefore, careful design choices are crucial for building a time-predictable as well as high performance multi-core architecture. In the following, we shall explore some important works on designing time-predictable multi-core hardware.

5.1 Resource isolation

As we have already mentioned, resource sharing is the key bottleneck for designing an efficient and precise WCET analysis framework for multi-core

platforms. Therefore, one way to improve the time-predictability is to reduce (or eliminate altogether) the amount of possible interferences at the hardware level. This will isolate the different tasks running on multi-core platforms. As a result, WCET analysis for a single task can be carried out independently of any other task running on the system. Such an architecture was developed under the MERASA project [63]. The purpose of this architecture was to provide a multi-core platform where both hard real-time and non hard real-time tasks can execute at the same time, while providing WCET analyzability only for the hard real-time tasks. This multi-core architecture ensures that any request to a shared resource by a hard real-time task is always bounded. This upper bound on the shared resource latency can be taken into account for a *sound* WCET analysis of hard real-time tasks. In the following, we shall describe some critical design choices of this architecture to support WCET analyzability.

5.1.1 Time-predictable access to shared resources

Shared bus access policies

The MERASA architecture designs a two-level arbiter – intra-core and inter-core. As the name suggests, intra-core bus arbiters maintain the bus requests from individual cores and the inter-core bus arbiter decides which of these requests will be granted the bus. By maintaining the bus arbiter for each core, the architecture ensures that a bus request from a particular core will be independent of the number of waiting bus requests from other cores. To bound the bus access delay from a hard real-time thread, the architecture provides a *round-robin* arbitration policy. Note that several hard real-time threads may coexist with non hard real-time threads. Therefore, for the sake of predictability, the inter-core arbiter prioritizes bus requests from hard real-time threads. Given these design choices, we can bound the bus access time from any hard real-time thread as follows: In the worst case, any request from a hard real-time thread (say HRT) might be delayed by all other HRTs executing in parallel. This amount of delay is bounded by $(N-1) \cdot S_l$, where N is the number of cores and S_l is the number of cycles alloted for each bus transaction. Moreover, the request from an HRT may appear just after a bus transaction is scheduled from a non hard real-time thread. In this case, the request from

an HRT might be delayed by a non hard real-time thread for one bus transaction. However, as any request from an HRT is prioritized over requests from non hard real-time threads, an HRT request cannot be delayed more than S_l cycles by a non hard real-time thread. Therefore, the upper bound on the bus access time for an HRT can be bounded by $N \cdot S_l$ cycles.

Once such an upper bound has been derived, it can directly be used for WCET analysis. In particular, WCET analysis tools for single core processors can easily be configured to use these upper bounds. However, given the predictable design, one can use the customized WCET analysis methodologies as discussed in section 3. Such customized WCET analysis will greatly improve the precision in WCET prediction by considering the arrival of bus requests from each core and not always blindly accounting the upper bound ensured by the micro-architecture (*i.e.* $N \cdot S_l$ cycles).

Shared cache access policies

Shared caches are, in general, partitioned into several banks. Different banks can transfer memory blocks in parallel, thus improving the overall throughput. However, if more than one memory requests are made to the same bank, shared cache access might be delayed due to the *bank conflict*. To ensure a predictable access time, the MERASA architecture employs the same policy as used for the bus arbitration. Different requests scheduled to a cache bank are served in a round-robin fashion among hard real-time threads (say HRT) and requests from HRT are always given higher priority than requests from non hard real-time threads. As a result, shared cache access time for any HRT can be bounded using the same logic explained in the preceding paragraph.

Shared cache allocation policies

Finally, shared cache is partitioned among hard real-time threads. This eliminates inter-core interferences. As a result, any WCET analysis for single core processors can be used by configuring the cache size appropriately. However, the MERASA architecture allows sharing the cache for non hard real-time threads. Specifically, MERASA investigates two different partitioning techniques: *(i)* bank-wise partitioning, where a hard real-time thread is given a set of banks which no other thread can use, and *(ii)* column-wise partition-

ing, where a hard real-time thread is given a set of cache *ways* in a similar fashion. Note that for bank-wise partitioning, bank conflicts are automatically removed. However, for column-wise partitioning, different hard real-time threads may access the same bank. Therefore, the shared cache access time is bounded as discussed in the preceding paragraph.

5.1.2 Time-predictable memory controller and interconnect

Besides the MERASA project, there have been different works under the T-CREST project to design time-predictable memory controllers and network interfaces. For instance, a time-predictable design of network-on-chip (NOC) has been discussed in [72]. A more recent work discusses a reconfigurable and time-predictable memory controller [32]. The basic intuition behind both works is to use a time-division-multiple-access (TDMA) scheme. A TDMA scheme is used to ensure that the service time of any bus request is independent of the bus contention. Specifically, the work in [32] analyzes the memory requirement of different tasks and configures a TDMA-based arbiter with appropriate parameters (*e.g.* bus slot allocation to tasks). Besides, the arbiter can be reconfigured via a reconfiguration protocol at appropriate intervals. This reconfiguration aims to improve the overall performance in the presence of varying memory requirement over time. In a similar way, the work in [72] discusses a TDMA-based design of NOC for hard real-time systems. This NOC guarantees an upper bound on communication delay by isolating each communication and providing connection between any two processing cores. Besides, this work also discusses the generation of static schedules to improve the WCET.

From the preceding discussion, we can conclude that TDMA-based/round-robin schemes have been used extensively for designing predictable multi-core hardware. This is due to the reason that such schemes are inherently predictable and they greatly simplify the underlying WCET analysis.

5.2 Usage of software controlled memory

Due to the inherent difficulty in analyzing caches, software controlled memories have been adopted in several embedded systems. These software con-

trolled memories are usually called *scratchpad*. Scratchpad is a fast on-chip memory and it is explicitly controlled by the user or managed by the 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. Since scratchpad memory contents can be controlled by a compiler, each memory access to scratchpad becomes predictable. Besides, as discussed in [13], using scratchpad memory leads to a reduced area and energy consumption compared to caches. Unfortunately, the use of scratchpad memory comes with a cost. Managing scratchpad memory by the user is cumbersome and also error-prone. It also requires rewriting the existing application to utilize the scratchpad memory. Compiler support is critical to systematically allocate appropriate memory blocks into scratchpad memory. In the past, researchers have studied the use of scratchpad memory for improving WCET on single core processors [77]. In the following, we shall describe a few related works on scratchpad allocation for multi-core processors. These works aim to improve the WCET prediction by appropriately allocating memory blocks in a shared scratchpad.

Partitioned scratchpad among multiple cores

The work in [79] studies scratchpad allocation techniques for concurrent software in multi-core systems. Specifically, the underlying architecture contains a private scratchpad for each core. However, multiple tasks may run on one core and, thereby, share the private scratchpad of the core. Memory blocks, which cannot be allocated in the scratchpad, are fetched from slow DRAM memory. The compiler decides the content of the scratchpad via several heuristics. The aim of these heuristics is to improve the worst case response time (WCRT) of the overall system. To improve the sharing strategy among different tasks, the work in [79] also studies *overlay* of memory contents. In particular, WCRT analysis is used to determine the execution-time interval of each task and check the interference among different tasks. If the execution interval of two tasks cannot overlap, they are assigned the same scratchpad space. Such a strategy increase the utilization of scratchpad space, which, in turn leads to an improved worst case behavior. Finally, the work in [79] also inserts artificial *slack-time* to reduce task interference. Of course, introduc-

tion of additional *slack-time* may increase the WCRT. However, reduction of task interference may increase the utilization of scratchpad space which, in turn, may improve the WCRT. Therefore, an iterative approach has been proposed which continues to insert the *slack-time* as long as WCRT improves. The work in [79] was primarily designed for private scratchpads. However, the allocation strategies in [79] were developed to work in the presence of multiple coexisting tasks. Therefore, such allocation strategies can also be investigated in the context of a shared scratchpad among multiple cores, where the coexisting tasks can run in the same core or in different cores.

Shared scratchpad among multiple cores

A more recent work [20] studies a multi-core architecture with shared scratchpad space. In this architecture, each core has a private scratchpad, however, it can also access scratch pads of different other cores (called remote scratchpads) via a fast on-chip communication. The work in [20] investigates scratchpad allocation techniques to improve the overall worst case response time (WCRT) of the application. In particular, the work in [20] extends and improves the work proposed in [79] along two different directions. First, due to the nature of scratchpad sharing among multiple cores, compilers can migrate memory blocks to a remote scratchpad if a core has high workload. Secondly, the work in [20] shows the influence of bus traffic in deciding the content of scratchpad memory. In particular, this work integrates the shared bus model discussed in Section 3.2.4 and it gradually allocates memory blocks into scratchpad memory to reduce the bus traffic.

In summary, scratchpad memories are a promising alternatives to cache memories in multi-core. However, managing scratchpads requires extensive compiler support. Specifically, for multi-core platforms, scratchpad allocation decisions face similar challenges as in WCET analysis. Such scratchpad allocation techniques must consider the tasks executing on different cores and their access requests to shared resources (*e.g.* shared bus traffic as shown in [20]). An interesting direction would be to consider a multi-core architecture with both shared caches and scratchpads. The problem here is to compute the set of memory blocks having high interference in the shared cache and selectively allocate such memory blocks in the scratchpad to improve the WCET.

5.3 Extension of instruction set architecture (ISA)

A different line of work aims to extend the instruction set architecture of a processor with time constraints. With this goal in mind, the precision timed machine (PRET) has been developed [16]. The PRET machine extends the ISA with temporal semantics. A few instances of such timing control are as follows:

- Ensure that a block of code takes *at least* a specified amount of time.
- During execution, if current time has exceeded a specified budget, throw an exception handler.
- Ensure that a block of code takes *at most* a specified amount of time.

Whereas the first two points are relatively easy to integrate in the ISA, the last point requires WCET analysis. To implement the above timing control at the ISA level, some key extensions have been proposed, as follows:

- $delay_time$ $\$t$: This ensures that the following instruction cannot be executed before time $\$t$.
- $exception_on_expire$: This throws an exception if the current time exceeds a specified time budget (deadline violation).
- $mtfd$ $\$t$: This extension ensures that the current time is $\leq \$t$ whenever this instruction is executed.

Note that the $mtfd$ instruction requires static WCET analysis methodologies to ensure that the execution time till the invocation of $mtfd$ is bounded.

To realize the advantage offered by PRET machines, a programming language PRET-C has been developed [8]. PRET-C is a synchronous language extension of the C programming language. The primary goal with designing such a language is to improve timing predictability by compiling a PRET-C program into a PRET compliant ISA [8].

In the presence of multi-core processors, however, time-predictability is seriously affected. In the context of PRET machines, therefore, it is harder to realize $mtfd$ instructions. As a result, besides the ISA extension, several

components of a multi-core processor need to be carefully designed to ensure time-predictability. Recent work (*e.g.* in [68]) discusses the bank level partitioning of memory hierarchy, including dynamic random access memory (DRAM). Besides, the multiprocessor PRET machine, as discussed in [16], also considers TDMA-based arbitration schemes for WCET analyzability and a better realization of PRET-specific ISA.

To conclude, we can say that customized hardware features, such as scratchpad memories and predictable arbitration policies can improve the WCET analyzability on multi-core processors. However, such changes should be complemented with efficient compiler schemes, such as sophisticated scratchpad allocation techniques and efficient generation of TDMA schedules to minimize bus delay. Moreover, controlling the time at ISA level may greatly reduce the inherent non-determinism in WCET analysis. For instance, by knowing the specific execution time for an instruction, we can improve the estimation of shared resource interference. This, in turn, improves the WCET analysis precision and WCET guided optimization methods (as discussed in section 3 and section 4, respectively).

6

Discussion and future work

In this Section, we shall summarize the contributions described in this monograph, outline the limitations imposed by current approaches, and we discuss the challenges for future research.

6.1 Summary of recent development

In this monograph, along with a literature survey, we have described recent work to achieve time-predictability on multi cores. Reviewing the current trend of research, we have pinpointed two different directions to accomplish time-predictability on multi-core: *(i)* developing WCET analysis frameworks to analyze shared resources on multi-core and *(ii)* software/architecture transformation to guarantee worst-case timing behavior. We have shown that sophisticated WCET analysis methodologies are pivotal to model complex micro-architectural features. However, it is also worthwhile to note that WCET analysis might be very pessimistic or have extremely high complexity for certain micro-architectural features, such as dynamic bus arbiters. As a result, predictable micro-architectural features also play a crucial role to guarantee worst-case timing behavior. For example, we have shown that TDMA-based arbitration schemes are well-suited for analyzing the worst-case be-

havior of multi-core embedded software. Therefore, we believe that a careful code/micro-architectural transformation will greatly help to build a fairly accurate WCET analysis methodology and such an integrated approach will be crucial to accomplish fully time-predictable multi-core systems.

6.2 Limitations imposed by current approaches

In spite of several contributions described in this monograph, current approaches suffer from many limitations. These limitations need to be overcome before adopting the current approaches in practice. In the following, we shall discuss a number of such limitations.

Virtual memory The discussed analysis techniques do not explicitly model memory management unit (MMU). MMUs are common in embedded processors (*e.g.* ARM) and they are used to employ spatial separation between tasks in the main memory. In particular, a developer writes embedded software without worrying about the placement of the respective code and data in main memory. At runtime, MMUs translate program addresses (or *virtual addresses*) to actual *physical addresses* in the main memory. In such a fashion, MMUs play an integral part in designing predictable embedded software. However, the presence of MMU requires changes in the analysis methodologies described in the monograph. This is primarily because of two reasons. First of all, address translation may vary at runtime. As a result, static analysis techniques (*e.g.* cache analysis), which relies on static prediction of accessed memory blocks, have limited applications in the presence of MMUs. This problem can be alleviated by assuming a fixed mapping between virtual and physical addresses. Secondly, address translation induces additional delay. To reduce this delay, translation look aside buffers (TLB) are used. TLB works as a fast cache memory and it can accommodate a partial address translation table. In case the respective translation is not found in the TLB, the required translation data is fetched from main memory and TLB content is updated. Since accessing main memory is much slower than accessing TLB, the presence of MMU induces additional timing delay. To accurately predict the WCET of an embedded software, these timing delays

need to be modeled. The discussed analysis methodologies assume that the MMU is disabled during the execution of the program.

DRAM timing model The discussed analysis methodologies assume a simplistic DRAM model. However, the design of commercial DRAMs is usually more complex. In general, DRAMs contain multiple banks. Memory requests from different DRAM banks can be serviced simultaneously. However, requests to the same DRAM bank are usually *serialized*. As a result, a memory request might face additional delay due to the congestion at a DRAM bank. This additional delay can be avoided by not allocating the same DRAM bank to multiple threads on different cores. Using this intuition, a recent proposal [86] provides a software-only solution to dynamically allocate DRAM banks and avoid bank sharing among multiple cores. Besides, researchers have also investigated techniques to bound the interference-delay (*e.g.* due to bank conflicts) in DRAM [49]. We believe that such works reduce the limitations of employing WCET analysis techniques on multi-core platforms.

6.3 Other limitations

Apart from the limitations mentioned in the preceding, there exist other factors in commercial processors which make the WCET analysis difficult. For instance, existing processors usually employ more complex bus arbitration policies (*e.g.* work-conserving and priority-based) compared to TDMA-based policies. Although such arbitration policies are preferred for improving average case performance, they adversely affect the timing predictability. To provide hard guarantees on performance, therefore, TDMA-based policies are also used in some commercial designs [31]. Besides, in existing embedded processors, the activities of I/O components also influences the running time of threads on different cores. This is due to the fact that I/O components and processor cores, in general, share the same bus. As a result, I/O activities might delay the processor by occupying the shared bus. This is similar to the problem of bounding the bus delay when several cores share the same bus. As a result, the analysis methodologies for shared buses, as discussed in this monograph, can be employed to take I/O activities into account. If the I/O

components and processor cores are connected via a non-TDMA bus, it is possible to provide temporal isolation between I/O activities and processor activities using software-level solutions [65].

To summarize, several features of commercial multi-core processors impose limitations on WCET analysis techniques. Such features (*e.g.* virtual memory and I/O contention) may potentially make the WCET analysis methodologies excessively pessimistic to be used in practice. Therefore, we believe that hardware and software level solutions are required to reduce such pessimism for applying WCET analysis in practice. Recent efforts in this direction (*e.g.* works in [65, 86, 49]) are promising. Besides, the works discussed in section 5 (*e.g.* works in MERASA [3] and T-CREST [4] projects) justify the need to balance timing predictability and performance at micro-architectural level.

6.4 Analysis pessimism

There are several cases in which the discussed analysis methodologies may potentially lead to overly pessimistic WCET estimation. One such source of pessimism is the model of interactions between branch predictors and caches. In particular, the analysis methodology in Section 3.3.9 conservatively assumes all possible predictions (*i.e.* correct or incorrect) for every program branch. This leads to several merge operations performed in the cache states. Specifically, cache states from both the correct execution path and the mispredicted path need to be merged. Although such a mechanism keeps the analysis simple and fast, this may lead to pessimistic WCET results, as the cache analysis does not use the outcome of branch predictor modeling. Such pessimism can be reduced by systematically integrating the branch predictor information within cache analysis.

The discussed shared-cache analysis assumes any possible interleaving among parallel threads for each shared cache access. This may potentially lead to excessive pessimism if a large task (in terms of code and data size) is generating inter-core cache conflicts. Analysis of shared caches might be improved by systematically ruling out infeasible interleavings and computing a thread-interleaving pattern that may potentially lead to the worst-case inter-core conflicts.

Pipeline modeling may add a high pessimism in the analysis, specifically in the presence of increased instruction-level parallelism, such as in superscalar processors. This happens due to many possible execution patterns even within a single basic block. In particular, with high instruction-level parallelism, the pipeline modeling might be unable to compute a precise timing interval for different pipeline stages. Consequently, the modeling may detect many infeasible resource contentions due to the overlap in the computed timing intervals. This, in turn, may lead to an overly pessimistic WCET estimation of a basic block. Besides, the presence of high instruction-level parallelism may also lead to pessimistic estimation of bus contexts. This might happen due to the consideration of several execution scenarios and the bus contexts generated from them. Overly pessimistic estimation of bus contexts may propagate the pessimism to the WCET estimation. This might occur due to the overestimation in predicting the waiting time to access shared buses. Such pessimism can be reduced by choosing more fine-grained abstractions for pipeline stages, as compared to *timing interval* (*e.g.* a set of possible arrival and completion time for each pipeline stage). In choosing such fine-grained abstractions, it is crucial to balance the increased analysis complexity with decreased analysis pessimism.

6.5 Research challenges in future

We have observed that a fair amount of research has already been generated to accomplish time-predictability on multi-core. However, significant research challenges still exist and such challenges need to be tackled for a solution to be applicable in practice and build industry-strength multi-core embedded systems. In the following, we shall discuss a few such technical challenges.

Analysis scalability for multi-core systems WCET analysis is of high complexity. In the presence of multi-core systems, WCET analysis is, in general, expensive due to the presence of a huge number of microarchitectural states. To handle this problem, existing strategies use several abstractions. On the one hand, such abstractions reduce the complexity of WCET analysis. On the other hand, such abstractions also lead to the reduction in analysis precision. Such a reduction in analysis precision might

be significant with increasing number of cores and therefore, the underlying WCET analysis technique might be very pessimistic. Finding suitable micro-architectural abstractions is challenging, as such abstractions should have reasonable WCET analysis complexity, without appreciable loss of analysis precision. Therefore, WCET analysis scalability (with respect to number of cores) remains a significant challenge in future.

Devising sound WCET analysis for accelerators In recent times, graphics processing units (GPUs) have gained popularity in the context of non-graphic applications. GPUs use massively parallel computing to accelerate an application. While a typical GPU may contain substantially larger number of cores than a general-purpose multi-core processor (CPU), each core of a GPU is much slower than a CPU core. Therefore, the acceleration provided by GPUs significantly depends on the amount of parallel computation. Recent work [61] shows promising results in accelerating non-graphics workloads on embedded GPUs. Therefore, GPUs appear to be a promising alternative to achieve real-time performance, as also discussed in [57]. Unfortunately, for hard real-time applications, usage of GPUs poses several difficulties. This is primarily due to the difficulty in designing a sound WCET analysis framework. Due to the massively parallel computation offered by GPUs, several thousand computations might be active in parallel. This, in turn, increases the amount of non-determinism by several orders of magnitude – adversely affecting the analysis of WCET on GPUs. Alternatively, one can investigate writing and compiling GPU programs in a time-predictable fashion or designing time-predictable GPU hardware. The primary goal is to get a GPU program where WCET analysis can be carried out with reasonable precision. We believe that time-predictability of embedded software for massively parallel processors (*e.g.* GPUs) will be a crucial step towards achieving high performance embedded systems with real-time guarantee.

Multi-threading and concurrency The key to exploit the performance of multi-core system is to run parallel threads on different cores. However, it is quite natural for such multi-threaded programs to communicate via shared memory. Accesses to such shared memory is, in general, protected by synchronization primitives. Whereas shared memory highly simplifies the de-

sign of multi-threaded software, it poses a serious challenge to achieve time-predictability. For cache-based systems, if shared data items reside in caches private to each core, coherence protocols are required to forbid access to *outdated* data items. To maintain such coherency among shared data items, additional bus transactions are required. This makes the bus traffic highly unpredictable, leading to a poor predictability of the overall system. However, coherency among data items can be maintained via bypassing the private store to each core or using a shared scratchpad memory. Of course, such a solution might be useful only with reasonable loss of performance. A more complex situation arises in the presence of synchronization primitives. Although synchronization primitives (*e.g.* locks) are elegant mechanisms to protect shared data access, they lead to unpredictable data access time. This is due to the variable waiting time (*i.e.* the time between requesting a lock and before the lock is granted) required to access a shared data item. Considering a worst-case waiting period may lead to gross overestimations. Therefore, careful WCET analysis techniques, as well as leveraging predictable implementation of synchronization primitives are critically important for time-predictability of multi-threaded software on multi-core platforms.

Bridging the gap between system-level and WCET analysis Since real-time applications are generally multi-tasking, system-level analysis (*i.e.* schedulability analysis) is required to check the timing constraints of the overall application. It is generally assumed that the WCET of each individual task is known *a priori*, meaning that the accuracy of system-level analysis highly depends on the low-level WCET analysis. However, it is worthwhile to note that the WCET analysis also requires inputs from system-level analysis. One such prominent example is the computation of shared cache latency. Shared cache latency depends on the task interference, which in turn is determined via system-level analysis. Our discussion in section 3 shows several such examples. In the past decade, there has been a massive progress in the area of multi-processor scheduling. Thus, leveraging such progress in system-level analysis into low-level WCET analysis is an important topic to be investigated in future. Besides, the impact of different micro-architectural delays (*e.g.* shared cache and shared bus delays on multi-core) on system-level analysis need to be studied at length. Finally, modern embedded soft-

ware runs in the presence of a supervisory software (*e.g.* an operating system). System-level analysis is usually employed at the level of a supervisory software. Therefore, the interaction between an application and hardware in the presence of a supervisory software opens up interesting research opportunities. A recent work in this area [23] investigates the problem on a single-core platform. Extending this direction for a generalized multi-core platform remains an open problem to be solved in future.

7

Conclusions

Ensuring the time-predictability of embedded software on multi-core platforms is an ongoing and important research topic. Through this monograph, we attempt to bring the attention of the research community towards this area. We have extensively discussed some work which addresses several technical challenges in this area. However, as discussed in the preceding section, important limitations still need to be addressed to adopt these techniques for a commercial embedded processor. We have also performed a survey of related works by several research groups. In spite of a number of existing results in this area, significant research challenges still exist for building practical and scalable solutions. In the previous section, based on our experience, we have discussed several of these open challenges. We hope that this monograph will help to build a background for the time-predictability on multi-core platforms. Last but not least, we believe that this monograph will also help in opening up high-quality research activities to address the remaining challenges.

Acknowledgements

Part of the material discussed in this monograph has previously been published in different proceedings and journals. Specifically, part of the discussion in section 3 has been published in [21], [22] and [18]. Besides, some content of section 4 has been published previously in [69], [9] and [70]. Finally, authors of this monograph acknowledge all the co-authors in their prior conference and journal publications ([21, 69, 22, 18, 9, 70]). This work is partially supported by Singapore Ministry of Education grant MOE2013-T2-1-115 and the Swedish National Graduate School on Computer Science (CUGS). These supports are gratefully acknowledged.

References

[1] aiT AbsInt. http://www.absint.com/ait.

[2] The KLEE Symbolic Virtual Machine. http://klee.llvm.org.

[3] Multi-Core Execution of Hard Real-Time Applications Supporting Analysability. http://ginkgo.informatik.uni-augsburg.de/merasa-web/.

[4] Time-predictable Multi-core Architecture for Embedded Systems. http://www.t-crest.org/.

[5] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. Impact of resource sharing on performance and performance prediction: A survey. In *International Conference on Concurrency Theory*, pages 25–43, 2013.

[6] Sebastian Altmeyer, Robert I Davis, and Claire Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Real-Time Systems Symposium*, pages 261–271, 2011.

[7] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. In *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 153–162, 2010.

[8] Sidharta Andalam, Partha Roop, and Alain Girault. Predictable multithreading of embedded applications using PRET-C. In *International Conference on Formal Methods and Models for Codesign*, pages 159–168, 2010.

[9] Alexandru Andrei, Petru Eles, Zebo Peng, and Jakob Rosén. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In *IEEE International Conference on VLSI Design*, pages 103–110, 2008.

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

[11] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastien, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, Accepted for publication.

[12] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, pages 5–23, 2004.

[13] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *International Symposium on Hardware/software Codesign*, pages 73–78, 2002.

[14] Christoph Berg. PLRU cache domino effects. *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[15] Bach Duy Bui, Rodolfo Pellizzoni, and Marco Caccamo. Real-time scheduling of concurrent transactions in multidomain ring buses. *IEEE Trans. Computers*, 61(9):1311–1324, 2012.

[16] Dai N. Bui, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Jan Reineke. Temporal isolation on multiprocessing architectures. In *Design Automation Conference*, pages 274–279, 2011.

[17] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[18] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multi-core platforms. *ACM Transactions on Embedded Computing Systems*, Accepted for publication (An earlier version appeared in IEEE Real-Time and Embedded Technology and Applications Symposium, 2012).

[19] Sudipta Chattopadhyay and Abhik Roychoudhury. Unified cache modeling for WCET analysis and layout optimizations. In *IEEE Real-time Systems Symposium*, pages 47–56, 2009.

[20] Sudipta Chattopadhyay and Abhik Roychoudhury. Static bus schedule aware scratchpad allocation in multiprocessors. In *ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems*, pages 11–20, 2011.

[21] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems*, 49(4):517–562, 2013 (An earlier version appeared in IEEE Real-time Systems Symposium, 2011).

[22] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *International Workshop on Software and Compilers for Embedded Systems*, pages 6:1–6:10, 2010.

[23] Lee Kee Chong, Clément Ballabriga, Van-Thuan Pham, Sudipta Chattopadhyay, and Abhik Roychoudhury. Towards parallel programming models for predictability. In *IEEE Real-time Systems Symposium*, 2013.

[24] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[25] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. 2004.

[26] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263.

[27] EG Coffman Jr and Ronald L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[28] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, 2000.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of programming languages*, pages 238–252, 1977.

[30] Georgia Giannopoulou, Kai Lampka, Nikolay Stoimenov, and Lothar Thiele. Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In *International Conference on Embedded Software*, pages 63–72, 2012.

[31] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design & Test of Computers, IEEE*, 22(5):414–421, 2005.

[32] Sven Goossens, Jasper Kuijsten, Benny Akesson, and Kees Goossens. A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In *International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, 2013.

[33] Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In *Static Analysis Symposium*, pages 120–136. 2009.

[34] Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Euromicro Conference on Real-Time Systems*, pages 155–164, 2010.

[35] Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In *International Workshop on Worst-Case Execution Time Analysis*, pages 23–35, 2010.

[36] Daniel Grund, Jan Reineke, and Gernot Gebhard. Branch target buffers: WCET analysis framework and timing predictability. *Journal of Systems Architecture*, 57(6):625–637, 2011.

[37] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In *International Workshop on Worst-Case Execution Time Analysis*, pages 136–146, 2010.

[38] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Real-Time Systems Symposium*, pages 57–66, 2006.

[39] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *IEEE Real-time Systems Symposium*, pages 68–77, 2009.

[40] Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *IEEE Real-time Systems Symposium*, pages 456–466, 2008.

[41] Damien Hardy and Isabelle Puaut. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture*, 57(7):677–694, 2011.

[42] Christopher Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert Van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2-3):129–156, 2000.

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

[44] Benedikt Huber and Martin Schoeberl. Comparison of implicit path enumeration and model checking based WCET analysis. In *International Workshop on Worst-Case Execution Time Analysis*, 2009.

[45] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, 2011.

[46] Ilog, Inc. Solver CPLEX, 2003. `http://www.ilog.fr/products/cplex/`.

[47] Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. Performance debugging of Esterel specifications. In *International conference on Hardware/Software codesign and system synthesis*, pages 173–178, 2008.

[48] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Euromicro Conference on Real-Time Systems*, pages 3–12, 2011.

[49] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Raj Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

[50] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[51] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. Pipeline modeling for timing analysis. In *Static Analysis Symposium*, pages 294–309, 2002.

[52] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007. `http://www.comp.nus.edu.sg/~rpembed/chronos`.

[53] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29(1):27–58, 2005.

[54] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, 2006.

[55] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *IEEE Real-time Systems Symposium*, pages 254–263, 1996.

[56] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48(6):638–680, 2012.

[57] Björn Lisper. Towards parallel programming models for predictability. In *International Workshop on Worst-Case Execution Time Analysis*, pages 48–58, 2012.

[58] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *International Symposium on Code Generation and Optimization*, pages 136–146, 2009.

[59] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-time Systems Symposium*, pages 12–21, 1999.

[60] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *IEEE Real-time Systems Symposium*, pages 339–349, 2010.

[61] Arian Maghazeh, Unmesh D Bordoloi, Petru Eles, and Zebo Peng. General purpose computing on low-power embedded GPUs: Has it come of age? In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2013.

[62] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 45–54, 2013.

[63] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *International Symposium on Computer Architecture*, pages 57–68, 2009.

[64] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Fast exploration of bus-based on-chip communication architectures. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 242–247, 2004.

[65] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011.

[66] Rodolfo Pellizzoni and Marco Caccamo. Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. *IEEE Trans. Computers*, 59(3):400–415, 2010.

[67] Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 51–60, 2008.

[68] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *International Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108, 2011.

[69] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *IEEE Real-time Systems Symposium*, pages 49–60, 2007.

[70] Jakob Rosén, C Neikter, Petru Eles, Zebo Peng, Paolo Burgio, and Luca Benini. Bus access design for combined worst and average case execution time optimization of predictable real-time applications on multiprocessor systems-on-chip. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 291–301, 2011.

[71] Erno Salminen, Vesa Lahtinen, Kimmo Kuusilinna, and Timo Hamalainen. Overview of bus-based system-on-chip interconnections. In *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, volume 2, pages II–372, 2002.

[72] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *International Symposium on Networks on Chip*, pages 152–160, 2012.

[73] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224, 2010.

[74] Rathijit Sen and YN Srikant. WCET estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 203–212, 2007.

[75] SPIN. SPIN Model Checker, 1991. `http://spinroot.com/spin/whatispin.html`.

[76] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *International conference on Compilers, architecture, and synthesis for embedded systems*, pages 132–140, 2001.

[77] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET centric data allocation to scratchpad memory. In *Real-Time Systems Symposium*, pages 10–pp. IEEE, 2005.

[78] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Design Automation Conference*, pages 358–363, 2006.

[79] Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. Scratchpad allocation for concurrent embedded software. *ACM Transactions on Programming Languages and Systems*, 32(4):13, 2010.

[80] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.

[81] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*, pages 157–167, 2013.

[82] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 309–322, 2004.

[83] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problemâĂŤoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36, 2008.

[84] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):966–978, 2009.

[85] Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.

[86] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

[87] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64, 2013.

[88] Mohamed Zahran, Kursad Albayraktaroglu, and Manoj Franklin. Non-inclusion property in multi-level caches revisited. *International Journal of Computers and Their Applications*, 14(2):99, 2007.