CACHE DESIGN AND TIMING ANALYSIS FOR PREEMPTIVE MULTI-TASKING REAL-TIME UNIPROCESSOR SYSTEMS

A Dissertation
Presented to
The Academic Faculty

by

Yudong Tan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2005

CACHE DESIGN AND TIMING ANALYSIS FOR PREEMPTIVE MULTI-TASKING REAL-TIME UNIPROCESSOR SYSTEMS

Approved by:

Vincent J. Mooney III, Committee Chair
School of Electrical and Computer Engineering
Georgia Institue of Technology

A. P. Sakis Meliopoulos
School of Electrical and Computer Engineering
Georgia Institue of Technology

David Schimmel
School of Electrical and Computer Engineering
Georgia Institue of Technology

Sudha Yalamanchili
School of Electrical and Computer Engineering
Georgia Institue of Technology

Milos Prvulovic
College of Computing
Georgia Institue of Technology

Date Approved: April 5, 2005

To my parents

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to everyone who made this thesis possible. First and foremost, I would like to give special thanks to my adviser, Professor Vincent Mooney, for his patience and guidance during my Ph.D. study at Georgia Tech. With his knowledge and experience, he has guided me to successfully achieve my research objective.

Second, I would like to thank my thesis committee members, Professor Sudha Yalamachili, Professor A. P. Sakis Meliopoulos, Professor David Schimmel and Professor Milos Prvulovic for their critical evaluation and valuable suggestions.

Third, I would like to thank all my colleagues in the Codesign group for their friendship, especially Pun H. Shiu who helped me greatly during my early period at Georgia Tech.

Finally, I would like to thank my parents for their love and support throughout my entire life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADPCM	Adaptive Differential Pulse Code Modulation.
CIIP	Cache Index Induced Partition.
CRPD	Cache-Related Preemption Delay.
DMA	Direct Memory Access.
FFT	Fast Fourier Transform.
IDCT	Inverse Discrete Cosine Transform.
ILP	Integer Linear Programming.
ISS	Instruction Set Simulator.
LRU	Least Recently Used.
MPEG	Moving Picture Experts Group.
OS	Operating System.
PCB	Printed Circuit Board.
PCFG	Program Control Flow Graph.
PrS	Program Segment.
QoS	Quality of Service.
RMS	Rate Monotonic Scheduling.
RTOS	Real Time Operating System.
SoC	System-on-a-Chip.
TLB	Translation Look-aside Buffer.
WCET	Worst Case Execution Time.
WCRT	Worst Case Response Time.

SUMMARY

The objective of this thesis is to tighten worst-case timing estimation technology for real-time systems executing on uniprocessors with caches. Such technology includes both custom cache design as well as novel analysis techniques. Specifically, we focus on analyzing the impact of cache behavior with respect to preemptions and interrupts. In conjunction with formal analysis, we design a prioritized cache in order to reduce the unpredictability in Worst Case Response Time (WCRT) analysis introduced by caches.

Real-time systems are widely used in Digital Signal Processing (DSP) applications, telecommunication devices, automobiles and robotics. In a conventional embedded real-time system, digital hardware used typically includes a processor core, several customized hardware units and memory. Software running on the processor communicates and cooperates with customized hardware units which provide fast and precise response times. With the quick development of technology, embedded processors become more and more powerful by exploiting advanced features, such as pipelining and caching, that earlier belonged to high-end processors. As a result, more complicated software can be implemented and executed on an embedded processor. For example, multiple tasks supported by a Real-Time Operating System (RTOS) can run on a uniprocessor to provide more versatile functionalities. As compared to customized hardware, software is more flexible and easier to develop, which can dramatically reduce design time and cost; however, a penalty is always paid in that software is typically an order of magnitude or more slower than a custom hardware implementation or the same application/algorithm.

For a real-time system, the most important feature is satisfaction of timing constraints. Especially in a hard real-time system, missing deadlines might cause disastrous results. Thus, designers must analyze the timing properties of software and customized hardware

in the system and guarantee that all tasks can be completed before their deadlines. Usually, customized hardware units have more strict timing characteristics. For software running on a processor, timing analysis is complicated, especially when advanced features in modern processors, such as caching and pipelining, are present.

The execution time of a software task relates to many factors. For example, a software task may execute along different paths because of branches. The branches can possibly depend on input data which cannot be determined in advance. Furthermore, memory access time may be non-deterministic if caches are used. Therefore, although implementing functionalities with software in an embedded system shortens development cycles, it worsens the timing analysis for the system.

In this thesis, we propose a novel worst case timing analysis approach for a preemptive multi-tasking uniprocessor real-time system. We focus on preemption related cache behavior analysis. In a multi-tasking system, multiple tasks, which are scheduled by an RTOS, are allowed to run concurrently. We address the typical case where each task is assigned a priority. A low priority task can possibly be preempted by tasks with higher priorities. During preemptions, the cache lines used by the low priority task may be evicted by cache lines used by high priority tasks. Thus, the low priority task has to reload such evicted cache lines if the lower priority task needs to use those cache lines after recovering from preemption. Such cache reload cost caused by preemptions is called Cache Related Preemption Delay (CRPD). CRPD increases the WCRT of low priority tasks. As a result, the schedulability of the entire system may also be affected.

We propose a new method to analyze CRPD by integrating inter- and intra-task cache eviction analysis techniques. It turns out that the address trace of a task induces a set of possible cache locations to which the addresses can be mapped; we refer to such a set as a Cache Index Induced Partition (CIIP). A novel approach based on CIIP is used for analyzing inter-task cache interference. CIIP is proposed in this thesis for the first time. Path analysis is also applied in our approach to tighten WCRT estimate. We develop

several applications to test the performance of our approach. The experiments show that our approach can achieve a reduction of up to 32% in WCRT estimate as compared to prior approaches.

Inter-task cache interference brings unpredictability in timing analysis because additional cache reload cost is introduced by preemptions. We can customize cache allocation policy in order to reduce inter-task cache interference. A prioritized cache is proposed in this thesis. WCRT analysis is simplified by using a prioritized cache. The experimental results show that task WCRT can be reduced up to 26% as compared to conventional set associative caches.

CHAPTER I

INTRODUCTION

In this thesis, we present a novel cache-related worst-case timing analysis approach for a preemptive multi-tasking uniprocessor real-time system. Specifically, we focus on preemption-related cache behavior analysis. The worst case timing estimate is tightened significantly in our approach as compared to prior approaches.

Furthermore, we design a prioritized cache to reduce inter-task cache interference for high priority tasks. We apply our worst-case timing analysis approach to the prioritized cache. By using a prioritized cache, cache behavior analysis in a preemptive multi-tasking system is simplified; thus, the worst-case timing estimation of tasks can be tightened.

In this chapter, we elaborate our motivation, formally state the problem we address and give a brief introduction to the thesis.

1.1 Motivation

The research presented in this thesis is motivated by the importance of timing analysis for software tasks in a real-time system. A software task can be a process or a thread. In this section, we first give an overview of characteristics in a embedded real-time system. Then, we explain the necessity and difficulty of timing analysis for software tasks in a preemptive multi-tasking real-time system.

A typical embedded real-time system usually consists of software and hardware. Digital hardware includes memory, a processor core, reconfigurable logic and custom logic. The processor provides a facility for executing software. Figure 1 gives a hardware platform example for an embedded real-time system. In this example, some digital signal processing functions such as Fast Fourier Transform (FFT), Moving Picture Experts Group (MPEG)

[45] standard video/audio decoding and Inverse Discrete Cosine Transform (IDCT) are implemented with customized hardware. A Digital Signal Processor (DSP) is also provided for additional digital signal processing ability. Reconfigurable logic can be used to implement other hardware such as Input/Output (I/O) functions. A Direct Memory Access (DMA) unit is provided for communication between memory and external devices. A processor with a Level 1 (L1) cache and a Level 2 (L2) cache is used to run software. Such a hardware platform can be integrated in a System-on-a-Chip (SoC) or a Printed Circuit Board (PCB).

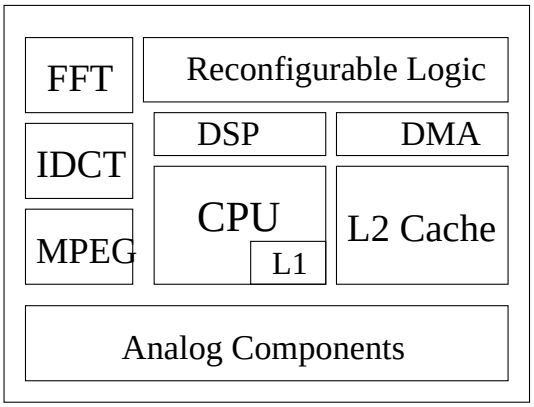


Figure 1: A hardware platform in an embedded system

Real-time systems are distinguished from other systems by the fact that typically most tasks in a real-time system have to meet some timing constraints. In some real-time systems, missing task deadlines can possibly cause disastrous consequences. Therefore, it is critical to guarantee that all timing constraints can be satisfied. Some real-time systems allow deadlines to be missed under some circumstances; however, meeting all deadlines in a tight schedule can possibly enhance Quality of Service (QoS) and improve utilization of resources. Performance and correctness of a real-time system rely on timing. This requires designers to analyze the behavior of tasks and acquire safe upper bounds on task execution time.

In a real-time system, hardware units such as custom logic and reconfigurable logic as

shown in Figure 1 have strict timing properties. Once designed, their behaviors are straightforward and predictable. Therefore, to improve real-time behavior, we can implement in hardware some functions that are traditionally implemented in software. For example, [44] gives a hardware-software real-time operating system framework in which some Operating System (OS) functionality such as spin locks [1], memory management [48, 49] and deadlock detection [24, 25, 50] are implemented in hardware. Some RTOS schedulers in hardware are presented in [26]. By transitioning OS functionality into hardware, time overheads spent on system services such as memory management, lock variables (semaphores) and deadlock detection can be reduced dramatically. The behavior of these hardware units has strict timing properties which are critical to real-time design and analysis. Besides OS components, applications such as MPEG encoding/decoding [62] and wavelet transforms [3] can also be implemented with hardware; indeed, the field of hardware/software codesign is composed of the general study of tradeoffs between implementing any set of applications in hardware, software or some judicious mix.

However, hardware is usually difficult to design, debug and upgrade. Also, it is usually impractical to transition all software to hardware. Thus, a typical real-time system possesses a processor to run software. The processor provides a very flexible platform for integrating software with custom hardware. Software is much easier to develop and thus has a short time-to-market period (as compared to hardware).

Software design is more flexible in terms of design changes and product evolution. However, as compared to hardware, the execution time of software is much more difficult to predict, especially when the architecture of the processor used becomes more and more complicated as a result of introducing out-of-order execution, caching, pipelining and dynamic branch prediction techniques. These advanced features initially only existed in high-end processors, but have been migrating to embedded processors in the past decade or so. In this thesis, we focus on cache-related timing analysis for a real-time system executing on a pipelined uniprocessor with an L1 cache.

Software applications can be accelerated significantly by using caches. Cache use has become a major factor to bridge the bottleneck between the relatively slow access time to main memory and the ever increasing clock rate of today's processors. A cache exploits temporal and spacial locality in memory access patterns of software applications. Cache performance is degraded when multiple memory reference streams with different locality compete for the same cache resource. For instance, when there are multiple tasks in a system and preemption is allowed, tasks can interfere with each other in the cache. Cache interference brings uncertainty to memory access time. It is possible to simply assume that the cache is always empty (i.e., the cache is "cold") when each task starts to run. With this assumption, one can obtain an upper bound on memory access time. However, this cold cache assumption is too pessimistic. A schedule of tasks based on pessimistic estimates wastes processing resources. Therefore, we aim to analyze cache behavior and estimate the timing properties of tasks more precisely. The cache-related timing analysis problem addressed in the thesis is formally stated in the next section.

1.2 Problem Statement

In a typical embedded real-time system, multiple software tasks supported by a Real-Time Operating System (RTOS) run on a processor. In this thesis, we only consider uniprocessor, multi-tasking real-time systems. In the rest of this thesis, any "system" uses only a single processor to execute real-time tasks. Figure 2 shows such a system. In this example, three tasks, an image processing task, a communication task and a robot behavior control task are scheduled by an RTOS. Tasks and the RTOS run on a processor. Additional hardware units are provided for controlling I/O, networking and graphics.

In order to guarantee safety of a real-time system, we need to analyze if a feasible schedule exists for the real-time system under consideration. A feasible schedule is defined as below.

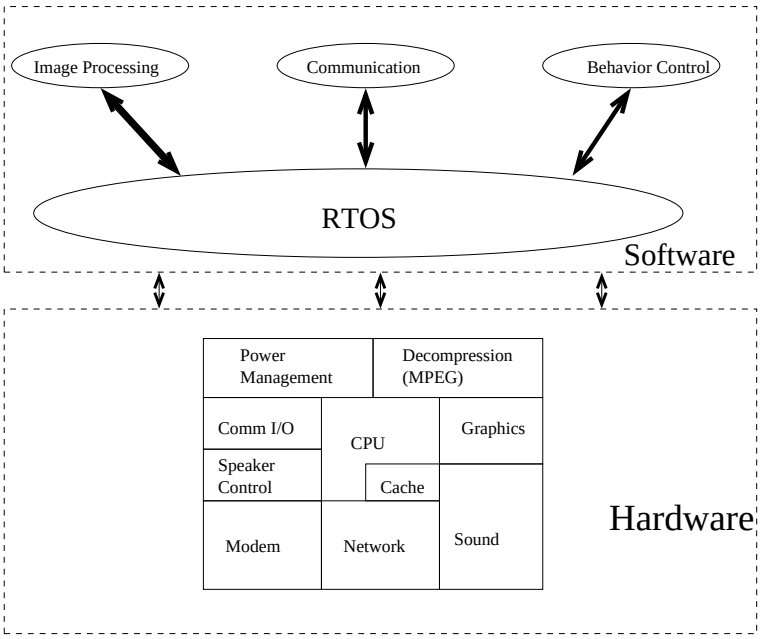


Figure 2: An example of a uniprocessor, multi-tasking real-time system

Definition 1. *Feasible Schedule*: If a schedule of tasks in a multi-tasking real-time system exists such that we can guarantee (prove) that all tasks always complete before their deadlines (excluding situations of hardware failure, e.g., a transistor malfunction), such a schedule is a feasible schedule. If a feasible schedule can be found for a real-time system, such a real-time system is said to be *schedulable*. □

Based on the definition of a feasible schedule, we can define schedulability analysis for a real-time system as below.

Definition 2. *Schedulability Analysis*: Schedulability analysis is any procedure performed to analyze if a feasible schedule exists for a particular real-time system under consideration. □

Schedulability analysis is possible only when the timing properties of tasks are predictable. In a single task system or a non-preemptive multi-tasking system, we can use the Worst Case Execution Time (WCET) of tasks to evaluate schedulability. The WCET of a task is defined as follows.

Definition 3. *Worst Case Execution Time (WCET)*: The WCET is the time taken by a task

to complete its computations in the worst case. The WCET of task T_i is denoted by C_i. □

The WCET of a task does not include the time when the task is suspended because of preemptions or interrupts. Thus, in a preemptive multi-tasking system, only knowing the WCET of a task is not sufficient because a task can be preempted even before its execution is completed. Instead, we need to know the Worst Case Response Time (WCRT). We first define response time as follows.

Definition 4. Response Time: The response time is the time taken by a task from its arrival to its completion of computations. □

Based on the definition of response time, we define WCRT in Definition 5.

Definition 5. Worst Case Response Time (WCRT): The WCRT is the response time of a task in the worst case. The WCRT of task T_i is denoted by R_i. □

After estimating the WCRT of each task, we can perform schedulability analysis for a multi-tasking real-time system.

The response time of a task is affected by many factors such as pipelining, caching and speculative execution. In this thesis, we focus on how the execution time of a task is affected by caches. We assume a uniprocessor system with an L1 cache. An RTOS is provided to support multi-tasking. Each task has a unique priority. Preemptions are allowed. In such a system, a lower priority task can be preempted by higher priority tasks. During preemptions, cache lines used by a low priority task may be evicted by cache lines used by higher priority tasks. When a low priority task resumes after being preempted, it may access cache lines which had been loaded into the cache already but were evicted during the preemption. Thus, the low priority task has to reload such cache lines. Such cache reload increases the delay in the response time of the low priority task. We call this delay *Cache-Related Preemption Delay (CRPD)*. CRPD is defined in Definition 6 and explained in Example 1.

Definition 6. Cache-Related Preemption Delay (CRPD): CRPD is the delay caused by

inter-task cache interference during preemptions. CRPD increases the WCRT of the pre-empted task. □

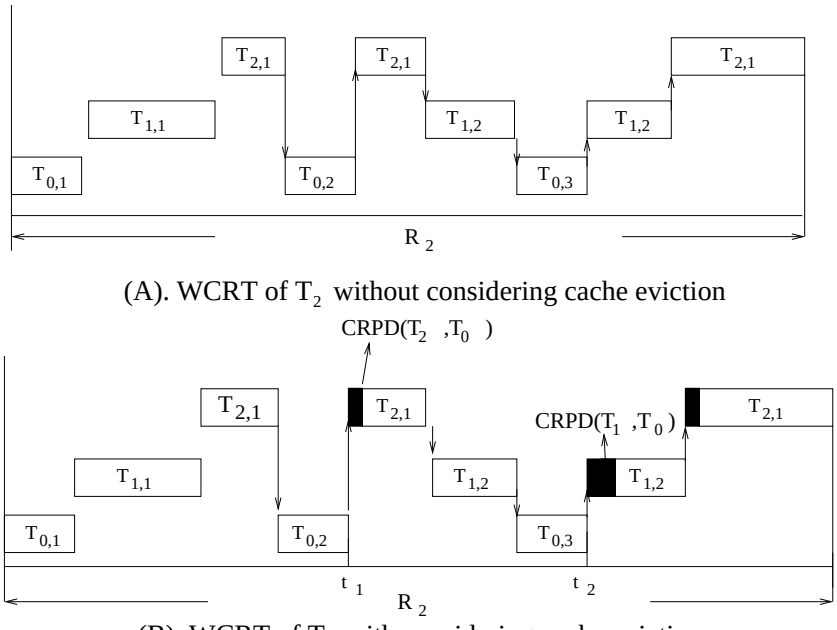


Figure 3: A WCRT example

Example 1: We have three tasks T_0, T_1 and T_2. T_0 is an Inverse Discrete Cosine Transform (IDCT) extracted from an MPEG2 decoder. T_0 is invoked every 4.5ms. T_1 is an Adaptive Differential Pulse Code Modulation Decoder (ADPCMD). T_2 is an ADPCM Coder (ADPCMC). ADPCMC and ADPCMD are taken from MediaBench [23, 39]. ADPCMC has a period of 50ms. ADPCMD has a period of 10ms. Rate Monotonic Scheduling (RMS) is used for scheduling. T_0 has the highest priority and T_2 has the lowest priority. Figure 3 shows this example. In this example, each task has a fixed period. In Figure 3, $T_{i,j}$ represents the j^{th} run of Task T_i; e.g., $T_{1,2}$ represents the second execution of task T_1. All three tasks arrive at time instant 0. However, T_2 is not executed until there are no instances of T_0 or T_1 ready to run. During the execution of T_2, T_2 could be preempted by T_0 or T_1 as shown in Figure 3. The response time of T_2 is the time from 0 to the time when T_2 is

completed. We need to estimate the response time of such a task in the worst case. If we do not consider inter-task cache evictions, the WCRT of T_2 is shown in Figure 3(A). However, because of inter-task cache evictions, T_2 has to reload some cache lines after preemption which imposes an overhead on the WCRT of the preempted task. This overhead is CRPD. Figure 3(B) shows the execution of three tasks when CRPD is considered. CRPD is indicated by black boxes in Figure 3(B). By comparing the response time of Task T_2 in Figure 3(A) with the response time of Task T_2 in Figure 3(B), we find that the WCRT of T_2 is increased in Figure 3(B). Such an increase in WCRT is caused by CRPD. Note that non-zero CRPD occurs only when some memory blocks evicted from the cache during the preemption are required again by the preempted task, which does not necessarily happen for every preemption. □

As shown in Example 1, CRPD affects task WCRT. In order to include CRPD in WCRT analysis for a multi-tasking preemptive system running on a single processor, we need to estimate the number of cache lines that need to be reloaded by the preempted task after each preemption. The number of cache lines to be reloaded determines the CRPD for each preemption. Actual CRPD relates to the memory access patterns and the program structures of both the preempted and preempting task.

In this thesis, we address three problems related to WCRT analysis. First, we aim to analyze CRPD in a preemptive multi-tasking real-time system because CRPD can potentially increase the WCRT of a task. Ignoring CRPD in WCRT analysis may result in the inability to acquire a safe design for a real-time system because schedulability of a system is directly determined by task WCRT.

Second, we want to provide a computationally efficient WCRT analysis approach. As compared to previous work, our WCRT analysis approach should be either more realistic due to incorporating CRPD or less complex in computation (or both).

Third, we want to simplify WCRT analysis in a preemptive multi-tasking real-time system by customizing the cache. The CRPD in our customized cache should be significantly reduced (or even eliminated) as compared to conventional caches. We will adapt our WCRT analysis approach to a particular customized cache (the "prioritized" cache, invented by the author of this thesis) in order to formally analyze the behavior of our customized cache in terms of impact on WCRT.

1.3 Thesis Contributions

This thesis proposes both a new approach to estimate task WCRT in a preemptive multi-tasking real-time system as well as a novel "prioritized" cache to reduce inter-task cache interference. Cache-related preemption delay incorporated in our WCRT estimate is tightened by integrating inter- and intra-task cache eviction analysis. We also apply our WCRT analysis approach to the prioritized cache. The following items are contributions of this thesis.

Contribution 1: A novel approach is proposed to analyze inter-task cache interference. We propose a new algorithm based on Cache Index Introduced Partition (CIIP) to analyze inter-task cache interference. As best we can tell, we are the first to ever define and use CIIP and the concepts inherent therein. CIIP abstracts the mapping relations between memory and cache without requiring knowledge of replacement algorithms used in the cache. By applying this CIIP based algorithm, we can quantitatively analyze inter-task cache interference.

Contribution 2: Inter-task cache eviction analysis is integrated with intra-task cache eviction analysis. We integrate our inter-task cache eviction analysis approach with the intra-task cache eviction analysis approach of Lee et al.[19, 20, 21, 22] in order to improve the estimate precision. The approach of Lee et al. focuses on analyzing CRPD by exploiting the program structure of the preempted task. They do not give a formal analysis for inter-task cache interference analysis, for which we utilize CIIP in our analysis.

Contribution 3: Path analysis is used to improve cache interference analysis. CRPD is related to the program structures of both the preempting task and the preempted task. Inter-task cache interference analysis can be refined by considering the control flow of tasks. We apply path analysis to the preempting task in order to achieve better performance of our WCRT estimation algorithm.

Contribution 4: A new WCRT estimate formula is proposed. Based on CRPD analysis, we propose a new iterative formula to estimate task WCRT. After deriving the WCRT of each task, we can also analyze the schedulability of the whole system. Our approach has a polynomial computational complexity, as compared to the exponential computational complexity of Lee's approach [19, 20, 21, 22].

Contribution 5: A novel "prioritized cache" design is presented to reduce CRPD. We design a prioritized cache in which inter-task cache interference is reduced. As a result, CRPD decreases significantly. We also adapt our WCRT estimate approach to analyze the behavior of our prioritized cache.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 1 introduces the motivation and briefly states the problem addressed in the thesis. Chapter 2 defines some terminology used throughout the thesis. In Chapter 3, we give the background of the research and investigate related previous work. Chapter 4 is an overview of the approach proposed in this thesis. Chapter 5 illustrates the approach to analyze Cache Related Preemption Delay (CRPD). Intra- and inter-task cache eviction analysis are discussed. Path analysis techniques used to tighten the estimate result are also given in Chapter 5. Chapter 6 gives a new WCRT estimate equation based on CRPD analysis. In addition, in Chapter 6, CRPD, which represents the inter-task cache interference caused by preemptions, is incorporated in our WCRT estimate approach. Chapter 7 presents a prioritized cache which can reduce inter-task cache interference by partitioning caches and assigning cache partitions to tasks exclusively. Chapter 8 explains

our experimental setup. Experimental results are given in Chapter 9. The last chapter, Chapter 10, concludes the thesis.

CHAPTER II

TERMINOLOGY

For clarity, in this chapter we define some terminology we will use throughout the thesis. First, we give notation for task periods, priorities and execution time. Then, we introduce a method to represent the program structure of a task. We also explain the mapping relationship between memory and a set associative L1 cache.

2.1 Task Representation and Properties

In this section, we give notation used to represent tasks and task properties such as periods, priorities and execution times.

We assume that a multi-tasking real-time system contains n tasks represented with $T_0, T_1, ..., T_{n-1}$. We focus on a case where a Fixed Priority Scheduling (FPS) algorithm (e.g., the Rate Monotonic Algorithm (RMA)[28] or statically assigned by designers) is used in the system. Each task T_i has a fixed and unique priority p_i. If $p_a < p_b$, T_a has a higher priority than T_b. We assume that the tasks are sorted in the descending order of their priorities so that we have $p_0 < p_1 < ... < p_{n-1}$. Tasks are executed periodically. Each task T_i has a fixed period P_i. T_i arrives at the beginning of its period and must be completed by the end of its period. The Worst Case Execution Time (WCET) of task T_i is denoted with C_i. C_i can be estimated with existing analysis tools such as Cinderella [30] or SYMTA [68]. We use SYMTA to derive C_i. We use $T_{i,j}$ to represent the j^{th} run of Task T_i. The WCET of a task is the execution time of the task in the worst case, assuming there are no preemptions or interruptions. In a preemptive multi-tasking system, WCET alone cannot determine the schedulability (see Definition 1 and Definition 2 in Chapter 1 for what it means to be "schedulable") of tasks in the system because of the existence of preemptions. Thus, our

goal is to provide an approach to estimate the Worst Case Response Time (WCRT), which is defined as Definition 5 in Chapter 1, for every task in a uniprocessor system. The WCRT of task T_i is denoted by R_i.

2.2 CPU Utilization

In a real-time system, CPU utilization as defined in Definition 7 (see below) can help measure how efficiently a processor is being exploited. The CPU utilization of a task can be estimated by using the task's WCET.

Definition 7. CPU Utilization: During a certain time period P, if a task occupies the CPU for a time C in total, the CPU utilization of this task is defined as C/P. The total CPU utilization of a task set is the sum of the CPU utilization of each task in the task set. □

For a periodic task T_i as described in Section 2.1, the CPU utilization of T_i can be estimated with C_i/P_i, where C_i and P_i are the WCET and the period of task T_i, respectively.

When there are branches in a task, task WCET and WCRT are related to the execution path. Thus, in order to find the path or paths along which the task has the WCET, we need to analyze the program structure of the task under consideration. In the next section, we explain the method used in this thesis to represent the program structure of a task.

2.3 Program Control Flow Graph

In this thesis, we perform path analysis on both the preempted task and the preempting task. The path analysis is based on a Program Control Flow Graph (PCFG) which we define here in order to describe the control structure of a program. A PCFG is defined on the basis of basic blocks. A basic block is defined as below.

Definition 8. Basic Block: A basic block is a sequence of assembly instructions with a single entry point, single exit point, and no internal branches. □

Definition 9. Program Control Flow Graph (PCFG): A PCFG is an abstract data structure to represent control dependency among basic blocks in a procedure or a program. A PCFG

is represented with a graph $G = (V,E)$, where $V = \{v_1, v_2, ..., v_m\}$ is the set of nodes and $E = \{e_1, e_2, ..., e_n\}$ is the set of directed edges. Each node in a PCFG is a basic block or program segment (Definition 10) in the procedure or the program. Each edge $e_i = (v_k, v_j)$ represents a control dependence between two nodes, v_k and v_j. A node may have up to two outgoing edges and any number of incoming edges. If a node has more than one outgoing edge, there is a branch in this node. □

As defined above, a node with two outgoing edges contains a branch. Depending on the branch condition, the program can take different paths after this node. We give an example below to explain this situation.

Example 2: Let us consider the PCFG shown in Figure 4(A). Each node is a basic block. Node v_2 has two outgoing edges e_2 and e_3. This means the program has two possible paths following node v_2. The last instruction in v_2 is a branch instruction. According to the branch condition, the program can take either the path with edge e_2 or the path with edge e_3. □

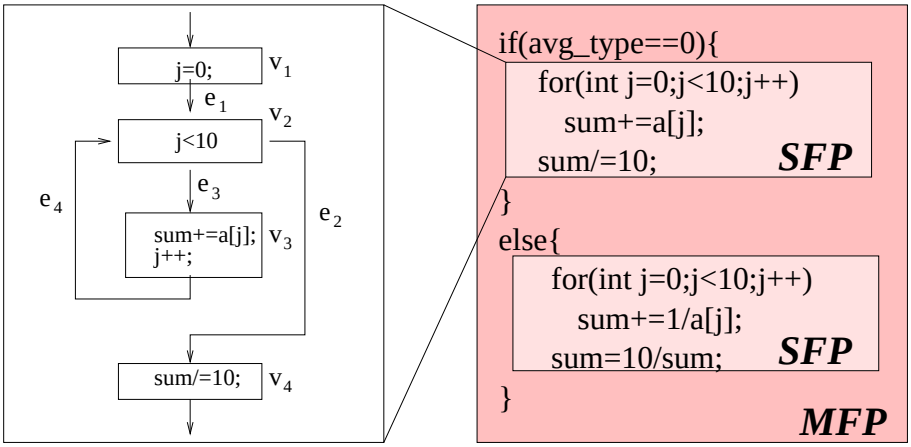


Figure 4: A program segment example

Conventionally, each node v_i in a PCFG represents a basic block in a program. Wolf and Ernst extend the basic block concept to Program Segment (PrS) in [68]. A PrS is defined as follows.

Definition 10. *Program Segment (PrS)*: A Program Segment is a sequence of basic blocks (perhaps non-contiguous) with exactly one entry and one exit. □

Example 3: Suppose we have a program as shown in Figure 4(A). This program consists of a few basic blocks. But the program has only one entry and one exit. Thus, the program in Figure 4(A) is a program segment. □

A program segment may contain other program segments. In the program segment in Example 3, each basic block can be viewed as a program segment because a basic block has only one entry and one exit.

Loops and conditional control structures (e.g., an *if-else* block) can be contained in a program segment. According to the number of feasible execution paths in a program segment, the program segment can be a Single Feasible Path Program Segment (SFP-PrS) or a Multiple Feasible Path Program Segment (MFP-PrS).

Definition 11. *Single Feasible Path Program Segment (SFP-PrS)*: SFP-PrS is defined as a program segment with exactly one path[68]. □

If all loops have constant loop bounds and conditions of all control structures are fixed inside a program segment, there is only one feasible path in a program segment.

Example 4: The program segment in Figure 4(A) contains a loop. The loop bound is fixed. Thus, this program segment only has one path, which means this program segment in is an SFP-PrS. □

Program segments that are not SFP-PrS are MFP-PrS. MFP-PrS is defined as below.

Definition 12. *Multiple Feasible Path Program Segment (MFP-PrS)*: MFP-PrS is defined as a program segment with more than one feasible path[68]. □

Figure 4(B) gives an example of MFP-PrS.

Example 5: In the program shown in Figure 4(B), both *for* loops have a fixed number of iterations. The execution path of the *for* loops are always the same. Therefore, each of the *for* loops is an SFP-PrS. However, the execution path of the overall control structure in Figure 4(B) depends on the input value of *avg_type*. Thus, the

control structure is an MFP-PrS. □

From Example 4, we can see that an SFP-PrS can contain more than one basic block, but shows the same property as a basic block in term of the number of execution paths. Program analysis can be simplified by using SFP-PrS instead of basic blocks. MFP-PrS consists of multiple SFP-PrS instances and thus can be analyzed on the basis of SFP-PrS.

In this thesis, each node in a PCFG corresponds to an SFP-PrS. The SFP-PrS represented by node v_j in the PCFG of task T_i is denoted by $SFP\_PrS(T_i, v_j)$.

2.4 Memory versus Cache

We also need to clarify some definitions regrading caches and memory. A set associative cache is defined by three parameters: the number of cache sets, the number of cache lines in each set (i.e., the number of ways) and the number of bytes/words in a cache line [11]. A direct mapped cache can be viewed as a special set associative cache which has only one way. The sets in a cache are indexed sequentially, starting from 0. All the cache lines in a cache set have the same index. A cache set with an index of i is represented with $cs(i)$. Accordingly, a memory address is divided into three parts: the tag, the index and the offset. We use $idx(a)$ to denote the index of a memory address a.

When a task runs, the task needs to access memory locations where the instructions and data for this task are saved. We assume that there is no dynamic memory allocation in tasks used in this thesis. All memory addresses accessed by a task are fixed. With this assumption, we can use simulation techniques as used in SYMTA [68] to derive the memory footprint of a task. The memory footprint of a task is defined as below.

Definition 13. *Memory Footprint*: The memory footprint of a task is the set of all memory blocks that can possibly be accessed by the task; the memory footprint is calculated considering all possible execution paths of the task. □

Example 6: Suppose we have a task with a PCFG as shown in Figure 5. Depending

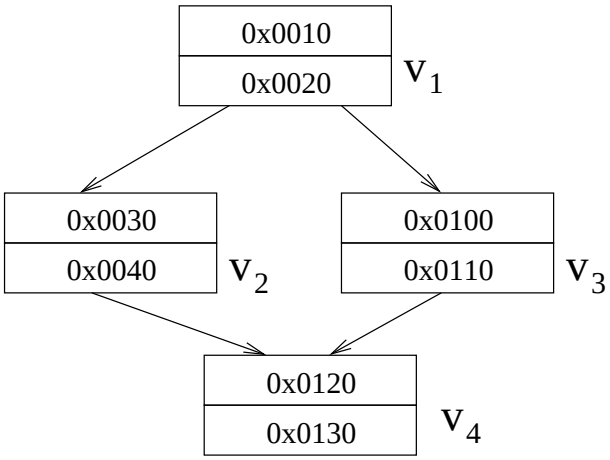


Figure 5: A memory footprint example

on the branch condition in v_1, v_2 or v_3 may be executed. However, the memory footprint of this task covers all the memory blocks that can potentially be accessed by the task when the task runs along all possible paths. In this example, the memory footprint of this task is $\{0x0010, 0x0020, 0x0030, 0x0040, 0x0100, 0x0110, 0x0120, 0x0130\}$. □

When a memory address is accessed, it is possible that only one byte or one word at this address is actually used by the program. However, when the byte/word at this address is loaded into the cache, the whole memory block that contains the byte/word requested is loaded into the cache instead of a single byte/word. A memory block has the same size as a cache line. Example 7 shows the relationship between cache and memory.

Example 7: Suppose we have a 4-way set associative cache with each line in the cache having 16 bytes. The size of the cache is 1KB. Thus, the maximum index of the cache is 15. If a memory address has 32 bits, we can derive each part (i.e., offset, index and tag) of the address for this cache as shown in Figure 6. When a memory address, $0x00000011$, is accessed and the byte at this address is not in the cache, the whole memory block that contains the byte at $0x00000011$ is loaded. The size of the memory block is also 16 bytes, starting from the address with an offset of 0. Because each cache line has 16 bytes, the lowest four bits in the address are

used for the offset. The index is determined by bit 4 to bit 7 in the address, which is 1. Thus, this memory is loaded to a cache line in the cache set with an index 1. □

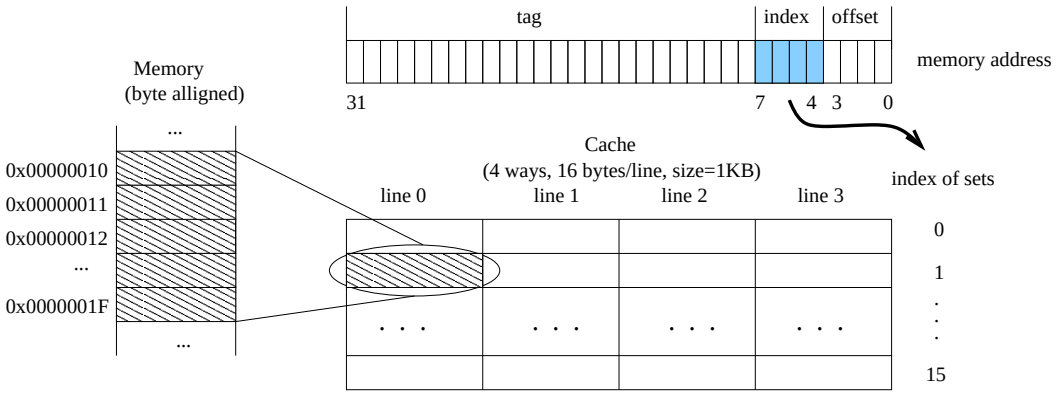


Figure 6: Cache vs. Memory

In the rest of this thesis, when we refer to a cache operation such as a cache load or a cache eviction, we always imply that the operation is performed on a unit of a memory block by default. We do not distinguish "byte/word at a memory address" and "memory block" explicitly; in any case occurring in this thesis, they have exactly the same meaning.

When a memory block with an address a is loaded into a set associative cache, it can only occupy a cache line in the set with an index of $idx(a)$. In this thesis, we assume that a Least Recently Used (LRU) algorithm is used for cache line replacement. As will be explained in Section 5.1, our approach can also be applied to caches with any other replacement algorithm (e.g., a Round-Robin algorithm).

2.5 Cache Partitioning

In this thesis and in prior work, customized cache hardware is typically designed via cache partitioning. In this section, we define what we mean by cache partitioning.

In a multi-tasking real-time system, cache interference among tasks may degrade cache performance and complicate timing analysis. Cache partitioning defined as below is a technique widely used to reduce such cache interference.

Definition 14. *Cache Partitioning*: Cache partitioning is a technique in which a cache is divided into several parts. Each part is called a partition. The size of a cache partition must be a multiple of the size of a cache line. □

When a cache is partitioned, a task can be assigned one or more cache partitions exclusively. Different tasks use different cache partitions. By this means, cache interference among tasks are reduced. However, additional hardware support is needed to implement cache partitioning. The cache controller has to be customized. Figure 7 gives an example of a partitioned cache.

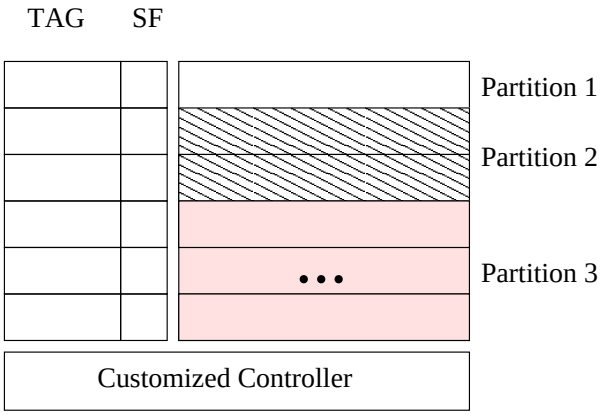


Figure 7: A cache partitioning example

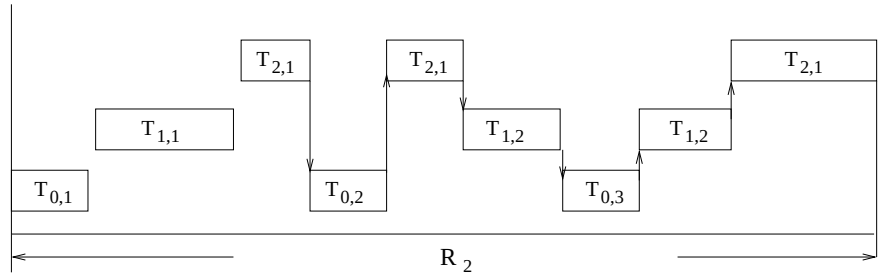
Example 8: Suppose we have a direct mapped cache with 16 cache lines. Each cache line has 16 bytes. We also assume a memory address of size 16 bits. As explained in Section 2.4, we can find that in an address, bit 3 to bit 0 is the offset, bit 7 to bit 4 is the index and the most significant eight bits constitute the tag. If the cache is divided into three partitions as shown in Figure 7, additional hardware overhead is inevitable. For example, Partition 1 in Figure 7 only has one cache line, which is the only possible location when a memory block is mapped to this partition. Thus, we do not need an index to find the cache location. In other words, except for bit 3 to bit 0 which are used for the offset, all the other bits are used as a tag. As a result, the tag size is 12 bits. In Partition 2 in Figure 7, there are only two cache lines. Thus, we only need 1 bit in the memory address to distinguish different cache

lines in this partition. The rest of the bits are used as a tag. Therefore, the tag size in Partition 2 is 11 bits. In order to accommodate reconfigurability of partitions and associated sizes, the tag for each cache line must be able to accommodate the case in which the cache line is alone in a partition. Thus, the tag for each cache line must have the maximum size, which is 12 bits in this example. In addition to the customized tag bit length, this partitioned cache needs a customized cache controller. Most cache partitioning methods allow a portion of cache partitions to be shared by tasks (as opposed to being exclusive to one particular task only). Thus, a Sharing Flag (SF) bit for each cache line may be necessary as shown in Figure 7. Note that some other bits for each line, such as the valid bit and the LRU bits (if LRU is used for cache replacement), are not shown in this example, but they may also be necessary in the cache. □

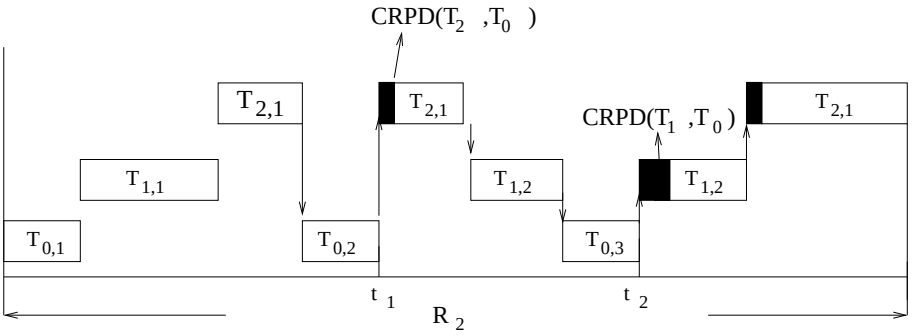
2.6 CRPD

When a multi-tasking real-time system running with a cache allows preemptions, CRPD as explained in Section 1.2 is added to the response time of tasks as a result of cache interference among tasks. We use $CRPD(T_i, T_j)$ to represent the CRPD generated by Task T_j preempting Task T_i, where T_j has a higher priority than T_i. $CRPD(T_i, T_j)$ is related to many factors such as the memory used by task T_i and T_j, cache size, cache management policy and cache miss penalty.

Example 9: Let us consider the tasks in Example 1. CRPD is represented with black boxes in Figure 3, repeated here as Figure 8 with extra information added regarding CRPD. CRPD occurring at time instant t_1 is $CRPD(T_2, T_0)$. CRPD occurring at time instant t_2 is $CRPD(T_1, T_0)$. □



(A). WCRT of T_2 without considering cache eviction



(B). WCRT of T_2 with considering cache eviction

CRPD

Figure 8: A CRPD example

2.7 Summary

In this chapter, we introduce some of the key terminology used in the thesis. Tasks and task properties are represented formally. A representation method based on PCFG is given to describe task structures. We also explain the mapping relationship between memory and a cache. Then, we give a notation for CRPD.

In the next chapter, we investigate previous work in cache design and timing analysis. We also compare our research with previous work.

CHAPTER III

PREVIOUS WORK

.

Worst case timing estimation is critical in a real-time system. However, use of a cache necessarily results in uncertainty in memory access time, which in turn complicates timing analysis. In order to predict the execution time of each software task in a system, three categories of approaches are widely used. First, one can customize caches, compilers and/or operating systems to make task behavior more predictable. For example, diverse cache partitioning schemes are proposed to reduce the unpredictability caused by cache usage. Second, we can also apply formal analysis approaches to analyze the execution times of software tasks by considering cache behavior and exploiting program structures of tasks. Typically, path analysis using Integer Linear Programming and other techniques are performed in order to find the longest path in a task's assembly code, which helps determine the worst case execution time of tasks. The third category is monitoring. One can attach some hardware monitoring units to the processor to assist timing analysis in software. Monitoring is out-of-scope for this thesis. Thus, we only investigate the first two categories of approaches in this Chapter.

In this chapter, we survey various approaches that help predict the timing properties of real-time systems. We further compare and contrast prior work with the research approach and results of this thesis.

3.1 Customizing Caches, Compilers and/or Operating Systems

Correctness in real-time computing depends on not only the logical result but also the time when the computing result is available. The timing behavior of each application has to be known in advance in order to ensure correct real-time system results. Thus, predictability in the underlying hardware operation is required. Unfortunately, standard cache management policies in embedded processors are designed for excellent average performance but lack predictability, especially in a multi-tasking environment. The lack of predictability may lead to an overly pessimistic estimate of the execution time of a task by an order of magnitude or more [7]. Moreover, in a multi-tasking system, cache performance may be degraded because of inter-task cache interference. Inter-task cache interference breaks the locality in memory access patterns of tasks.

Cache interference among tasks in a multi-tasking system can be reduced or even eliminated by customizing cache management policies. Cache management customization can be achieved by partitioning a cache in hardware as introduced in Section 3.1.1 or by software approaches as investigated in Section 3.1.2.

3.1.1 Hardware Cache Partitioning

In this section, we introduce various approaches to cache partitioning via hardware customization. In hardware cache partitioning approaches, a cache is divided into partitions. Some cache partitions are assigned to tasks. Some partitions are shared. Except for shared partitions, tasks use the assigned partitions exclusively. By this means, cache interference among tasks can be reduced.

Kirk presents Strategic Memory Allocation for Real-Time (SMART) cache design in [17, 18]. Each task is assigned to a cache partition the size of which is proportional to the CPU utilization of the task. Some cache partitions are set as shared. Shared cache location can be used by all tasks. The cache controller has to be customized in a SMART

cache. In the SMART cache, cache memory is allocated to each task exclusively based on its CPU utilization. However, only the CPU utilizations of tasks are considered in the cache allocation algorithms, no matter which scheduling algorithm is actually used by the system outside of the cache. Especially, priorities of tasks, which may determine the criticality of tasks in using resources such as processors and caches, are not considered. (Note: a detailed comparison of the SMART cache with the work presented in thesis is available at the end of this Section 3.1.1.)

In [36], Maki et al. propose a data-replace-controlled cache, in which users can control every cache line by setting a cache line in a status of *lock* or *release*. Only when a cache line is in the *release* status can it be replaced. Cache lines in a data cache can be locked in order to prevent replacement. As presented in [36], application programmers have to use additional instructions to specify the data that needs to be locked/released in the cache. Implementing cache lock/release at the level of each individual data element brings one major shortcoming. It is not easy to automate cache lock/release by simply modifying the OS or the compiler because users have to indicate each data to be locked/released in any case. OS or compilers have no knowledge about what data should be locked/released. As a matter of fact, as best this author can tell, no one has published any ideas regarding an automation tool that can help in using the data-replace-controlled cache. This disadvantage increases the work of software development. In [63], Vera et al. propose a cache locking approach to assist static timing analysis. Similar to the approach in [36], in the approach of Vera et al., as presented in [63], each data to be locked has to be known in advance, and furthermore the approach is only applied to data caches while instruction caches are not considered.

Cache lock techniques have been widely adopted in commercial processors. For example, in the Intel XScale processor, cache lines in the instruction cache can be locked if needed [12]. Intel provides special instructions to lock/unlock cache lines. Motorola PowerPC 7400 and Intel 960 also offer similar ability in locking cache lines.

Rudolph et al. [5, 13, 52] propose a column cache. In this design, caches are partitioned at the granularity of "columns." Here, a cache column is the same as a "way" in a set associative cache. Columns in a set associative cache can be assigned to a task exclusively so that the cache lines in these columns are not kicked out. As presented in [5, 13, 52], proper utilization of a column cache requires users to partition the cache statically. The Translation Look-aside Buffer (TLB) has to be modified in order to assign memory used by data in each task to the corresponding columns in the cache. This technique has been targeted only to data caches. This thesis author is not aware of any published work applying this column cache approach to instruction caches.

Juan et al. design a "split" cache in [15]. A split cache can be dynamically split into partitions. Each partition has cache lines of any number that is a power of two. A program can choose a cache partition into which to save the program's data. By carefully choosing the cache partitions, cache misses caused by conflicts can be reduced. Some help from the compiler is required in this design. This technique partitions the cache at a very fine granularity (a cache partition can be as small as one cache line). In a partition of small size, the tag size increases. As explained in Example 8 in Section 2.5, the tag for each cache line must have the maximum size which can accommodate the case where a cache line is a partition. Therefore, as compared to a conventional set associative cache of the same size, a split cache has a larger tag hardware cost because of the increased tag size. A customized cache controller is also needed.

Comparison with prior work in hardware cache partitioning

In this thesis, we present a prioritized cache by applying cache partitioning techniques. In our prioritized cache, cache partitions are allocated to tasks based on their priorities. As compared to previous work, the prioritized cache has the following advantages. First, in our work, cache partitions are allocated to tasks dynamically. Unlike any known prior work, in our approach, task priorities are explicitly considered in cache partition allocation. This strategy conforms to the fact that high priority tasks in a real-time system usually are more

critical and thus require priority in using resources such as the processor and cache. As a comparison, SMART cache [17, 18] does not consider task priority in cache partitioning allocation. A low priority task with a high CPU utilization may consume a large cache partition, which does not benefit highly time critical tasks that may have relatively short run times.

Second, we provide WCRT analysis for our prioritized cache, which allows us to use the prioritized cache in a real-time system safely. This thesis author has been unable to find any prior published approach to estimate task WCET/WCRT using any customized caches. Without WCET/WCRT estimation, it is nearly impossible to analyze schedulability correctly.

Finally, it is easy to use a prioritized cache. We can modify the scheduler in an OS slightly to support a prioritized cache transparently (see Section 7.2.2). In the case when an OS cannot be modified by users, we provide APIs to control a prioritized cache directly by tasks (for details, please see Section 7.2.1). As a comparison, in the cache lock technique of Maki et al. in [36], special instructions are required to be inserted into a program for every cache lock operation, which may impose a heavy burden on designers. As a comparison with a column cache [5, 13, 52], there is no need to customize the TLB in our prioritized cache (please see associated discussion in Section 7.1.2).

3.1.2 Customizing Cache Management with Software Approaches

Apart from customizing cache hardware, one can also reduce or even avoid inter-task cache interference by optimizing memory mapping of software. Memory mapping can be changed by using specific compilers or operating systems.

Wolfe proposes a software-based cache partitioning approach in [73, 74]. The address space of each task is restricted to a certain portion of locations. These portions are scattered over the entire address space for the real-time system. As a result, the address space of each task is mapped to a certain partition of the cache. Cache conflicts among tasks are

avoided by carefully allocating portions of address space to different tasks. Based on this approach, Mueller [46] customizes a compiler to support software cache partitioning with automatic code transformations. After compilation, additional instructions are introduced to the software tasks in order to scatter data/instructions used by each task in the address space intentionally. As a result, cache conflicts are reduced when memory blocks used by each task (including instructions and data) are loaded to the cache. Since memory blocks used by a task are scattered in the entire address space intentionally, this approach can potentially cause memory fragmentation.

Wagner [64] designs some memory-to-cache mapping rules in order to avoid cache conflicts among different data streams. Additional instructions are introduced to help re-map memory. The compiler is customized to automate memory remapping.

Liedtke et al. propose an application-transparent cache partitioning technique in [33]. Cache partitioning is achieved by using a particular memory mapping strategy which is supported by the operating system. The resulting cache partitions can be transparently assigned to tasks for their exclusive use. Löser et al. [35] apply this approach to real-time network applications and demonstrate a significant reduction of cache miss rate by using this OS-controlled cache partitioning technique.

Software cache partitioning techniques can be combined with hardware cache partitioning. May et al. give a caching method for multi-tasking processors [37] by combing a customized cache and a modified compiler[38]. A cache architecture where the cache can be divided into partitions is proposed. Each task is assigned a set of partitions. Such a hardware partitioning approach is combined with a software approach by completely automating partition assignment in the compiler.

Comparison with prior work in software cache usage customization approaches

As compared to software cache usage customization approaches, our prioritized cache is more friendly for users because no custom compilers are required. We also do not need to

be explicitly concerned about the mapping from memory to caches; in other words, assignment of and explicit knowledge of memory addresses do not require special control, which in any case can become quite complicated, e.g., when linking in pre-compiled libraries. As will explained in Chapter 7, we only need to change the scheduler in the OS slightly to transparently support a prioritized cache. Another advantage is that the cache behavior is simplified (and thus easier to analyze) in a prioritized cache, which is favorable for formal timing analysis.

As stated by Dropso in [8], none of the cache management policies (including cache management policies for conventional caches and for customized caches) are superior to any of the others when considering a large variety of general purpose computing scenarios such as scientific computing and real-time control applications. The effectiveness of each policy depends on the target applications. Our prioritized cache targets real-time applications. As shown in our experiments in Chapter 9 in which a robot application and a DSP application are used, our prioritized cache outperforms a conventional set associative cache in terms of WCRT estimate. Chapter 9 shows a reduction of up to 26% in WCRT estimate can be achieved by using a prioritized cache versus using a conventional set-associative cache.

3.1.3 Summary

In this section, Section 3.1, we introduced various approaches to reducing cache interference in a multi-tasking system by customizing cache usage via software (Section 3.1.2) or hardware (Section 3.1.1) methods. Cache interference can be reduced by partitioning caches in hardware or customizing the OS or the compiler to change to a more favorable memory-to-cache mapping.

As compared to prior work, our prioritized cache has the following advantages. First, task priorities are considered in cache allocation, which no prior work of which we are aware does. Second, a prioritized cache only needs minor OS modifications. Third, a

formal timing analysis approach can be easily adapted for our prioritized cache. As a comparison, this author has been unable to find WCET or WCRT analysis approaches for any of the prior customized cache hardware approaches described in Section 3.1.1.

In the next two sections, we investigate previous work in static timing analysis.

3.2 WCET Static Analysis Approaches

Cache behavior can be simplified by customizing cache management polices. However, in order to provide a safe design for a real-time system, we have to know the worst case timing properties of tasks. We can use static timing analysis methods to predict program execution time on a specific processor with a cache. Such methods analyze cache behavior and make restrictive assumptions in order to predict Worst Case Execution Time (WCET) or Worst Case Response Time (WCRT) of tasks in a real-time system. Usually WCET is used for a single task or tasks in a non-preemptive multi-tasking system. In a preemptive multi-tasking system, we have to estimate WCRT for all tasks.

Our research focuses on WCRT analysis. However, we need to estimate the WCET of each task first. Some techniques used in WCET analysis are applied in WCRT analysis as well. For example, cache behavior analysis and Integer Linear Programming (ILP) techniques are important in both WCET [30, 31, 32] and WCRT [19, 20, 21, 22] analysis. In our research, we choose SYMTA [68] to estimate WCET. But as explained in Section 6.2.3, we can use other WCET analysis approaches by simply replacing the WCET estimate derived with SYMTA with the WCET estimate derived with other WCET analysis approaches in our WCRT estimate formula. Various WCET analysis approaches can be found in [2, 9, 10, 30, 31, 32, 58, 65, 67].

Since we choose to build upon SYMTA, we here give an overview of SYMTA. Wolf and Ernst extend the concept of basic blocks to program segments and developed a framework for timing analysis, SYMTA [68, 70, 71, 72]. SYMTA performs WCET analysis by following the steps below.

Step 1. The source code of a task is analyzed to identify SFP-PrS and obtain the PCFG. Each node in the PCFG is an SFP-PrS.

Step 2. Every SFP-PrS is simulated individually by carefully setting the conditions for control structures in a task. The WCET of each SFP-PrS is derived based on the simulation results.

Step 3. ILP equations and an objective function are built by using the PCFG and the WCET of each SFP-PrS.

Step 4. ILP equations are solved to find the maximum value for the objective function. The maximum value of the objective function is the WCET of the task.

We give an example below to explain how to use SYMTA to derive the WCET of a task.

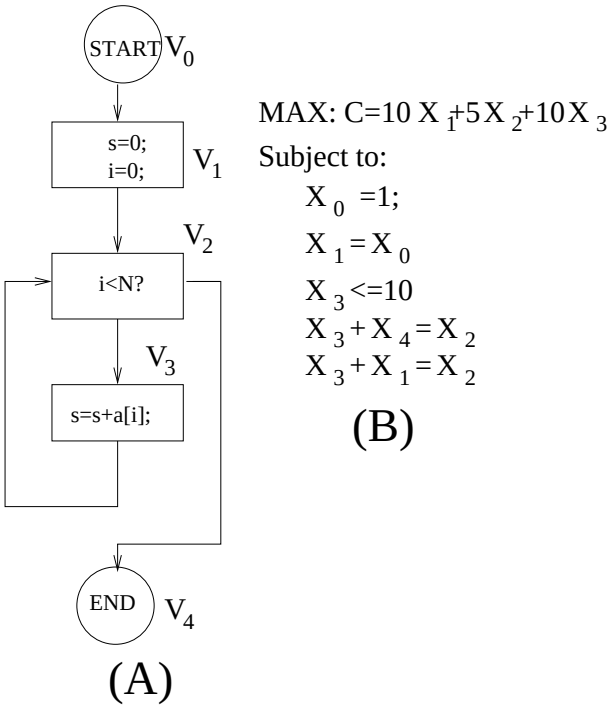


Figure 9: An example of SYMTA flow

Example 10: Figure 9(A) shows a PCFG of a program. Each node, which is represented with V_i, in this PCFG, is an SFP-PrS. The overall program is an MFP-PrS because the loop bound in the program is determined by an input parameter N.

In order to estimate the WCET of this program, we first simulate each node in the PCFG to obtain the WCET of each node. For simplicity, we assume the WCETs of V_0 and V_4 are zero (there may be cost in starting and ending a program in practice, so this assumption may not always be valid). We also assume the WCETs of V_1, V_2 and V_3 are 10, 5 and 10 clock cycles respectively. Notice that we need to obtain an upper bound for each loop in the program; otherwise, it is impossible to estimate the WCET. Here we assume that N is always less than or equal to 10. Based on the PCFG and the WCET of each SFP-PrS derived with simulation as stated in *Step 2* above, we can build ILP equations as shown in Figure 9(B), where X_i represents how many times V_i is executed. The objective is to find the maximum value of function $C = 10X_1 + 5X_2 + 10X_3$, which is the WCET of this program, while $X_i, (i = 0, 1, 2, 3, 4)$ subject to the constraints listed in Figure 9(B). These constraints are defined by the PCFG. For example, $X_3 + X_4 = X_2$ means the number of times that V_2 is executed is equal to the number of times that V_3 is executed plus the number of times that V_4 is executed. By solving this ILP equation, we can find that in the worst case, V_1, V_2 and V_3 are executed once, eleven times and ten times respectively (i.e., $X_1 = 1$, $X_2 = 11$ and $X_3 = 10$), which results in a WCET of 165 clock cycles. □

In this thesis, we always apply SYMTA to derive WCET for each task. WCET is then used in our WCRT analysis approach. Since in this thesis we make no contributions to WCET analysis per se, we do not review the multitude of prior work in WCET analysis. Instead, in the next section, we survey previous work in WCRT analysis and compare with our work in WCRT analysis.

3.3 WCRT Analysis

In this section, we introduce key prior state-of-the-art in WCRT analysis. WCRT analysis is useful in preemptive multi-tasking real-time systems. By using WCRT, we can verify the

schedulability of a system. In Section 3.3.1, we first introduce some basic WCRT analysis approaches in which cache influence is not considered. Then, we survey the prior work of cache related WCRT analysis in Section 3.3.2. In Section 3.3.3, we point out the necessity of performing formal timing analysis for customized caches.

3.3.1 Basic WCRT Analysis

In this section, we introduce important prior research in WCRT analysis. In the works presented in this section, Section 3.3.1, cache influence and context switch cost are not considered. Although these assumptions are not practical, the WCRT analysis approaches listed in this section provide a base for later research presented in this thesis.

Joseph and Pandya [14] give an iterative approach to estimate WCRT for tasks in preemptive multi-tasking systems. The iterative calculation procedure starts with an initial value of WCRT of each task which is set as its WCET. If the iteration converges, the WCRT estimate is derived. If the WCRT estimate of at least one task is greater the deadline or the iteration diverges, a feasible schedule of tasks cannot be found with this method. This approach provides a generic framework for WCRT estimation. In this work, tasks are assumed to have static priorities. Deadlines of tasks cannot be later than their periods. However, some real-time systems are not so stringent and can accept task deadlines greater than task periods. Lehoczky [29] gives an approach to analyze WCRT of tasks in a real-time systems where a given task can have arbitrary deadlines. Tindell et al. [60] extend WCRT analysis by relaxing constraints on deadlines, arrival and release time of tasks. A real-time system with jitters is considered in Tindell's approach. Approaches in [14, 29, 60] simply assume that the WCET of each task is already known and fixed. The impact of cache on WCET/WCRT is not considered in any of the work cited in this section.

Comparison with basic WCRT analysis approaches

All WCRT analysis approaches introduced in this section do not consider the influence of underlying hardware architectures such as pipelining, speculative execution, memory

hierarchy and context switch cost. A feasible schedule based on these WCRT analysis approaches may fail in reality. In our WCRT analysis presented in this thesis, we focus on analyzing the impact of cache interference on WCRT of tasks. Our WCRT analysis gives a more realistic and safe WCRT estimate. In the next section, we introduce key prior WCRT analysis approaches that also consider cache behavior.

3.3.2 Cache Related WCRT Analysis

WCRT analysis approaches in Section 3.3.1 can be extended by considering cache behavior. In this section, we introduce some cache-related WCRT analysis approaches.

Busquests-Mataix et al. propose an approach to estimate WCRT by considering cache behavior in a uniprocessor multi-tasking system [4]. The iterative equation of Tindell et al. in [60] is modified to include Cache-Related Preemption Delay (CRPD) in the approach of Busquests-Mataix et al. [4]. They conservatively assume that all the cache lines used by the preempting task need to be reloaded by the preempted task when the preempted task is resumed. Obviously, this assumption is very pessimistic.

Tomiyama et al. give an approach to calculate CRPD by using ILP [61]. However, they only consider a direct mapped instruction cache and do not address data caches at all.

Lee et al. also give an approach for cache analysis in preemptions [19, 20]; their approach counts the number of "useful" memory blocks by performing path analysis on the preempted task. However, they assume that all "useful" memory blocks of the preempted task are evicted from the cache by the preempting task, which might not be true. For example, if there is no dynamic data allocation in any task and the cache lines used by the preempted task are disjoint with the cache lines used by the preempting task, the cache reload cost induced by preemption will be zero. In the approach of Lee et al., the cache reload cost is still the same as the cost to reload all "useful" memory blocks in the preempted task. Lee et al. enhance their approach in [21, 22]. In [21, 22], all preemption scenarios are explored to find the cache reload cost. The number of preemption scenarios

increases exponentially with the number of tasks. Thus, calculating the cache reload cost for each preemption scenario separately is not efficient to compute. Moreover, although Lee et al. mention that the cache reload cost is calculated based on the intersection of cache lines used by the preempting task and the preempted task, no method is given to show how the intersection is calculated. In the approach of Lee et al. , for each preemption, the program structure of the preempted task is used to calculate useful memory blocks. However, the program structure of the preempting task is not considered at all. As shown in Chapter 5 in this thesis, overestimation may occur if we do not analyze execution paths in the preempting task.

Negi et al. [47] refine the CRPD estimate method used by Lee et al. in [21, 22]. Negi et al. apply path analysis to tighten the CRPD estimate. However, inter-task cache eviction is not considered in their CRPD analysis. Also, WCRT analysis is not mentioned in [47]. This CRPD analysis approach can be applied in a cache related WCRT analysis approach (e.g., the approach of Lee et al. [19, 20, 21]) for a direct mapped cache because the CRPD analysis approach presented in [47] targets direct mapped caches. This CRPD analysis approach may be difficult to use for a set associative cache because cache lines that are deemed conflicting in a direct mapped cache in the approach of Negi et al. may in fact not conflict in a set associative cache. In short, no method for analyzing cache conflicts in a set associative cache is given in their approach, and there seem to be difficulties in modifying their approach to handle caches which are not direct mapped.

Comparison with prior work in cache-related WCRT analysis

In this thesis, we propose an approach for inter-task cache eviction analysis in Section 5.1. A Cache Index Induced Partition (CIIP) estimates the number of cache lines evicted during a preemption (see Section 5.1.1 for details). By using CIIP, we provide a formal method to calculate the intersection of cache lines used by two tasks. This method can be uniformly applied to direct mapped caches and set associative caches. Also, path

analysis is applied to the preempted task in order to tighten the estimate. We furtherer enhance our approach by incorporating "useful" memory block analysis of Lee et al. [19, 20] (see Section 6.2 for details of how we enhance our approach). We integrate inter- and intra-task cache eviction analysis in order to tighten the WCRT estimate. A new WCRT analysis formula is proposed based on cache behavior analysis.

As compared with previous work, our WCRT analysis approach presented in this thesis has the following merits. First, a novel approach using CIIP is proposed to analyze inter-task cache evictions. Inter-task cache eviction analysis is not addressed in the work of Lee et al. nor in any other publication of which we are aware. Second, we propose a new WCRT estimation formula which has a polynomial computational complexity in term of the number of tasks (proved in Section 6.3.2). As a comparison, the approach of Lee et al. has an exponential computational complexity.

All cache related WCRT analysis approaches introduced in this section only consider conventional set associative caches or conventional direct mapped cache; as best we are aware, no extensions of WCRT analysis to customized caches have been published by the authors covered in this section, Section 3.3.2. In the next section, we investigate the problem of WCRT analysis for customized caches.

3.3.3 WCRT Analysis for Customized Caches

Customized caches show benefits in accelerating executions of applications in multi-tasking environments. However, a connection between customized caches and formal WCET/WCRT analysis is still missing. We find no prior WCET/ WCRT analysis approaches for customized caches published. Instead, usually, the effectiveness of cache partitioning methods as mentioned above in Section 3.1.1 are evaluated with experiments. Some typical benchmark applications are executed with cache partitioning approaches. The average execution time or the cache miss rate is used to measure the performance. In [51], Suh et al. give an analytical cache model to analyze the cache miss rate of a partitioned cache. This model

predicts the overall cache miss rate of general applications. The worst case is not considered. Dropso [8] compares some existing customized caches by using an analytical cache model. Again, the analysis targets the average performance of general applications.

Comparison with prior work

We appear to be the first to publish WCRT analysis for a customized cache. As compared to using benchmarks to evaluate the average performance of customized caches, a formal WCRT analysis provides a safe base for applying customized caches in real-time systems. On the contrary, benchmarking cannot guarantee deriving the worst timing properties of a real-time system with a customized cache. Without a knowledge of the worst case timing properties, a system may fail in practice because of violating timing constraints.

3.3.4 Summary of Prior WCRT Analysis Approaches

WCRT analysis provides a safe base for designing preemptive multi-tasking real-time systems. When the underlying hardware architecture is simple (e.g., no cache(s) and no pipelining), estimating WCRT without considering hardware features [14, 29, 60] is acceptable. However, as illustrated in [19, 20, 21, 22, 47, 61], cache effects cannot be ignored in WCRT analysis in modern processors.

As compared to previous work in cache related WCRT analysis, our approach have the following advantages. First, our approach gives a formal method to analyze inter-task cache eviction behavior. Second, we integrate inter- and intra-task cache eviction analysis to tighten the WCRT estimate. Third, a novel WCRT estimate formula is proposed. As compared to WCRT analysis of Lee et al. [19, 20, 21, 22] (the best known approach in prior work), our approach can tighten the WCRT estimate by 32%. The complexity of our approach is $O(n^2)$ where n is the number of tasks, while the approach of Lee et al. has an exponential complexity in terms of the number of tasks.

3.4 Schedulability Analysis

Timing analysis in a multi-tasking system is tightly related to scheduling. By using WCRT estimates, one can analyze schedulability of tasks in a real-time system. In this section, we give references to some previous work in schedulability analysis and then place our approach in context.

Some theoretical work have been presented in schedulability analysis for real-time systems [27, 28, 34]. In this thesis, however, we do not make any new contribution in schedulability analysis. Instead, we simply use Worst Case Response Time (WCRT) to analyze schedulability. We assume that a Fixed Priority Scheduling (FPS) algorithm is used in the system. Each task has a unique priority. Task priorities can be derived by using an existing algorithm such as the Rate Monotonic Algorithm (RMA)[28] or can be assigned by designers directly. We further assume a single processor with a set associative L1 cache and secondary memory (the secondary memory can be either on- or off-chip). We do not consider the problem of handling the problem of missing deadlines. If the WCRT of a task is greater that its deadline, we simply deem that a feasible schedule is not found.

The schedulability analysis in this thesis results from the WCRT estimate derived with our WCRT analysis approach. Because cache influence is considered in our WCRT estimate, the conclusion of schedulability derived with our approach is more realistic and safer as compared to known prior approaches in [28, 34, 47, 60, 61] all of which ignore possible cache effects. As compared with the approach of Lee et al. [19, 20, 21, 22], a better schedule can be derived with our approach because we have a tighter WCRT estimate for each task. A tighter schedule allows utilizing computing resources more efficiently. Furthermore, we may derive a feasible schedule for a system which is deemed not schedulable by the approach of Lee et al.

3.5 Summary of Previous Work

In this chapter, we introduce previous work in WCET/WCRT analysis and customized cache usage design. Cache usage can be customized by partitioning a cache in hardware or manipulating memory-to-cache mapping in software (e.g., via compiler and/or OS modifications). By customizing caches, compilers or OS, inter-task cache interference can be reduced in a multi-tasking system. Thus, the cache behavior becomes more predictable. Cache behavior can also be predicted by using formal timing analysis approaches. By considering cache effects in WCET/WCRT analysis, one can predict timing properties of tasks more precisely. Both of these methods can help design a safe real-time system.

As compared to previous WCRT analysis approaches, our WCRT analysis method presented in this thesis gives a tighter WCRT estimate because both inter- and intra-task cache eviction are included in cache behavior analysis in our approach. Furthermore, our WCRT analysis has a polynomial computational complexity, which is an advantage over the exponential computational complexity in what is, in our opinion, the best prior work in WCRT analysis, the approach of Lee et al. [19, 20, 21, 22].

As compared to previous work in reducing cache interference by customizing caches, compilers and operating systems, our customized cache has the following two advantages. First, a prioritized cache allocates cache partitions according to task priority, which is not considered in previous customized cache methods. Second, a prioritized cache only needs slight modifications in the OS to transparently support a prioritized cache (i.e., without requiring the writing of special user-level code or significant efforts to modify a compiler and/or OS).

As best we can tell, we are also the first to apply a WCRT analysis approach to formally analyze the performance of a customized cache. By formally analyzing the behavior of our prioritized cache, we demonstrate that a tighter WCRT can be achieved by using the prioritized cache as result of reducing cache interference among tasks.

In the next chapter, Chapter 4, we give a big picture of the research presented in this

paper. The main idea behind a prioritized cache and the major steps in our WCRT analysis approach are briefly introduced.

CHAPTER IV

OVERALL APPROACH

In this chapter, we give an overview of the research presented in this thesis. We address two problems related to timing analysis for preemptive multi-tasking real-time systems. First, we propose a novel cache-related WCRT analysis approach for preemptive multi-tasking systems. Our WCRT analysis approach is an extension of the basic WCRT analysis approach introduced in Section 3.3.1. Especially, we focus on incorporating CRPD into WCRT of tasks. Second, the CRPD is caused by inter-task cache interference in essence. We can customize the cache management policy to reduce or eliminate inter-task cache interference. A prioritized cache is proposed for this purpose.

We first give the steps in our WCRT approach in Section 4.1. The main idea underlying our proposed prioritized cache is explained in Section 4.2. Section 4.3 explains the relationship among main chapters in the thesis. Finally, Section 4.4 summarizes this chapter.

4.1 WCRT Analysis

In this section, we give a big picture of the WCRT analysis approach presented in this thesis. Our research focuses on the impact of cache behavior on WCRT estimation. Inter-task cache interference causes CRPD in preemptive multi-tasking systems. CRPD is determined by the number of cache lines to be reloaded after preemptions. Intuitively, we know that the cache lines causing reload overhead after preemption(s) need to satisfy two conditions.

Condition 1. These cache lines are used by both the preempted and the preempting task.

Condition 2. The memory blocks mapped to these cache lines are accessed by the preempted task before the preemption and are also required by the preempted task after the preemption (i.e., when the preempted task is resumed).

Condition 1 implies that memory blocks accessed by the preempting task conflict in the cache with memory blocks accessed by the preempted task. Thus, some of the memory blocks loaded to the cache by the preempted task before the preemption are evicted from the cache by the preempting task during the preemption. This cache eviction involves memory access patterns of both the preempted task and the preempting task. Thus, we call this type of cache eviction an inter-task cache eviction. We use Example 11 to explain Condition 1.

Example 11: Suppose we have a direct mapped cache with 16 cache lines. Each cache line has 16 bytes. As explained in Section 2.4, we use $cs(i)$ to represent a cache set of index i. In a direct mapped cache, there is only one cache line in a cache set. Thus, we use $cs(i)$ here to indicate the cache line with an index of i. We assume that a memory address has 16 bits. Now, we have two tasks, T_1 and T_2. T_1 has a lower priority than T_2 and thus can be preempted by T_2. Figure 10(A) shows a few memory blocks that are accessed by T_1 and T_2. T_1 uses memory blocks at $0x0010$ and $0x0020$ which are mapped to $cs(1)$ and $cs(2)$ respectively. T_2 uses memory blocks at $0x0120$ and $0x0110$ which are mapped to $cs(2)$ and $cs(1)$ respectively. If T_1 is preempted by T_2 after $0x0020$ is accessed (where the arrow S points as shown in Figure 10), $cs(1)$ and $cs(2)$ may be reloaded when T_1 is resumed because $cs(1)$ and $cs(2)$ satisfy Condition 1. Such as a cache reload extends the

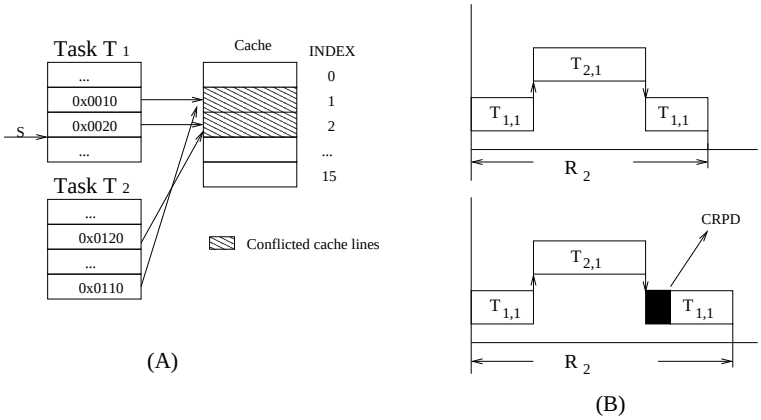


Figure 10: An example of Condition 1

response time of the preempting task, T_1, as shown in Figure 10(B). □

Note that Condition 1 is only a necessary condition for cache eviction. If cache lines evicted by the preempting task during the preemption are not requested by the preempted task after the preemption, these cache lines do not cause cache reload cost. In other words, all cache lines to be reloaded have to satisfy Condition 1, but not all cache lines that satisfy Condition 1 need to be reloaded in reality. If we only apply Condition 1 to look for cache lines that can be possibly reloaded, we can obtain a safe upper bound on cache reload cost. However, this cache reload cost may be overestimated.

Condition 2 reveals that memory blocks causing cache reload cost must have been present in the cache prior to the preemption. Furthermore, these memory blocks must be accessed again by the preempted task after the preemption, thus requiring reload to the cache. These memory blocks are called "useful memory blocks" in the work of Lee et al.[19, 20]. Only cache lines mapped from useful memory blocks potentially need to be reloaded. We can use the algorithm of Lee et al. to find useful memory blocks. Example 12 shows a case in which we can use both Condition 1 and Condition 2 to analyze cache interference between two tasks.

Example 12: Suppose we have two tasks, T_1 and T_2, the memory footprints of which are shown in the context of associated PCFGs as shown in Figure 11. T_2 has a higher priority than T_1. We have a direct mapped cache which has 16 lines with each line of size 16 bytes. As explained in Section 2.4, we use $cs(i)$ to represent a cache set of index i. In a direct mapped cache, there is only one cache line in each cache set. Thus, we use $cs(i)$ here to indicate the cache line with an index of i. T_1 can possibly access memory blocks $0x0110$, $0x0120$ and $0x00F8$, which are mapped to $cs(0x1)$, $cs(0x2)$ and $cs(0xF)$ respectively. T_2 accesses memory blocks $0x10F0$, $0x1110$ and $0x1120$, which are also mapped to $cs(0x1)$, $cs(0x2)$ and $cs(0xF)$, respectively. By using Condition 1 only, we can conclude that $cs(0x1)$, $cs(0x2)$ and $cs(0xF)$ can possibly cause cache reload cost. Now, let us assume T_1 is preempted

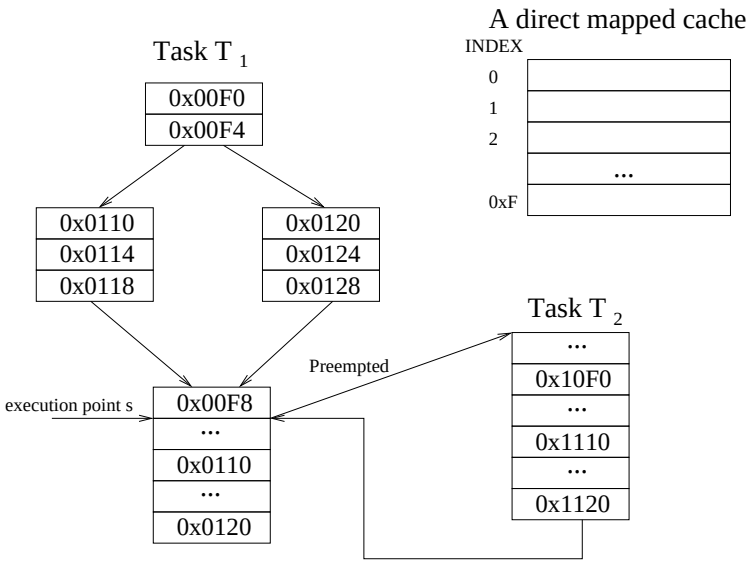


Figure 11: An example of cache interference analysis

by T_2 at the execution point s as shown in Figure 11. By analyzing the memory footprint of T_1, we can find that only memory blocks $0x0110$ and $0x0120$ are loaded to the cache before the preemption and requested by T_1 after the preemption. In other words, only $cs(0x1)$ and $cs(0x2)$ mapped from memory blocks $0x0110$ and $0x0120$ satisfy Condition 2. By combining Condition 1 and Condition 2, we can conclude that only $cs(0x01)$ and $cs(0x02)$ may possibly need to be reloaded in this example. □

Based on the two facts explained, Condition 1 and Condition 2, we can give an overview of our approach presented in this thesis. Our approach has five steps which are shown in Figure 12.

Step 1. In the first step, we derive the memory footprint of each task with the simulation method as used in SYMTA[68]. Here, we assume that there is no dynamic data allocation in any task and that addresses of all data structures are fixed [68]. We also assume a two level memory hierarchy which consists of an L1 cache and a single main memory.

Step 2. In the second step, we perform intra-task cache access analysis (due to Condition 2) on the preempted task to find useful memory blocks accessed by the preempted task. Only

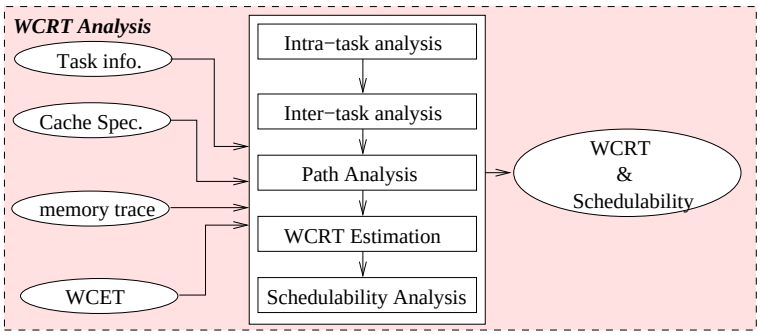


Figure 12: WCRT Analysis Flow

useful memory blocks can possibly cause cache reload delay. Useful memory blocks are calculated on the basis of path analysis for the preempted task.

Step 3. In the third step, we analyze inter-task cache conflicts (due to Condition 1) between the preempted task and the preempting tasks (i.e., all the tasks that have higher priorities than the preempted task). A low priority task might be preempted more than once by a higher priority task, depending on the period of the low priority task as compared to the period of the high priority task. By integrating intra- and inter-task cache eviction analysis, we can estimate CRPD.

Step 4. In the fourth step, in order to tighten the estimate of the number of cache lines to be reloaded, we apply path analysis to the preempting task using its PCFG. By using path analysis, we can refine the CRPD estimate derived in *Step 3*.

Step 5. In the last step, we preform WCRT analysis for all tasks based on the results from the fourth step. An iterative WCRT estimation formula is proposed. If the WCRT of a task converges to a value less than the deadline of this task, the task can meet its deadline. Otherwise, we are not able to schedule this task successfully via utilization of the proposed task priorities and WCRT estimation formula.

Example 13: Let us consider the case in Example 12. In order to estimate the WCRT of tasks T_1 and T_2, we use SYMTA to obtain the program structure, the WCET and the memory footprint (as defined in Definition 13) of each task. Next, we apply Condition 2 to find the useful memory blocks in the preempted task. In this example, T_1 is preempted and

the useful memory blocks are $0x0110$ and $0x0120$. Then, we use Condition 2 to perform inter-task cache eviction analysis. As shown in Example 12, we find that that $cs(1)$ and $cs(2)$ can possibly cause CRPD. Based on this result, we can estimate CRPD caused by T_2 preempting T_1. CRPD can be further tightened by applying path analysis techniques on the preempting task. In this example, path analysis is not necessary because the preempting task T_2 only has one path. Section 5.5 will explain the steps of CRPD estimation in detail. After deriving a CRPD estimate, we incorporate CRPD into WCRT analysis and use our WCRT estimate for each task to analyze the schedulability of such a system. Section 6.2 will explain how these WCRT estimates were derived. □

In our WCRT analysis, cache interference among tasks adds CRPD to the response time of the preempted task. On one hand, CRPD increases the complexity of timing analysis because of increased unpredictability in cache access patterns. On the other hand, CRPD extends the response time of a task, which can possibly result in a task missing its deadline. By manipulating cache replacement policy, cache interference can be reduced or eliminated. In the next section, we introduce a customized cache that can help reduce cache interference.

4.2 Customized Cache Design

In this section, we briefly explain the rationale behind our "prioritized" cache and introduce a prototype of the proposed prioritized cache.

In real-time systems, high priority tasks are usually more critical and thus should be granted priority in using processor, memory, cache and other resources. Based on this intuition, we propose a prioritized cache. The prioritized cache is evolved from a conventional set associative cache. We also borrow the notation of "column" from the column cache [13, 52]. A "way" in a set associative cache is viewed as a column in a prioritized cache. Our prioritized cache is partitioned at the granularity of columns. Some cache partitions are then allocated to tasks according to task priorities. Cache partitions used by low priority

tasks can be used by high priority tasks. Cache partitions used by high priority tasks cannot be used by low priority tasks. In this way, high priority tasks are granted higher priority in using cache resources. In order to prevent the situation where some tasks do not have cache partitions to use, some partitions can be set to be "shared." Shared cache columns can be accessed by any task.

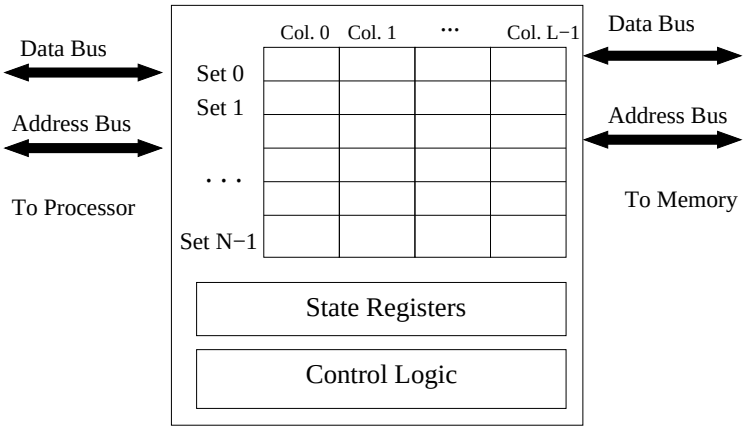


Figure 13: A prototype of a prioritized cache

Figure 13 shows a prototype of a prioritized cache. As compared to a conventional set associative cache, some state registers are added and minor changes are needed in the cache replacement control logic of a prioritized cache. Software support for a prioritized cache can be provided in two ways. First, APIs for accessing the registers in a prioritized cache are provided. By setting values in these registers, users can manipulate a prioritized cache directly. On the other hand, we can modify the scheduler in an operating system slightly in order to support a prioritized cache. With the modified scheduler in an operating system, application developers do not need any APIs explicitly. Instead, they can use a prioritized cache transparently.

We extend our WCRT approach to analyze the behavior of a prioritized cache. WCRT estimation derived from a formal analysis provides a solid base for safely applying a prioritized cache to a real-time system. Example 14 shows the main idea of applying WCRT analysis to a prioritized cache.

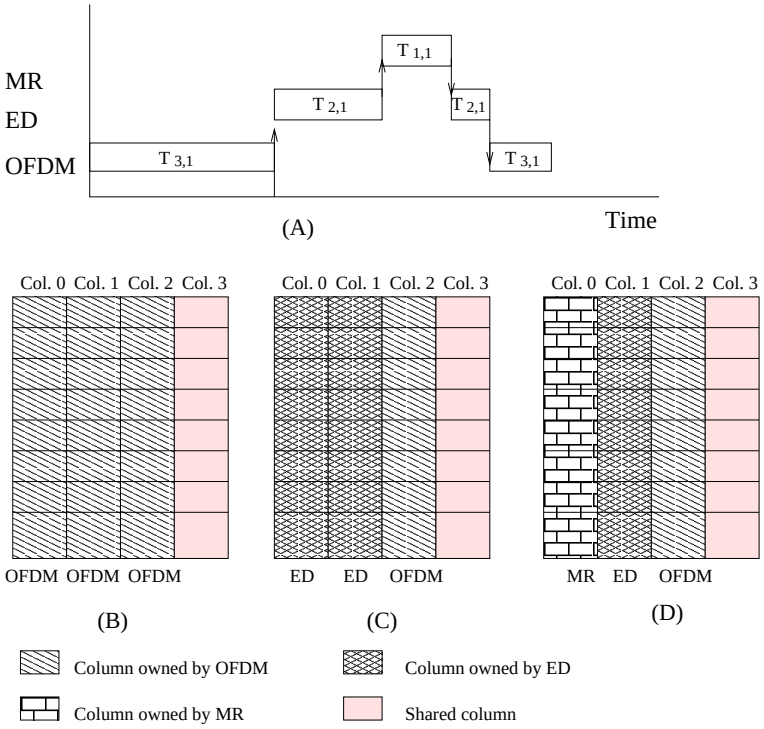


Figure 14: An example of using the prioritized cache

Example 14: Suppose we have three tasks, T_1, T_2 and T_3. T_1 is a Mobile Robot control application (MR). T_2 is an Edge Detection (ED) application. T_3 is an OFDM transmitter application. MR updates the behavior of a robot periodically. ED processes images detected by the robot and OFDM is used to communicate among robots. MR has the highest priority and OFDM has the lowest priority. Also, we assume that a 4-way prioritized cache is used in the system, and one column of the cache is set as shared.

Consider the scenario in Figure 14(A). OFDM runs first. Then, ED arrives and preempts OFDM. ED is then itself preempted by MR.

When OFDM runs, it uses all columns. Column 0, Column 1 and Column 2 are owned by OFDM as shown in Figure 14(B). Column 3 is shared and cannot be owned by any task. After OFDM is preempted by ED, ED uses Column 0 and Column 1 because ED has a higher priority than OFDM. Now, Column 0 and Column 1 are owned by ED as shown in Figure 14(C). OFDM cannot load memory

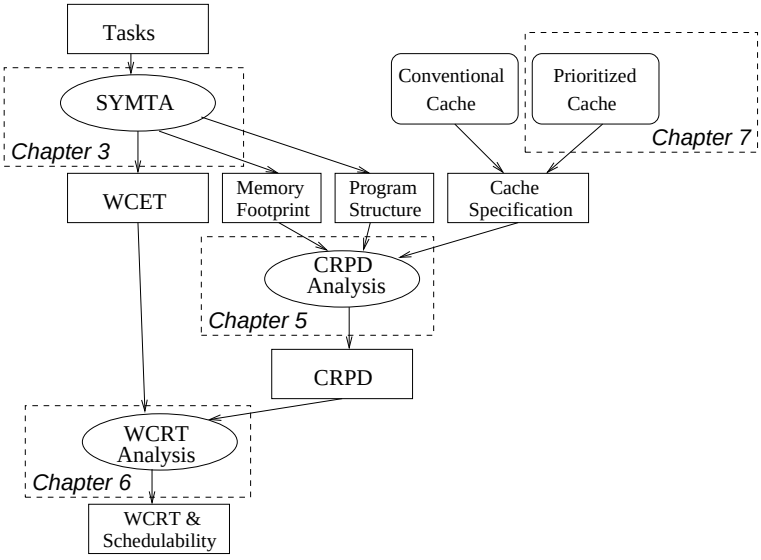


Figure 15: The flow of the thesis

blocks to cache lines in Column 0 and Column 1. However, OFDM can still read cache lines in Column 0 and Column 1 in the case of cache hit. Similarly, after ED is preempted by MR, MR owns Column 0 as shown in Figure 14(D). It turns out that all cache lines required by MR fit into Column 0. □

In this section, we have given a brief overview of the design ideas underlying our prioritized cache. In the next section, we explain the flow of the main chapters in the thesis.

4.3 Research Overview of Chapter Flow

Let us give an overview of how the main chapters in this thesis present the research accomplished. Figure 15 shows the main steps in our approach and the relationship among the central chapters in this thesis. SYMTA is used for WCET estimation in our approach as explained briefly in Chapter 3. By using SYMTA, we can derive the WCET estimate, memory footprint and program structure for each task. Important cache parameters are specified such as associativity and size. The cache could be a customized cache such as a prioritized cache presented in Chapter 7. Based on the memory footprint, the cache specification and the program structure of each task, we perform CRPD analysis as explained in Chapter 5 in

detail. We then incorporate CRPD into WCRT analysis which is elaborated in Chapter 6. By using the WCRT analysis approach presented in Chapter 6, we can estimate the WCRT for each task and evaluate the schedulability for the system.

4.4 Summary

In this chapter, we give an overview of the research presented in this thesis. First, we explain the flow of our WCRT analysis approach in Section 4.1. Then, we introduce the main design ideas behind the prioritized cache in Section 4.2. We also use a graph as shown in Figure 15 to show in Section 4.3 the relationship among main chapters in this thesis.

In the next chapter, we elaborate the CRPD analysis approach used in this thesis.

CHAPTER V

CACHE RELATED PREEMPTION DELAY ANALYSIS

In this chapter, we present a new approach to analyze the CRPD caused by cache interference among tasks. We first give a method to formally analyze inter-task cache interference by using the concept of a Cache Index Induced Partition (CIIP). Then, we combine the intra-task cache eviction analysis approach of Lee et al. [19, 20] with our inter-task cache interference analysis. Next, we use path analysis to refine the cache interference analysis. Finally, we illustrate our method of CRPD estimation based on cache interference analysis.

5.1 Inter-task Cache Eviction Analysis

In this section, we explain our inter-task cache eviction analysis approach in detail. We first propose a technique, CIIP, which abstracts memory-to-cache mapping regardless of the replacement policy used in the cache. Then, based on this technique, we estimate the number of cache lines that cause conflicts between two tasks.

5.1.1 Cache Index Induced Partition

In this section, we elaborate the concept of CIIP and a way of deriving the CIIP based on a memory footprint.

When a task first accesses instructions or data, those instructions and data are loaded to the cache from the memory. In a direct mapped cache, the cache location to which a memory block is loaded is determined uniquely by the index of the memory address. In a set associative cache, the index of a memory address determines the cache set in which the memory block is located. Then, a cache line in that set is selected by a particular cache replacement algorithm such as the Least Recently Used (LRU) algorithm. In any case,

the index of a memory address plays an important role in selecting a cache line for the corresponding memory block.

An obvious observation about caches is that memory blocks mapped to different cache sets will never conflict in the cache. In other words, only memory blocks that have the same index can possibly evict each other because these memory blocks are loaded to the same cache set. Intuitively, we can divide memory blocks into different subsets according to their index.

Suppose we have a set of q memory block addresses, $M = \{m_0, m_1, ..., m_{q-1}\}$, and an L-way set associative cache. The index of the cache ranges from 0 to $N-1$. We can derive N subsets of M as follows.

$$\widehat{m}_i = \{m_k \in M | idx(m_k) = i\},\ (0 \leq i < N) \tag{1}$$

We give an example below.

Example 15: Suppose we have a memory block set $M = \{0x0010, 0x0210, 0x1100\}$ and a direct mapped cache. We have 16-bit memory addresses. The cache has 16 lines with each line of size 16 bytes, thus, as explained in Example 7, bit 3 to bit 0 is the offset and bit 15 to bit 4 is the tag. In this case, we have $\widehat{m}_0 = \{0x1100\}$ and $\widehat{m}_1 = \{0x0010, 0x0210\}$. All the memory blocks in $\widehat{m}_0$ have the same index of 0. All the memory blocks in $\widehat{m}_1$ have the same index of 1. Note that in this example, $\widehat{m}_i = \emptyset$ for all i such that $1 < i < 16$. □

When the memory blocks in the same subset $\widehat{m}_i$ are accessed, these memory blocks are loaded into the same set in the cache because the memory blocks have the same index. Thus, cache evictions can happen among these memory blocks (i.e., with the same index).

We denote $\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\}$, where $\emptyset$ is the empty set and $\widehat{m}_i$ is defined as Equation 1, An example of $\widehat{M}$ is given in Example 16.

Example 16: Considering the memory block M in Example 15, we have $\widehat{M} = \{\widehat{m}_0, \widehat{m}_1\} = \{\{0x1100\}, \{0x0010, 0x0210\}\}$. □

By using $\widehat{M}$, we can define the CIIP of a memory block set M. The definition of CIIP

is given in Definition 15. As best this thesis author can tell, this author is the first to ever define and use CIIP and the concepts inherent therein.

Definition 15. Cache Index Induced Partition (CIIP) of a memory block address set: Suppose we have a set of memory block addresses, $M = \{m_0, m_1, ..., m_{q-1}\}$, and an L-way set associative cache. The index of the cache ranges from 0 to $N-1$. We define the CIIP of M based on the mapping from memory blocks to cache sets, which is denoted by $\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\}$. Each $\widehat{m}_i = \{m_k \in M | idx(m_k) = i\}$ is a subset of M. □

The CIIP of a memory address set categorizes the memory block addresses according to their indices in the cache. Cache evictions can only happen among memory blocks that are in the same subset $\widehat{m}_i$ in $\widehat{M}$, the CIIP of memory address set M. We first defined and introduced CIIP in [57].

Example 17: Suppose we have a set of memory block addresses $M = \{0x000, 0x100, 0x010, 0x110, 0x210\}$. Also, we have a set associative cache as defined in Example 7. Therefore, $0x000$ and $0x100$ have the same index 0. $0x010$, $0x110$ and $0x210$ have the same index 1. So, the CIIP of this memory block address set is $\widehat{M} = \{\widehat{m}_0, \widehat{m}_1\}$, where $\widehat{m}_0 = \{0x000, 0x100\}$ and $\widehat{m}_1 = \{0x010, 0x110, 0x210\}$. Any block in $\widehat{m}_0$ will be loaded into the cache set with index 0 when the memory block is accessed. Any block in $\widehat{m}_1$ will be loaded into the cache set with index 1 when the memory block is accessed. Cache eviction can only happen among memory blocks in $\widehat{m}_0$ or among memory blocks in $\widehat{m}_1$. A memory block in $\widehat{m}_0$ can never be replaced by a memory block in $\widehat{m}_1$ and vice versa because the memory blocks in $\widehat{m}_0$ and the memory blocks in $\widehat{m}_1$ are loaded into different sets in the cache. □

CIIP connects memory blocks and cache lines by using indices in memory addresses without considering the cache replacement policy. This approach can be applied to any type of set associative cache. CIIP can be used to analyze inter-task cache conflicts.

One important CIIP property is that $\widehat{M}$ is a "proper partition" of M, where we define proper partition as follows according to the definition of partition in [6, 16]. This property

is important to show exact, unduplicated distribution of memory addresses with $\widehat{M}$ (i.e., within a CIIP).

Definition 16. Partition: If $\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\}$ is a partition of M, the following three conditions have to be satisfied [6, 16].

(1). For all $\widehat{m}_i$ in $\widehat{M}$, $\widehat{m}_i \neq \emptyset$, which means no subset in $\widehat{M}$ is empty.

(2). For all $\widehat{m}_i$, $\widehat{m}_j$ in $\widehat{M}$, $\widehat{m}_i \bigcap \widehat{m}_j = \emptyset$, which means all subsets in $\widehat{M}$ are disjoint.

(3). $\bigcup \widehat{m}_i = M$, which means all subsets in $\widehat{M}$ exactly cover the original set M. □

We can prove $\widehat{M}$ is a partition of M as below.

Proof: The first condition is trivial.

$\forall \widehat{m}_i, \widehat{m}_j \in \widehat{M}, i \neq j$, if $m_a \in \widehat{m}_i$, $m_b \in \widehat{m}_j$, we have $idx(m_a) = i$ and $idx(m_b) = j$. Thus, $m_a \notin \widehat{m}_j$ according to the definition in Equation 1. For the same reason, we have $m_b \notin \widehat{m}_i$. Therefore, $\widehat{m}_i \bigcap \widehat{m}_j = \emptyset$. Condition (2) is thus also satisfied.

According to the definition in Equation 1, if $\widehat{m}_i \in \widehat{M},\ (0 \leq i < N)$, we have $\forall m_k \in \widehat{m}_i$, $m_k \in M$. In other words, any element m_k in any $\widehat{m}_i \in \widehat{M}$ is also in the set M, thus, $\bigcup_{\widehat{m}_i \in \widehat{M}} \widehat{m}_i \subseteq M$. On the other hand, $\forall m_i \in M$, $m_i \in \widehat{m}_{idx(i)}$, where $\widehat{m}_{idx(i)} \in \widehat{M}$. Thus, $\forall m_i \in M$, $m_i \in \bigcup_{\widehat{m}_i \in \widehat{M}} \widehat{m}_i$. Therefore, we have $M \subseteq \bigcup_{\widehat{m}_i \in \widehat{M}} \widehat{m}_i$. Because both $\bigcup_{\widehat{m}_i \in \widehat{M}} \widehat{m}_i \subseteq M$ and $M \subseteq \bigcup_{\widehat{m}_i \in \widehat{M}} \widehat{m}_i$ are true, we have $M = \bigcup_{\widehat{m}_i \in \widehat{M}} \widehat{m}_i$. Thus, Condition (3) is also satisfied.

Therefore, we can conclude that $\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\}$ is a partition of M. □

We have introduced CIIP, with examples, in this section. In the next section, we explain how to apply CIIP to CRPD estimation.

5.1.2 Applying CIIP to Estimate CRPD

The definition of CIIP provides us a formal representation to analyze inter-task cache evictions. By using CIIP, we can estimate CRPD. This section gives a way of using CIIP to estimate CRPD.

As explained in Section 5.1.1, we use a set of q memory block addresses, $M = \{m_0, m_1, \ldots, m_{q-1}\}$ to represent all the memory addresses that can possibly be accessed by a task.

Such a memory block set can be obtained by using simulation as used in SYMTA [68]. Then, we can derive the CIIP of M, which is represented with $\widehat{M}$ as given in Definition 15.

The memory block addresses in the same subset $\widehat{m}_i$ of a task's CIIP have the same index. Therefore, when these memory blocks are loaded into the cache, they might conflict with each other. Memory blocks in different subsets $\widehat{m}_i$, $\widehat{m_j}$, $i \neq j$, of the CIIP can never conflict in the cache.

Let us go back to Condition 1 in Section 4.1. Condition 1 states that the cache lines to be reloaded are used by both the preempted and preempting task. This requires that memory blocks accessed by the preempted task and memory blocks accessed by the preempting task are mapped to the same set in the cache.

Suppose we have two tasks T_a and T_b. All the memory blocks accessed by T_a and T_b are in the set $M_a = \{m_{a,0}, m_{a,1}, ..., m_{a,k_a}\}$ and $M_b = \{m_{b,0}, m_{b,1}, ..., m_{b,k_b}\}$ respectively, where $m_{a,i} (0 \leq i \leq k_a)$ are memory blocks accessed by T_a and $m_{b,j} (0 \leq j \leq k_b)$ are memory blocks accessed by T_b. T_b has a higher priority than T_a. An L-way set associative cache with a maximum index of $N-1$ is used in the system. In the case T_a is preempted by T_b, the cache lines to be reloaded when T_a resumes are used by both the preempting task (T_b) and the preempted task (T_a). Example 18 explains this point.

Example 18: Suppose we have a cache as defined in Example 15. Two tasks T_1 and T_2 run with this cache. T_1 has a lower priority than T_2. The memory block addresses accessed by T_1 and T_2 are contained in $M_1 = \{0x000, 0x110, 0x210\}$ and $M_2 = \{0x200, 0x310\}$, respectively. Both tasks use cache lines $cs(0)$ and $cs(1)$. Thus, these two cache lines can potentially need to be reloaded after T_2 finishes preempting T_1. □

We can look for the conflicting memory blocks accessed by the preempting task and the preempted task in order to estimate the number of cache lines to be reloaded. We use the CIIPs of M_a and M_b to solve this problem.

We use $\widehat{M}_a = \{\widehat{m}_{a,0}, \widehat{m}_{a,1}, ..., \widehat{m}_{a,N-1}\}$ to represent the CIIP of M_a and $\widehat{M_b} = \{\widehat{m}_{b,0}, \widehat{m}_{b,1},$

$\ldots, \widehat{m}_{b,N-1}\}$ to represent the CIIP of M_b. For $\widehat{m}_{a,k_1} \in \widehat{M}_a$ and $\widehat{m}_{b,k_2} \in \widehat{M}_b$, only when $k_1 = k_2$ can memory blocks in $\widehat{m}_{a,k_1}$ possibly conflict with memory blocks in $\widehat{m}_{b,k_2}$ in the cache. Also, when the memory blocks in $\widehat{m}_{a,k_1}$ and $\widehat{m}_{b,k_2}$ are loaded into the cache, the number of cache lines conflicting in one set of the cache can neither exceed the number of memory blocks that mapped to this set nor exceed the total number of cache lines in this set. In other words, the maximum number of cache lines conflicting in the set with index k_1 (or k_2 because $k_1 = k_2$) in the cache is $min(|\widehat{m}_{a,k_1}|, |\widehat{m}_{b,k_2}|, L)$, where L is the number of ways of the cache. Therefore, we can conclude that the following formula gives an upper bound for the number of cache lines that could be reloaded after Task T_a resumes following a preemption by Task T_b:

$$S(M_a, M_b) = \sum_{r=0}^{N-1} min\{|\widehat{m}_{a,r}|,\ |\widehat{m}_{b,r}|,\ L\} \tag{2}$$

where $\widehat{m}_{a,r} \in \widehat{M_a}$ and $\widehat{m}_{b,r} \in \widehat{M_b}$.

$S(M_a, M_b)$ denotes an upper bound on the number of cache lines that may conflict when the memory blocks in M_a and M_b are loaded into the cache. This number can be used to estimate the cache lines to be reloaded due to T_b preempting T_a.

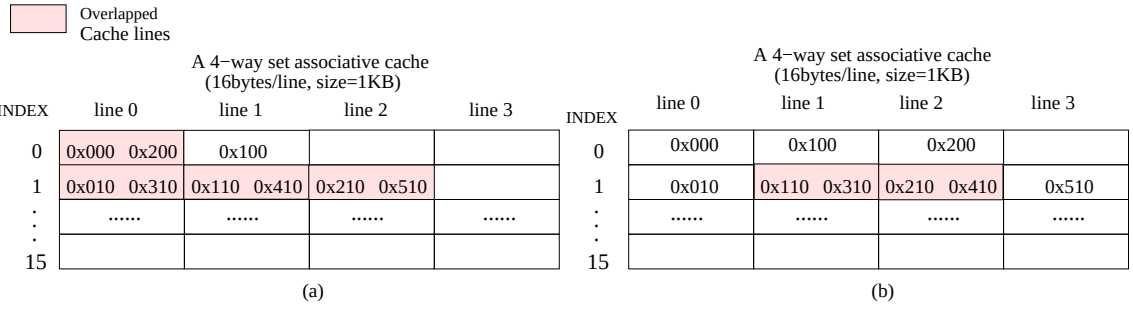


Figure 16: Conflicts of cache lines in a set associative cache

Example 19: Suppose we have a cache as defined in Example 7. Two tasks T_1 and T_2 run with this cache. The memory block addresses accessed by T_1 and T_2 are contained in $M_1 = \{0x000, 0x100, 0x010, 0x110, 0x210\}$ and $M_2 = \{0x200, 0x310, 0x410, 0x510\}$, respectively. The CIIPs of M_1 and M_2 are $\widehat{M}_1 = \{\{0x000, 0x100\}, \{0x010, 0x110,$

$0x210\}\}$ and $\widehat{M}_2 = \{\{0x200\}, \{0x310, 0x410, 0x510\}\}$, respectively. If we map the memory blocks in M_1 and M_2 to the cache as shown in Figure 16(a), we find that the maximum number of overlapped cache lines, which is 4, is the same as the result derived from Equation 2. Note that the memory blocks can be mapped to cache lines in other ways (e.g., $0x100$ can possibly be mapped to line 0 instead of line 1, but in this case $0x100$ would kick out $0x200$ or vice versa). In any case, the mapping given in Figure 16(a) gives a case in which the largest amount of cache line overlaps occurs. Let us consider another case. If we map the memory blocks in M_1 and M_2 to the cache as shown in the Figure 16(b), only two cache lines overlap. Obviously, the actual number of overlapped cache lines is related to the cache replacement policy and memory access pattern of the preempted task and the preempting task. However, Equation 2 gives an upper bound of the number of overlapped cache lines. □

In this section, we use CIIP to estimate the number of cache lines to be reloaded after the preempted task resumes from a preemption. Based on this estimate, we can obtain CRPD.

5.1.3 Summary

In this section, we first give the definition of CIIP and elaborate CIIP with some examples. We also explain the method of analyzing inter-task cache interference by using CIIP, which is a major contribution as we claimed in Section 1.3, **Contribution 1: A novel approach is proposed to analyze inter-task cache interference.**

However, only Condition 1 in Section 4.1 is used for cache interference analysis in this section. In the next section, we discuss how to apply Condition 2 in Section 4.1 to tighten the estimate of CRPD.

5.2 Intra-task Cache Eviction Analysis

In this section, we introduce briefly the intra-task cache eviction analysis approach proposed by Lee et al. [19, 20].

According to Condition 2 (Section 4.1), the memory blocks of the preempted task that can possibly cause cache reload cost must be present in the cache before the preemption and must be accessed by the preempted task again after the preemption. These memory blocks are called "useful memory blocks" by Lee et al., who give an approach to calculate the maximum set of useful memory blocks [19, 20, 21]. The useful memory blocks are only related to the memory access pattern and the program structure of the preempted task. Thus, we call this analysis intra-task cache analysis.

As we mentioned in Chapter 2, a task can be represented with a PCFG. Each node in a task PCFG is an SFP-PrS. A task can be preempted at any point in any SFP-PrS of the task. We call such a point an *execution point*. When a preemption happens, a task can be viewed as two parts, one part before the preemption and the other part after the preemption. The pre-preemption part (the part before the preemption) of the preempted task loaded memory blocks to the cache. Some of these memory blocks might be accessed again by the post-preemption part of the preempted task. These memory blocks are useful memory blocks. Only useful memory blocks of the preempted task can possibly cause cache reload after preemption(s).

For a formal description, we use the notation of *Reaching Memory Blocks (RMB)* and *Living Memory Blocks (LMB)* as defined in [19]. The set of *Reaching Memory Blocks* of a cache set $cs(i)$ at an execution point s of task T_a is denoted by $RMB_a(s,i)$. $RMB_a(s,i)$ contains all possible memory blocks that may reside in cache set $cs(i)$ when task T_a reaches execution point s. Suppose a cache set has L cache lines (i.e., a L-way set associative cache). If a memory block can reside in $cs(i)$, this memory block must have an index of i. Moreover, in order to be contained in $RMB_a(s,i)$, this memory block must be one of the last L distinct references to the cache set $cs(i)$ when the task runs along some execution

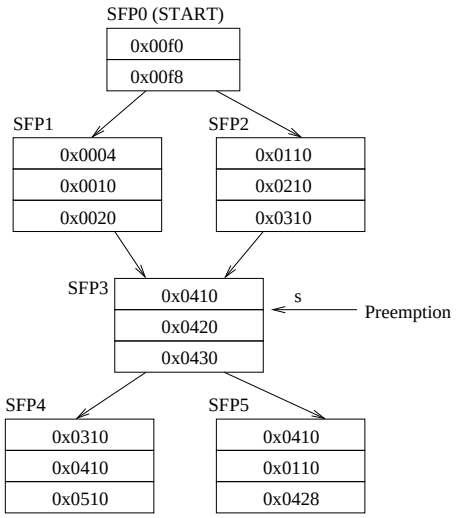


Figure 17: Example of RMB and LMB

path reaching execution point s. Otherwise, this memory would have been evicted from the cache by other memory blocks. Similarly, the set of *Living Memory Blocks* of cache set $cs(i)$ at execution point s, denoted by $LMB_a(s,i)$, contains all possible memory blocks that may be one of the first L distinct references to cache set $cs(i)$ *after* execution point s. By using *LMB* and *RMB*, we can calculate the *Useful Memory Blocks (UMB)* for task T_a. We use $UMB_a(s)$ to represent the UMBs at the execution point s of task T_a. We have $UMB_a(s) = \bigcup_{i=0}^{N-1}\{RMB_a(s,i) \bigcap LMB_a(s,i)\}$, where N is the number of cache lines in the cache. Example 20 uses the task shown in Figure 17 to explain *RMB* and *LMB*.

Example 20: Suppose we have a 2-way set associative cache with each line in the cache having 16 bytes. The maximum index of the cache is 15. Hence, the size of the cache is 512 bytes. We assume that a memory address has 16 bits. Thus, as in Example 7, bit 4 to bit 7 in a memory address determine the cache set to which the memory block in this address will be loaded. For example, the memory block with an address of $0x0010$ will be loaded into cache set 1.

In this example, a task with the PCFG shown in Figure 17 is executed. Each node in the PCFG is an SFP-PrS. The memory addresses accessed by each SFP-PrS

are given in Figure 17. Note that we do not distinguish instruction versus memory addresses. The cache is empty before the task runs. Suppose a preemption happens at the execution point s after the first memory block in SFP3 is accessed. Now, let us consider the RMB of cache set 1 at the execution point s where the preemption happens as indicated in the figure. The memory block of SFP3 that may reside in cache set 1 is only $0x0410$. Thus, $0x0410$ is included in $RMB(s,1)$. However, SFP3 has two predecessors, SFP1 and SFP2, which implies the task can take a path of {SFP1,SFP3} or {SFP2,SFP3} to reach the execution point s. If the task takes the path {SFP1,SFP3}, $0x0010$ may reside in cache set 1. If the task takes the path {SFP2,SFP3}, $0x0310$ may reside in cache set 1. Although $0x0210$ and $0x0110$ in SFP2 were also loaded to cache set 1, they are evicted before execution point s because in this example there are only two lines in each cache set and the LRU replacement algorithm is used. Now, considering all possible paths reaching execution point s, $RMB(s,1)$ should be $\{0x0010, 0x0310, 0x0410\}$. After the preemption, if the task takes the path {SFP3,SFP4}, $\{0x0310, 0x0410\}$ are the first two memory blocks that are loaded into cache set 1. If the task takes the path {SFP3,SFP5}, $\{0x0410, 0x0110\}$ are the first two memory blocks that are loaded into cache set 1. Thus, we have $LMB(s,1) = \{0x0310, 0x0410, 0x0110\}$. The intersection of $LMB(s,1)$ and $RMB(s,1)$ is $\{0x0310, 0x0410\}$. If there is no preemption, when the task accesses the memory block in $0x0410$ in SFP5, this memory block does not need to be loaded from the memory because it is already loaded into the cache in SFP3. However, due to the preemption, the preempting task may evict this cache line. So, the memory block $0x0410$ may potentially cause a cache line reload. Similar reasoning can be used to show that the memory block $0x0310$ can also possibly cause a cache line reload after preemption. Of course, whether a cache line reload is really needed is also dependent on the actual path the task takes and the exact cache lines used by the preempting task.

The intersection of *LMB* and *RMB* provides a superset of all memory blocks that may cause cache line reload(s). □

In [19], Lee et al. demonstrate that the intersection of *RMB* and *LMB* can give a superset of memory blocks in the preempted task that can potentially cause cache line reload overhead after preemption. The details of their algorithm can be found in [19, 20, 21]. Of course, whether those memory blocks will really cause cache line reloading still depends on the actual path the preempted task takes and the cache lines used by the preempting task. In the approach of Lee et al.[19, 20, 21], they conservatively assume that all the memory blocks in the intersection of *RMB* and *LMB* will be reloaded. Consider an extreme counter example for this assumption: if the cache lines used by the preempted task and the preempting task are completely disjoint based on cache index, the preempting task will not evict any cache lines used by the preempted task. In this case, there is no cache reload overhead imposed on the preempted task even there are preemptions, yet the approach of Lee et al. would indicate significant reload overhead (due to the fact that the approach of Lee et al. does not distinguish cache lines based on indices, i.e., the approach of Lee et al. does not utilize CIIP at all).

In the next section, we show how to improve the accuracy of the estimate of cache interference among tasks by combining inter- and intra-task cache eviction analysis.

5.3 Integrate Inter- and Intra-task Cache Eviction Analysis

In this section, we discuss our method for combining inter- and intra-task cache eviction analysis in order to tighten CRPD estimation.

In Section 5.1, the memory footprint used to calculate CIIP for a task covers all the memory blocks that can be possibly accessed by the task. However, for a task which is preempted, only the useful memory blocks of the preempted task can possibly cause cache reload. Lee et al. provide an approach to find the useful memory blocks of a task [19]. Now, we can calculate the intersection of useful memory blocks of the preempted task as derived

from the approach of Lee et al. and the memory blocks used by the preempting task in order to refine the CRPD estimate. By replacing the memory block set of the preempted task that is used in Equation 2 (Section 5.1.2) with the set of useful memory blocks, we can expect to reduce the estimate of the number of cache lines to be reloaded after a preemption.

Suppose we have two tasks, T_a and T_b. Suppose further that T_b has a higher priority than T_a; thus, T_b can preempt T_a. In the case that T_b preempts T_a, we want to know a tight upper bound on the number of cache lines that need to be reloaded by T_a after T_a resumes from the preemption. To help calculate such an upper bound, we define the Maximum Useful Memory Blocks Set on the basis of Useful Memory Blocks (UMB) as introduced in Section 5.2.

Definition 17. The Maximum Useful Memory Blocks Set (MUMBS): The maximum useful memory blocks set is the union of all useful memory blocks sets defined by all possible execution points of a task. We represent the maximum useful memory blocks set of task T_a with $\tilde{M}_a$. $\widehat{\tilde{M}}_a$ is the CIIP of $\tilde{M}_a$. We have $\tilde{M}_a = \bigcup_s UMB_a(s)$. $\tilde{M}_a$ is a subset of M_a. □

We use the approach of Lee et al. to derive the useful memory blocks at each execution point of the preempted task. Then, we calculate the union of useful memory blocks over all the execution points to obtain the MUMBS of the preempted task. Only the memory blocks in this MUMBS set can possibly be reloaded by the preempted task. Example 21 explains the method of calculating MUMBS.

Example 21: Suppose we have a task with the PCFG shown in Figure 18. The memory blocks accessed by the task is also shown in Figure 18. Let us consider the execution point s. Memory block $0x0020$ is accessed by the task before the execution point s as well as after the execution point s. By using the useful memory block analysis approach in [19], we can find that $0x0020$ is a useful memory block. The useful memory block set at the execution point s is $\{0x0020\}$. We perform the same analysis overall the execution points in the task and calculate the union of all useful memory block to obtain MUMBS. In this example, the MUMBS

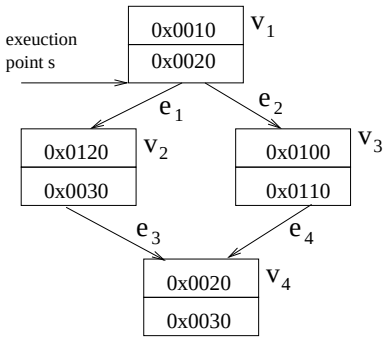


Figure 18: An example of MUMBS

is $\{0x0020, 0x0030\}$. □

The simulation method in SYMTA [68] is used to obtain all the memory blocks that can possibly accessed by the preempting task T_b. All these memory blocks are contained in a set M_b. $\widehat{M}_b$ is the CIIP of M_b. Only the memory blocks in M_b can possibly evict the cache lines used by the preempted task.

Then, we apply Equation 2 to calculate the intersection of memory block set $\tilde{M}_a$ and M_b, which is shown in Equation 3. This result gives an upper bound on the number of cache lines that can possibly need to be reloaded after T_b preempts T_a.

$$S(\tilde{M}_a, M_b) = \sum_{r=0}^{N-1} min\{|\widehat{\tilde{m}}_{a,r}|,\ |\widehat{m}_{b,r}|,\ L\} \tag{3}$$

In Equation 3, N is the number of cache sets, r is the index of a CIIP element, $\widehat{\tilde{m}}_{a,r} \in \widehat{\tilde{M}_a}$ and $\widehat{m}_{b,r} \in \widehat{M_b}$.

Example 22: Suppose we have two tasks T_1 and T_2. T_1 is shown in Example 21. Memory blocks accessed by T_2 are in the set $M_2 = \{0x0110, 0x0120, 0x0020\}$. We have a two way set associative cache with 16 cache sets; thus, we have $L = 2$. Each cache line has 16 bytes. The MUMBS of M_1, $\tilde{M}_1$, is $\{0x0020, 0x0030\}$ as shown in Example 21. By using the definition of CIIP, we can obtain the CIIP of $\tilde{M}_1$, $\widehat{\tilde{M}}_1 = \{\widehat{\tilde{m}}_{12}, \widehat{\tilde{m}}_{13}\}$, where $\widehat{\tilde{m}}_{12} = \{0x0020\}$ and $\widehat{\tilde{m}}_{13} = \{0x0030\}$. We can also calculate the CIIP of M_2, which is $\widehat{M}_2 = \{\widehat{m}_{21}, \widehat{m}_{22}\}$, where $\widehat{m}_{21} = \{0x0110\}$ and $\widehat{m}_{22} = \{0x0120, 0x0020\}$. Now, we can use Equation 3 to estimate cache conflicts.

Since $\tilde{M}_1$ does not have memory blocks mapped in the cache line with an index of 1 and M_2 does not have memory blocks mapped in the cache line with an index of 3, cache conflicts can only occur in the cache line with an index of 2. $min\{|\widehat{\tilde{m}}_{12}|, |\widehat{m}_{22}|, L\} = min\{1,2,2\} = 1$. Thus, $S(\tilde{M}_1, M_2) = 1$. $\square$

Notice that in Equation 3, $\tilde{M}_a$ is a subset of M_a. Thus, the result derived from Equation 3 is definitely smaller than or equal to the result derived from Equation 2. In other words, the estimate of the number of cache lines to be reloaded can only be tightened by integrating intra- and inter-task cache eviction analysis. Therefore, the CRPD estimate is more precise. In this section, we present the second major contribution claimed in Section 1.3, **Contribution 2: Inter-task cache eviction analysis is integrated with intra-task cache eviction analysis.**

CRPD overestimation can be further reduced by applying a path analysis technique as explained in the next section.

5.4 Path Analysis

In this section, we show that CRPD can be tightened by exploiting path analysis of the preempting task. Note that for the preempted task, as explained in Section 5.2, path analysis is already performed in order to calculate the useful memory blocks[19, 20, 21]. Thus, we do not consider path analysis for the preempted task in this section.

Suppose we have two tasks, T_a and T_b, and T_a has a lower priority than T_b. Thus, T_a can be preempted by T_b. Here, we do not consider nested preemptions. Nest preemptions are discussed in Section 6.2.2. We assume M_b is the memory block set which contains all the memory blocks that can possibly be accessed by the preempting task T_b. If we do not use any path analysis methods, the CRPD caused by T_b preempting T_a can be estimated with Equation 3. However, since the preempting task might have multiple feasible paths only one of which is executed, some memory blocks included in M_b may in fact not be accessed; thus, there is no need to reload the cache lines mapped from these memory blocks

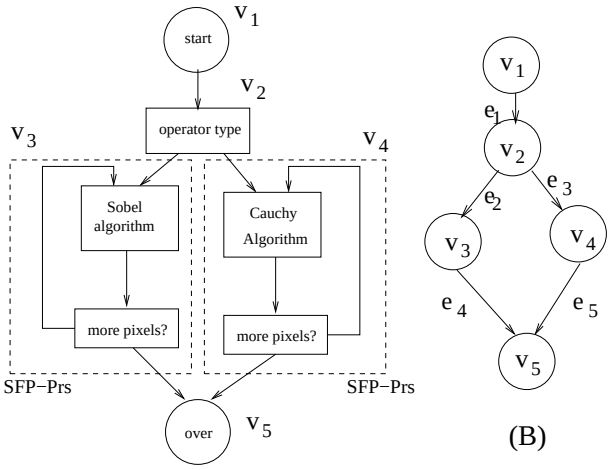


Figure 19: PCFG of ED

not accessed. Example 23 gives such a case.

Example 23: Figure 19 shows the PCFG of ED (ED was explained in Example 14). Let us consider the three nodes v_2, v_3 and v_4 shown in Figure 19(A) – please note that nodes v_3 and v_4 are defined by the dashed lines shown in Figure 19(A). When the image size is fixed (i.e., the number of pixels to be processed is fixed), the loop bounds in the dashed-line rectangles of v_3 and v_4 are fixed. Other than the loop bounds in v_3 and v_4, there are no other branches depending on input data in v_3 and v_4. Thus, nodes v_3 and v_4 can be viewed as SFP-PrS. The PCFG of ED can be simplified as the graph shown in Figure 19 (B). Each node in this graph represents an SFP-PrS in the ED program. According to an input parameter selected by the user, the program can only take either the path $(v_1, e_1, v_2, e_2, v_3, e_4, v_5)$ or the path $(v_1, e_1, v_2, e_3, v_4, e_5, v_5)$; thus, only one of v_3 or v_4 can be accessed in one run. In this case, the potentially evicted cache lines used by v_3 and the potentially evicted cache lines used by v_4 never both need to be reloaded at the same time after one solitary execution of ED. □

The issue presented in Example 23 can be described more generally. Suppose we have

two tasks, T_a and T_b, in a system with an L-way set associative cache. The largest index of any cache line in the cache is $N-1$. T_b has a higher priority than T_a. Thus, T_b can preempt T_a. We use M_a to represent the set of all memory block addresses that can be possibly accessed by T_a. $\tilde{M}_a$ is the MUMBS of the preempted task (MUMBS was defined in Section 5.2). We define the PCFG of T_b to be $G_b = (V_b, E_b)$, where $V_b = \{v_{b,1}, v_{b,2}, ..., v_{b,n}\}$ and $E_b = \{e_{b,1}, e_{b,2}, ..., e_{b,m}\}$. A path in G_b can be represented with $Pa_b^k = \{v_{b,i_1}, e_{b,i_1}, v_{b,i_2}, e_{b,i_2}, ..., v_{b,i_p}\}$. We use M_b^k to denote the set of memory block addresses accessed by the task T_b when T_b runs along the path Pa_b^k. The CIIP of M_b^k is $\widehat{M}_b^k = \{\widehat{m}_{b,0}^k, \widehat{m}_{b,1}^k, ..., \widehat{m}_{b,N-1}^k\}$. When Pa_b^k is determined, $M_{b,k}$ can be derived from simulation with the method used in SYMTA [68] as outlined in Section 4.1.

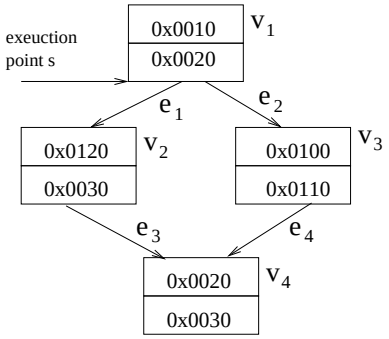


Figure 20: PCFG of T_b in Example 24

Example 24: Suppose we have a task, T_b, with a corresponding PCFG shown in Figure 18 (repeated here as Figure 20 for convenience). This task has two feasible paths, $Pa_b^1 = \{v_1, e_1, v_2, e_3, v_4\}$ and $Pa_b^2 = \{v_1, e_2, v_3, e_4, v_4\}$. When T_b runs along Pa_b^1, the memory blocks accessed by T_b are $M_b^1 = \{0x0010, 0x0020, 0x0120, 0x0030\}$. When T_b runs along Pa_b^2, the memory blocks accessed by T_b are $M_b^2 = \{0x0010, 0x0020, 0x0100, 0x0110, 0x0030\}$. We assume a cache as described in Example 22. Thus, we have the CIIP of M_b^1, $\widehat{M}_b^1 = \{\widehat{m}_{b,1}^1, \widehat{m}_{b,2}^1, \widehat{m}_{b,3}^1\}$, where $\widehat{m}_{b,1}^1 = \{0x0010\}$, $\widehat{m}_{b,2}^1 = \{0x0020, 0x0120\}$ and $\widehat{m}_{b,3}^1 = \{0x0030\}$. The CIIP of M_b^2 is $\widehat{M}_b^2 = \{\widehat{m}_{b,0}^2, \widehat{m}_{b,1}^2, \widehat{m}_{b,2}^2, \widehat{m}_{b,3}^2\}$, where $\widehat{m}_{b,0}^2 = \{0x0100\}$, $\widehat{m}_{b,1}^2 = \{0x0010, 0x0110\}$, $\widehat{m}_{b,2}^2 = \{0x0020\}$ and $\widehat{m}_{b,3}^2 = \{0x0030\}$. □

Note that Pa_b^k is generic notation for any path in T_b. Among all the paths in T_b, there exists a particular path in T_b which, when T_b takes this path, the memory blocks loaded to the cache have the largest overlap with the cache lines used by memory blocks in the MUMBS of the preempted task T_a. In other words, when T_b takes this path, the number of cache lines evicted by T_b and also used by T_a is the largest. This problem can be transformed to a problem of finding the "worst MUMBS path" in a graph. We describe this transformation in the following paragraphs. Note that there may be more than one "worst MUMBS path" because different paths may cause the same largest estimate of the number of cache conflicts. However, we only need to find one "worst MUMBS path" because we are only interested in the upper bound on the estimate of the number of cache conflicts, not the exact path itself.

We define a cost function for a path Pa_b^k in the preempting task T_b.

$$C(Pa_b^k) = S(\tilde{M}_a, M_b^k) = \sum_{r=0}^{N-1} min\{|\hat{\tilde{m}}_{a,r}|,\ |\hat{m}_{b,r}^k|,\ L\} \tag{4}$$

The cost of a path Pa_b^k in the preempting task T_b is defined as the maximum number of cache lines that can possibly overlap the cache lines mapped by useful memory blocks of the preempted task T_a, when the preempting task T_b runs along the path Pa_b^k.

Definition 18. The Worst MUMBS Path (WMP): The worst MUMBS path is an execution path of a task such that when the task runs along this path, the objective function in Equation 4 has the largest value. We use Pa_b^{WMP} to represent the WMP of a task T_b. □

Example 25: Suppose we have two tasks T_a and T_b. T_a has a lower priority than T_b. The MUMBS of T_a is $\tilde{M}_a = \{0x1100, 0x1110, 0x1120, 0x1130\}$. We assume a cache as described in Example 22. The CIIP of $\tilde{M}_a$ is $\hat{\tilde{M}}_a = \{\hat{\tilde{m}}_{a,0}, \hat{\tilde{m}}_{a,1}, \hat{\tilde{m}}_{a,2}, \hat{\tilde{m}}_{a,3}\}$, where $\hat{\tilde{m}}_{a,0} = \{0x1100\}$, $\hat{\tilde{m}}_{a,1} = \{0x1110\}$, $\hat{\tilde{m}}_{a,2} = \{0x1120\}$ and $\hat{\tilde{m}}_{a,3} = \{0x1130\}$. T_b is described in Example 24. The PCFG is shown in Figure 21(A). Figure 21(B) shows possible cache conflicts between T_a and T_b when T_b run along the path Pa_b^1. In this case, $C(Pa_b^1) = S(\tilde{M}_a, M_b^1) = min\{|\hat{\tilde{m}}_{a,1}|, |\hat{m}_{b,1}^1|, 2\} + min\{|\hat{\tilde{m}}_{a,2}|, |\hat{m}_{b,2}^1|, 2\} +$

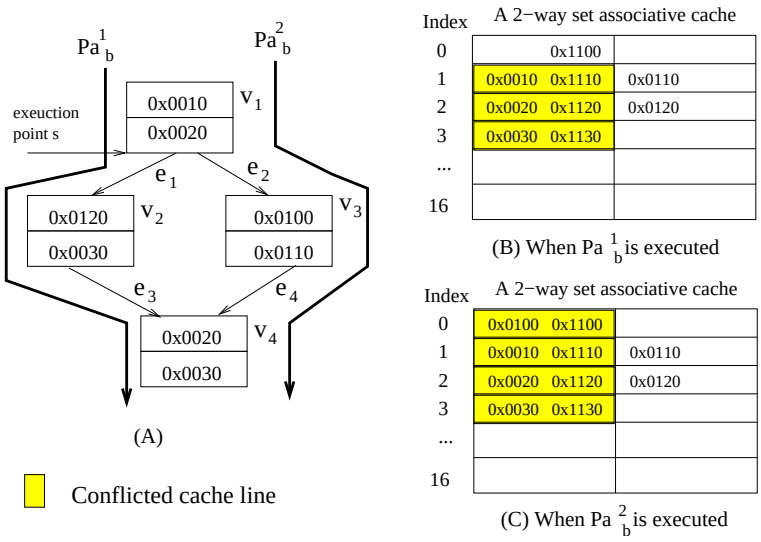


Figure 21: A WMP Example

$min\{|\widehat{\widehat{m}}_{a,3}|, |\widehat{m}^1_{b,3}|, 2\} = 1+1+1 = 3$. Figure 21(C) shows possible cache conflicts between T_a and T_b when T_b run along the path Pa^2_b. Similarly, we have $C(Pa^2_b) = 4$. Thus, $C(Pa^{WMP}_b) = C(Pa^2_b) = 4$. □

By using the cost function of Equation 4, we search all the paths of the preempting task to find the worst MUMBS path in the PCFG of T_b. Suppose the worst MUMBS path in task T_b is represented with Pa^{WMP}_b; then, the cache lines to be reloaded in the worst case is bounded by the cost of Pa^{WMP}_b as defined by Equation 4. Potentially we need to search all paths to find Pa^{WMP}_b in the preempting task. We use an example to explain this path searching procedure.

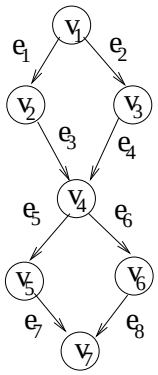


Figure 22: An example of path analysis

Example 26: Suppose we have a task, the PCFG of which is shown in Figure 22. There are two nodes, v_1 and v_4, that have more than one outgoing edge. We have to consider four paths $\{v_1, e_1, v_2, e_3, v_4, e_5, v_5, e_7, v_7\}$,$\{v_1, e_1, v_2, e_3, v_4, e_6, v_6, e_8, v_7\}$, $\{v_1, e_2, v_3, e_4, v_4, e_5, v_5, e_7, v_7\}$,$\{v_1, e_2, v_3, e_4, v_4, e_6, v_6, e_8, v_7\}$ in order to find the WMP for this task. Thus, in the worst case, the number of paths we search is exponential in the number of nodes that have more than one outgoing edge. □

Path analysis performed on the preempting task may be exponential in the number of nodes that have more than one outgoing edge in the PCFG of the preempting task. However, in practice, many embedded programs have PCFGs with a reasonably small number of paths. Thus, our approach can still apply to many such systems. In our approach, users have an option to select path analysis or not. Usually, path analysis can be used to refine CRPD estimates when the control structure in the preempting task is not complex or computation time is not a major concern in the design phase. Otherwise, we can simply not use path analysis in order to avoid time consuming computation.

Compared to Equation 3, the estimate given by Equation 4, if there is a change, can only be further reduced; this is true because only a subset of the memory blocks in M_b are considered in the calculation of Equation 4.

In this section, we introduce a method to use path analysis to improve cache interference analysis, which is the third contribution claimed in Section 1.3, **Contribution 3: Path analysis is used to improve cache interference analysis**.

In Sections 5.1, 5.2, 5.3 and 5.4, we discuss how to estimate the number of cache lines to be reloaded after preemptions. Equation 4 gives a method to combine intra-task eviction, inter-task cache eviction analysis and path analysis in order to obtain a tight estimate of the number of cache conflicts between the preempted and preempting task. In the next section, we discuss how to use the estimate of cache conflicts to analyze CRPD.

5.5 CRPD Estimation

In this section, we give our improved method to estimate CRPD. CRPD relates to two factors: (i) the number of cache lines to be reloaded and (ii) the cache miss penalty. We use $CRPD(T_a, T_b)$ to represent the CRPD imposed on task T_a when T_a is preempted by task T_b. Suppose the penalty for a cache miss is a constant, C_{miss} (see Section 4.1 for a discussion of our system-level assumptions including memory hierarchy). Then, $CRPD(T_a, T_b)$ can be calculated with the following equation:

$$CRPD(T_a, T_b) = C(Pa_b^{WMP}) \times C_{miss} = S(\tilde{M}_a, M_b^{WMP}) \times C_{miss} \quad (5)$$

Example 27: Let us consider two tasks, T_a and T_b, as described in Example 25. We assume a fixed cache miss penalty, $C_{miss} = 10$ clock cycles. In this case, we have $CRPD(T_a, T_b) = C(Pa_b^{WMP}) \times C_{miss} = 4 \times 10 = 40$ clock cycles. □

Equation 5 gives an estimate of the CRPD induced by T_b preempting T_a. By incorporating CRPD, we can derive a new approach to estimate the WCRT of each task in a preemptive multi-tasking system.

Note that the cache miss penalty in practice may not always be constant. For example, in a wrap-around-fill cache [11], a cache line is not filled completely at one time. Instead, the CPU starts to run as soon as the requested memory contents are fetched from the memory to the cache. The rest of the cache line is filled while the CPU continues execution. In this type of cache, the cache miss penalty varies. An cache miss delay analysis approach is presented in [66] for the wrap-around-fill cache. Our approach can be easily extended to handle the wrap-around-fill cache by replacing the constant C_{miss} with a cache miss penalty function $C_{miss}(m_i)$, where m_i is in the intersection set of the memory blocks used by the preempting task and the preempted task, as found by the approach presented in [66].

5.6 Summary

In this chapter, we introduce our new approach to analyze and estimate an upper bound for CRPD. A new concept, CIIP, is proposed to estimate the inter-task cache interference based on the memory footprints of tasks. We also apply path analysis to refine the CRPD result derived with CIIP. In the next chapter, we incorporate CRPD into WCRT analysis. A novel WCRT analysis formula is proposed. Nested preemptions are also considered.

CHAPTER VI

WCRT ANALYSIS

In this chapter, we first introduce a simple WCRT analysis approach provided by Joseph and Pandya [14]. Cache behavior is not considered in this prior approach. Next, we incorporate CRPD into WCRT analysis. Finally, system schedulability is evaluated based on this new WCRT analysis approach.

6.1 Basic WCRT Analysis Method

This section introduces a basic WCRT analysis method which does not consider cache interference and context switch costs.

A simple WCRT analysis approach is proposed in [14] to find the WCRT of a task T_i. The minimum arrival interval and the Worst Case Execution Time are assumed to be known in advance. Also, a *critical instant* where all tasks are released together gives the worst case scenario for tasks. The following iterative equation gives the WCRT of Task T_i assuming that all tasks are released as soon as they arrive.

$$R_i^k = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times C_j \tag{6}$$

In Equation 6, $hp(i)$ is the set of tasks whose priorities are higher than T_i. R_i^k is the WCRT of T_i in the k_{th} iteration. Recall that C_j is the WCET of T_j and P_j is the period of Task T_j as defined in Section 2.1. The deadline of T_i is at the end of period P_i of T_i. We set the initial WCRT of T_i, R_i^0, to be the same as the WCET of T_i. Because we assume that all tasks are sorted in the descending order of their priorities in this paper, we have $hp(i) = \{k | 0 \leq k < i\}$. In Equation 6, the term $\sum_{j \in hp(i)} \lceil \frac{R_i}{P_j} \rceil \times C_j$ reflects the worst case interference of preempting tasks during the time T_i is either executing or is interrupted

(preempted). This equation can be calculated iteratively. The iteration terminates when R_i converges or R_i is greater than the deadline of T_i. If R_i exceeds the deadline associated with T_i, then we are not able to schedule task T_i successfully using this approach. A theorem is given to evaluate if Equation 6 converges.

Theorem. If the CPU utilization (see Definition 7) of the i highest priority tasks is less than 1, the sequence of R_i calculations (Equation 6) converges in a finite number of steps. □

The proof of this theorem is given in [14] and [59].

Task T_i meets its deadline if and only if R_i converges to a value less than P_i. If R_i diverges or converges to a value greater than the deadline of T_i, we simply conclude that a feasible schedule is not found for this task. This WCRT analysis approach does not consider cache effects but does provide a basis for our work presented in this thesis.

The WCRT analysis approach given in Equation 6 does not consider the cache interference and context switch cost. These additional costs related to preemptions extend task WCRT and can potentially make tasks miss their deadlines. Thus, for the sake of safety in a real-time system, we need to include these costs in WCRT analysis. The next section explains how to accomplish this goal of adding cache interference and context switch costs to WCRT analysis.

6.2 Enhancement of WCRT Analysis

In this section, we enhance the basic WCRT analysis approach presented in Section 6.1. Our enhancement consists in incorporating the context switch cost and the CRPD.

6.2.1 Context Switch

In this section, we discuss how context switch cost caused by preemptions may affect WCRT.

In multi-tasking systems, when a task switch takes place, the context of the old task is

stored in the kernel stack and the context of the new task is loaded from the kernel stack. A context switch can be as simple as changing the value of the program counter and stack pointer or may be more complicated, e.g., involving resetting the MMU to make a different set of memory pages available. Some processors provide special context switch instructions in order to accelerate the context switch operation. Preemptions invoke context switches as shown in Example 28.

Example 28: Figure 23 shows two periodic tasks in a preemptive multi-tasking system. Task A has a higher priority than Task B. Thus, Task B can be preempted by Task A. We use $T_{A,1}$ and $T_{B,1}$ to represent the first run of Task A and Task B respectively. In each preemption, two context switches happen. The first context switch occurring at time instant t_1 loads the preempting task. The second context switch occurring at time instant t_2 resumes the preempted task. The response time of the preempted task is extended due to context switch cost as shown with black boxes in Figure 23. We assume a constant upper bound for a context switch cost, C_{miss}. In this example, we use $2C_{miss}$ to estimate the context switch cost in one preemption. □

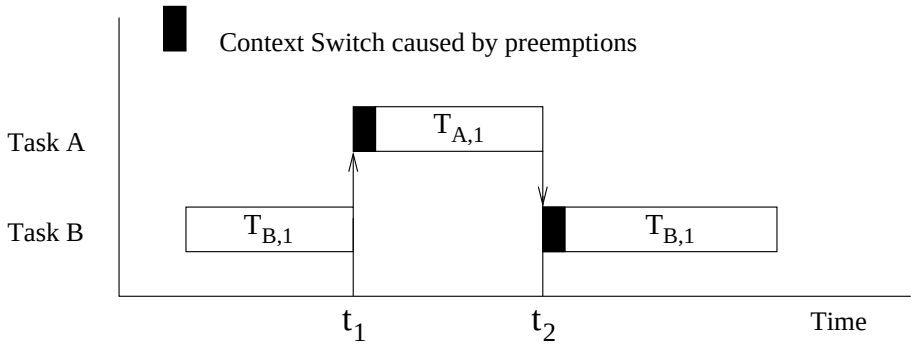


Figure 23: A context switch example

As compared to the execution time of a task, the context switch cost is typically small. We assume that the context switch function cannot be preempted. Thus, the context switch cost is not affected by inter-task cache eviction. Therefore, it is reasonable to assume the context switch cost has a constant upper bound. In this thesis, we assume the context

switch has a constant upper bound C_{cs}, which is equal to the WCET of a context switch. As shown in Example 28, each preemption adds an additional context switch cost of $2C_{cs}$ to the WCRT of the preempted task. Example 29 gives an example of context switch cost, which is derived from a realistic commercial simulation platform which we use for the experiments in this thesis.

Example 29: An ARM9TDMI processor with two levels of memory, a 32KB 4-way set associative L1 cache and 256MB of SRAM main memory, is used in our experiment. The cache miss penalty is 20 cycles. The Atalanta RTOS developed at Georgia Tech [53] is used for task management. We use SYMTA to obtain the WCET of a context switch, which implies that the instructions of the context switch function and the memory blocks where contexts of the preempted and the preempting tasks are saved are not in the L1 cache when the context switch function is called. In this case, the WCET of a single context switch estimated with SYMTA is 1049 cycles. □

6.2.2 CRPD and Nested Preemptions

CRPD as discussed in Chapter 5 only considers a preemption "base case" in which only one level of preemption occurs. However, in a realistic multi-tasking systems, preemptions can be nested. In other words, during a preemption, the preempting task can be preempted further by another task. This section gives a way to estimate CRPD in nested preemptions.

First, we use an example, Example 30, to show a case of nested preemptions.

Table 1: Tasks in Example 30

Task	WCET(us)	Period(us)	Preemptions	Cache reload cost (us)
T_0	5	20	T_0 preempting T_1	5
T_1	11	30	T_0 preempting T_2	2
T_2	12	100	T_1 preempting T_2	2

Example 30: Consider the tasks in Example 1. We assume that WCET, period and cache reload cost for each task are listed in Table 1. Task T_2 has the has the

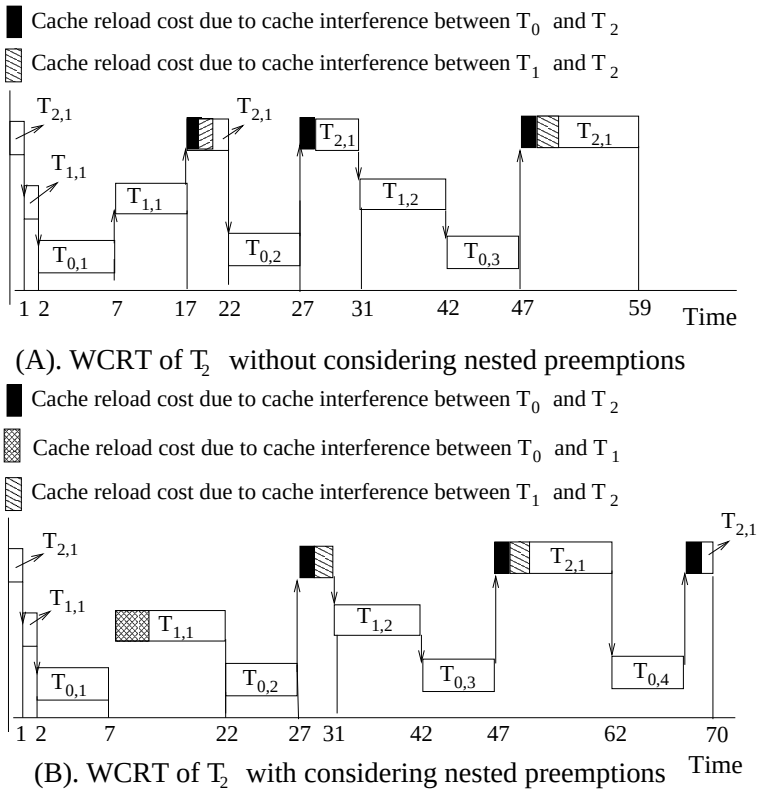


Figure 24: An example of nested preemptions

lowest priority and Task T_0 has the highest priority. Here we ignore the context switch cost. At time instant 1 (measured in ms), T_2 is preempted by T_1 directly. Then, at time instant 2, T_1 is preempted by T_0. The second preemption is nested in the first preemption. Thus, T_2 is also preempted by T_0, albeit indirectly, at time instant 2. We call this type of preemption an *indirect preemption*.

Note that at time instant 27 in Figure 24(B), T_2 is preempted again as soon as cache lines are reloaded. This only happens in the worst case where all cache lines to be reloaded have to be reloaded at the beginning of T_2 resuming from a preemption before T_2 starts to run. We use this case which may rarely occur in order to show the impact of cache reload cost in the worst case.

Note that another indirect preemption similar to what happened at time instant 2 occurs at time instant 42. In this case, the CRPD caused by T_0 indirectly preempting T_2 consists of two parts, cache reload cost due to cache interference between

T_2 and T_0 and cache reload cost due to cache interference between T_1 and T_0 as shown in Figure 24(B). However, by using Equations 4 and 5, only cache reload cost due to cache interference between T_2 and T_0 is included in CRPD caused by T_0 indirectly preempting T_2, which is shown in Figure 24(A). Comparing Figure 24(A) with Figure 24(B), we find that cache reload cost due to cache interference between T_1 and T_0 can possibly extend the response time of T_2. Notice that when the cache reload cost due to cache interference between T_0 and T_1 is considered, the WCRT of T_2 is 70 (instead of 59). Thus, we need to include this factor in our WCRT analysis; in this specific case, while T_0 can arrive at most three times in 59 time units, in fact T_0 can arrive up to four times in 70 time units – as shown in Figure 24(B). □

Example 30 shows the effect of nested preemptions on WCRT. Note that in this example, we explain the difference between an *indirect preemption* and a *direct preemption*. When we estimate the WCRT of a task T_a, we need to consider all possible preemptions caused by each task, T_b, $0 \leq b < a$, which has a higher priority than T_a. T_b can preempt T_a directly, which brings a cache reload cost of $CRPD(T_a, T_b)$ to the WCRT of T_a. T_a can also potentially be preempted by T_b indirectly if there exists a task T_l with a priority lower than T_b, but higher than T_a. In this case, when an instance of T_b arrives while T_a is preempted by T_l, T_l is further preempted by T_b. This indirect preemption introduces a cache reload cost of $CRPD(T_l, T_b)$ to the WCRT of T_a. In the worst case, T_{a-1} preempts T_a first, then T_{a-1} is preempted by T_{a-2}, ..., until finally T_{b+1} is then preempted by T_b. Thus, there are $a-b$ nested preemptions in the worst case.

In Equation 4 (Section 5.4), repeated below as Equation 7 for convenience, the number of cache conflicts between T_a and T_b results from a calculation using $\tilde{M}_a$, the MUMBS of T_a, and M_b^k, the memory blocks that are accessed by T_b when T_b runs along a particular path.

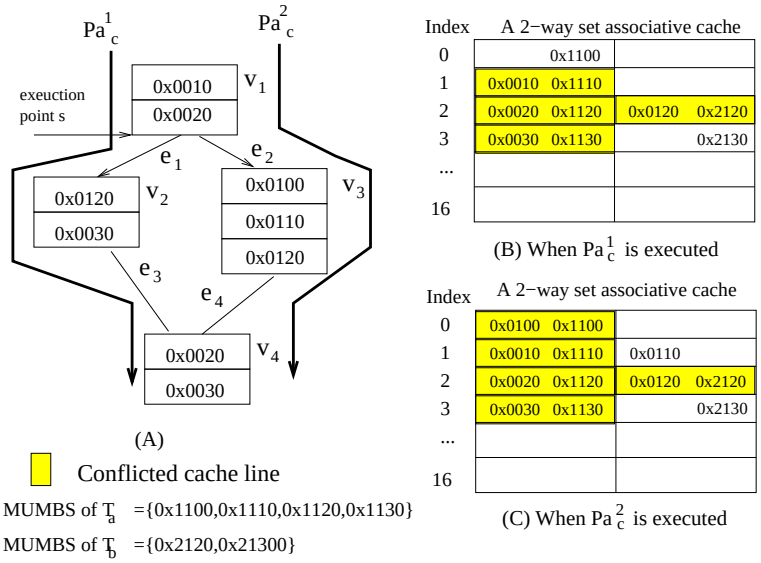


Figure 25: Cache conflicts in Example 31

$$C(Pa_b^k) = S(\tilde{M}_a, M_b^k) = \sum_{r=0}^{N-1} min\{|\hat{\tilde{m}}_{a,r}|,\ |\hat{m}_{b,r}^k|,\ L\} \tag{7}$$

However, when nested preemptions exist, T_b may evict cache lines used by useful memory blocks of all tasks that have higher priorities than T_a but lower priorities than T_b. In order to include nested preemptions, Equation 4 (Equation 7) is extended as follows:

$$C(Pa_b^k) = S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b^k) = \sum_{r=0}^{N-1} min\{|\bigcup_{l=b+1}^{a} \hat{\tilde{m}}_{l,r}|,\ |\hat{m}_{b,r}^k|,\ L\} \tag{8}$$

Example 31: Suppose we have three tasks, T_a, T_b and T_c. T_a has the lowest priority and T_c has the highest priority. The PCFG of T_c is shown in Figure 25(A). The MUMBS of T_a is $\tilde{M}_a = \{0x1100, 0x1110, 0x1120, 0x1130\}$ as in Example 25. The MUMBS of T_b is $\tilde{M}_b = \{0x2120, 0x2130\}$. We assume a unified cache as described in Example 22. When estimating CRPD caused by T_c preempting T_a (directly or indirectly), if we do not consider cache reload cost that can possibly be caused by cache interference between T_a and T_b, we have $C(Pa_c^{WMP}) = 4$ as explained in Example 25. (Note that T_c in this example is the same as T_b in Example 25; thus, we can use the same calculation.) However, we have to consider the possible cache

reload cost caused by cache interference between T_b and T_c in the case of T_a indirectly preempting T_c. In this example, we have $\tilde{M}_a \bigcup \tilde{M}_b = \{0x1100, 0x1110, 0x1120, 0x1130, 0x2120, 0x2130\}$. We use $\widehat{\tilde{M}}_{ab}$ to represent the CIIP of the union set $\tilde{M}_a \bigcup \tilde{M}_b$. We have $\widehat{\tilde{M}}_{ab} = \{\widehat{\tilde{m}}_0, \widehat{\tilde{m}}_1, \widehat{\tilde{m}}_2, \widehat{\tilde{m}}_3\}$, where $\widehat{\tilde{m}}_0 = \{0x1100\}$, $\widehat{\tilde{m}}_1 = \{0x1110\}$, $\widehat{\tilde{m}}_2 = \{0x1120, 2120\}$ and $\widehat{\tilde{m}}_3 = \{0x1130, 2130\}$. Now, we can use Equation 8 to estimate the number of cache conflicts when T_c runs along different paths. Figure 25(B) shows possible cache conflicts when T_c runs along the path Pa_c^1. Figure 25(C) shows possible cache conflicts when T_c runs along the path Pa_c^2. Similar to Example 25, we can find that when T_c runs along the path $Pa_c^2 = \{v_1, e_2, v_3, e_4, v_4\}$, we find the largest number of possible cache conflicts. Thus, Pa_c^2 is the WMP, Pa_c^{WMP}. When T_c runs along Pa_c^{WMP}, the memory blocks accessed by T_c are $M_c^{WMP} = \{0x0010, 0x0020, 0x0100, 0x0110, 0x0120, 0x0030\}$. We have the CIIP of M_c^{WMP}, $\widehat{M}_c^{WMP} = \{\widehat{m}_{c,0}^{WMP}, \widehat{m}_{c,1}^{WMP}, \widehat{m}_{c,2}^{WMP}, \widehat{m}_{c,3}^{WMP}\}$, where $\widehat{m}_{c,0}^{WMP} = \{0x0100\}$, $\widehat{m}_{c,1}^{WMP} = \{0x0110, 0x0010\}$, $\widehat{m}_{c,2}^{WMP} = \{0x0020, 0x0120\}$ and $\widehat{m}_{c,3}^{WMP} = \{0x0030\}$. By applying CIIP calculation as shown in Example 25, we have

$C(Pa_c^{WMP}) = S(\widehat{\tilde{M}}_{ab}, M_c^{WMP}) = min(|\widehat{m}_{c,0}^{WMP}|, |\widehat{\tilde{m}}_0|, 2) + min(|\widehat{m}_{c,1}^{WMP}|, |\widehat{\tilde{m}}_1|, 2)$
$+ min(|\widehat{m}_{c,2}^{WMP}|, |\widehat{\tilde{m}}_2|, 2) + min(|\widehat{m}_{c,3}^{WMP}|, |\widehat{\tilde{m}}_3|, 2) = 1 + 1 + 2 + 1 = 5.$ □

In Equation 9, we show the combination of Equation 8 with Equation 5 to estimate the cache reload cost caused by T_b preempting T_a, where T_b has a higher priority than T_a and there may be additional tasks with priorities higher than T_a but lower than T_b.

$$CRPD(T_a, T_b) = C_{miss} \times S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b^{WMP}) = C_{miss} \times \sum_{r=0}^{N-1} min\{|\bigcup_{l=b+1}^{a} \widehat{\tilde{m}}_{l,r}|, |\widehat{m}_{b,r}^{WMP}|, L\} \quad (9)$$

In Equation 9, $\tilde{M}_l$ is the MUMBS of task T_l and M_b^{WMP} is the set of memory blocks accessed by task T_b when task T_b runs along the worst MUMBS path.

Example 32: We use the same tasks, T_a, T_b and T_c and the same cache as presented in Example 31. From Example 31, we know that the largest number of

cache lines that possibly ever need to be reloaded after T_a resumes from a preemption caused by T_c (indirectly or directly) is $C(Pa_c^{WMP}) = S(\tilde{M}_a \bigcup \tilde{M}_b, M_c^{WMP}) = S(\tilde{M}_{ab}, M_c^{WMP}) = 5$. With this number, we can estimate the CRPD caused by T_c preempting T_a by using Equation 9. Assuming the cache miss penalty is 10 clock cycles, we have $CRPD(T_a, T_c) = 5 \times 10 = 50$ clock cycles. □

In this section, we give a method to estimate CRPD where the estimation includes all effects due to any possible nested preemptions. In the next section, we show how to incorporate CRPD into WCRT analysis.

6.2.3 Enhanced WCRT Analysis Approach

The simple WCRT analysis given in Equation 6 does not include additional costs invoked by preemptions such as context switch cost and CRPD. By simply ignoring these additional costs, a theoretically schedulable system may fail in practice and miss a critical deadline. In this section, we discuss a way to overcome this problem.

An improved WCRT analysis is given by Busquests-Mataix et al. [4]. Inter-task cache interference is considered. The WCRT estimation equation is updated correspondingly as follows:

$$R_i^k = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + \gamma_j) \tag{10}$$

In Equation 10, γ_j is the additional overhead caused by inter-task cache interference. In the approach of Busquests-Mataix et al. [4], the authors assume that all cache lines used by the preempting task need to be reloaded after the preemption. As we pointed out in Chapter 5, this assumption exaggerates the cache reload cost for each preemption. We can apply the inter-task and intra-task cache eviction analysis techniques described earlier in Sections 5.1, 5.2 and 5.3 to reduce the overestimation in Equation 10. By incorporating the CRPD estimate given in Equation 8 and the context switch cost as given in Section 6.2.1 into Equation 10, we develop the following equation.

$$\begin{aligned}\gamma_j &= 2C_{cs} + CRPD(T_i, T_j) = 2C_{cs} + C(Pa_j^{WMP}) \times C_{miss} \\ &= 2C_{cs} + C_{miss} \times \sum_{r=0}^{N-1} min\{|\widehat{\widehat{m}}_{i,r}|,\ |\widehat{m}_{j,r}^{WMP}|,\ L\}\end{aligned} \tag{11}$$

The WCRT estimation equation can be updated correspondingly as below.

$$\begin{aligned}R_i^k &= C_i + \sum_{j \in hp(i)} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs}) \\ &= C_i + \sum_{j \in hp(i)} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + C(Pa_j^{WMP}) \times C_{miss} + 2C_{cs})\end{aligned} \tag{12}$$

We use Example 33 to help explain this equation.

Example 33: Suppose we have two tasks, T_1 and T_2. T_1 has a higher priority than T_2. The WCETs of T_1 and T_2 are 5 ms and 49 ms respectively. In other words, we have $C_1 = 5$ ms and $C_2 = 49$ ms. The period of T_1, P_1, is 30 ms. The period of T_2, P_2, is 100 ms. The CRPD caused by T_1 preempting T_2, $CRPD(T_2, T_1)$, is 3 ms. The context switch cost, C_{cs}, is 1 ms. Initially, we have the WCRT of T_2 be the same as its WCET, thus, $R_2^0 = 49$. By using Equation 12, we have $R_2^1 = 49 + \lceil \frac{49}{30} \rceil \times (5 + 3 + 2) = 69$. Continuing this way, we find $R_2^2 = 79$ and $R_2^3 = 79$. Since $R_2^2 = R_2^3$, we deem R_2 converges at 79. Thus, the WCRT estimate of T_2 is 79 ms. □

In this section, we elaborate our WCRT analysis approach, which incorporates CRPD. A new WCRT estimate formula is given. The new WCRT analysis approach can accommodate nested preemptions. In the next section, we summarize our WCRT analysis approach.

6.3 Overall WCRT Analysis and Schedulability Analysis

We call our WCRT analysis approach *WCRT Integrating Inter- and Intra-task cache timing analysis approach*, WI^3 for short. In this section, we summarize the steps performed by WI^3. We also discuss the computational complexity of WI^3.

6.3.1 Overall Approach of WI^3

To sum up, we follow the steps below to estimate the WCRT of each task in preemptive multi-tasking real-time system where each task is assigned a unique priority (for a full list

of assumptions, please see Section 2.1). The cache related preemption delay is included in our WCRT analysis.

Step 1. Derive the memory footprint of each SFP-PrS in each task by using the simulation method as proposed in SYMTA.

Step 2. Derive a WCET estimate for each task using SYMTA.

Step 3. Apply intra-task cache eviction analysis as given in Section 5.2 of Chapter 5 to obtain the MUMBS for every task except the task with the highest priority. The task with the highest priority cannot be preempted. Thus, its WCRT is the same as its WCET. Recall that MUMBS is the union of useful memory blocks over all execution points (see Definition 17). For a task T_a, the MUMBS of T_a, $\tilde{M}_a$, can be calculated as $\tilde{M}_a = \bigcup_s UMB_a(s)$, where $UMB_a(s)$ is the useful memory block set of T_a at the execution point s.

Step 4. Apply path analysis and inter-task cache eviction analysis as proposed in Section 5.1 and Section 5.4 of Chapter 5 to estimate the CRPD $CRPD(T_i, T_j)$ for each pair of tasks T_i and T_j where T_i has a lower priority than T_j. In this step, we follow the procedure below to estimate CRPD.

(i). Calculate CIIP for each memory block set. For preempted tasks, we calculate the CIIP for the MUMBS of this task. For each preempting task, we calculate the CIIP of the memory block set which contains all memory blocks that can possibly be accessed by the preempting task. Supposing we have a memory block set M, the CIIP of M, $\widehat{M}$, can be calculated with the following equation, as also given in Definition 15 in Section 5.1.1.

$$\widehat{M} = \{\widehat{m}_i | \widehat{m}_i \neq \emptyset,\ 0 \leq i < N\} \tag{13}$$

where, $\widehat{m}_i = \{m_k \in M | idx(m_k) = i\}$.

(ii). Use CIIPs to estimate the number of cache conflicts between the memory blocks accessed by the preempted task and the preempting task. Note that we need to perform such a calculation for every pair of tasks. Suppose we have two memory block sets M_a and M_b. The CIIPs of M_a and M_b are $\widehat{M}_a$ and $\widehat{M}_b$ respectively. We use the following equation to estimate the number of cache conflicts between M_a and M_b. This equation was given in

Equation 2 in Section 5.1.2.

$$S(M_a, M_b) = \sum_{r=0}^{N-1} min\{|\widehat{m}_{a,r}|,\ |\widehat{m}_{b,r}|,\ L\} \tag{14}$$

where N is the number of cache sets in a cache, $\widehat{m}_{a,r} \in \widehat{M_a}$ and $\widehat{m}_{b,r} \in \widehat{M_b}$.

To properly consider nested preemptions, we replace Equation 14 with the following equations, as explained in Section 6.2.2.

$$S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b) = \sum_{r=0}^{N-1} min\{|\bigcup_{l=b+1}^{a} \widehat{\tilde{m}}_{l,r}|,\ |\widehat{m}_{b,r}|,\ L\} \tag{15}$$

where $\tilde{M}_l$ is the MUMBS of task T_l. T_l is a task which has a priority higher than the priority of T_a but lower than the priority of T_b.

(iii). We perform path analysis on the preempting task. This step is optional as discussed in Section 5.4. When path analysis is performed, we have to search all paths in the preempting task. Thus, Equation 15 is updated correspondingly as follows.

$$S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b^k) = \sum_{r=0}^{N-1} min\{|\bigcup_{l=b+1}^{a} \widehat{\tilde{m}}_{l,r}|,\ |\widehat{m}_{b,r}^k|,\ L\} \tag{16}$$

where M_b^k is the memory block set which contains all memory blocks accessed by T_b when T_b runs along a particular path.

Step 5. Use the following iteration to calculate the WCRT for each task except the highest priority task. This iteration is a version of Equation 12.

$R_i^0 = C_i$;

$R_i^1 = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

...

$R_i^k = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

This iteration terminates when R_i converges or R_i is greater than the deadline of T_i. If the iteration converges, T_i can be scheduled. Otherwise, we cannot find a feasible schedule. In short, using the iterative WCRT calculation approach presented above, we can analyze the schedulability of the system based on the WCRT estimate of each task. In a system

where there are jitters, we assume that P_i is the minimum arrival interval of Task T_i [14]. With this assumption, our formula can handle jitters [14].

6.3.2 Computational Complexity

In this section, we discuss the computational complexity of the WI^3 approach. As introduced in Section 6.3.1, there are five steps in our WCRT analysis. We analyze the computational complexity of these five steps in this section.

Computational Complexity of Step 1.

In Step 1, we simulate each SFP-PrS to find the WCET of each SFP-PrS. We assume the WCETs of SFP-PrSes have a constant upper bound. This assumption is reasonable if we assume that the number of instructions in an SFP-PrS has a constant upper bound and all loops in a task have a constant upper bound as well. Because the number of instructions in an embedded application is limited, it is reasonable to assume a constant upper bound for the number of instructions in a SFP-PrS. Also, in order to analyze the WCET/WCRT of a task, we have to know the upper bounds of all loops in the task. Thus, the assumption of a constant loop upper bound is a requirement. Suppose we have a task T with a PCFG $G = \{V, E\}$, where V is the set of SFP-PrS nodes and E is the set of edges in the PCFG. We need to simulate all SFP-PrS nodes, which implies the computational complexity in this step is $O(|V|)$.

Please note that we could not find computational complexity analysis for this portion of SYMTA in [68, 70, 71, 72].

Computational Complexity of Step 2.

In Step 2, we build ILPs based on the task PCFG and the WCET of each individual SFP-PrS in the PCFG. The number of variables in the ILP equations is the same as the number of SFP-PrS nodes in the PCFG. The computational complexity of ILP in the worst case is exponential. In other words, the worst case computational complexity in this step is $O(2^{|V|})$. However, as stated by Wolf in [69], "As the numbers of equations for both the

granularity of basic blocks and program segments are far below the computation capabilities of efficient ILP solvers, the computation times are very low."

Computational Complexity of Step 3.

In Step 3, we calculate MUMBS for each task. MUMBS of a task is the union of useful memory blocks over all execution points of the task. For a given task T with PCFG=$\{V,E\}$, we calculate the set M of memory blocks that can possibly be accessed by T. According to the result given by Lee et al. [19, 20], the computational complexity of calculating useful memory blocks at one execution point in T is bounded by $O(|V||E||M|)$. Let the number of execution points in T be W. Thus, the computational complexity of MUMBS calculation is $O(|V||E||M|W)$. Let the number of nodes in any task PCFG be bounded by Γ. Similarly, let the number of edges in any task PCFG be bounded by Θ. Furthermore, let the number of memory blocks accessed by any task be bounded by Ω. Then, we can conclude that the computational complexity of MUMBS for any task is $O(\Gamma\Theta\Omega W)$. These bounds are reasonable because for a limited number of tasks, we can always find the task with the largest number of nodes in the PCFG, the task with the largest number of edges in the PCFG and the task with a largest number of memory blocks to be potentially accessed.

Computational Complexity of Step 4.

In Step 4, we first use Equation 13 in Step 4 in Section 6.3 to calculate the CIIP associated with the MUMBS of each preempted task and to calculate the CIIP of the memory block set that contains all memory blocks possibly accessed by a preempting task. In order to calculate CIIP of a memory block set M, we only need to find the index of each memory block in M and put each memory block into the corresponding element in CIIP. Thus, the computational complexity of CIIP calculation for a single memory block set M is $O(|M|)$.

CIIP is applied in Equation 15 in Step 4 in Section 6.3 to estimate the number of cache conflicts. In Equation 15, we have to calculate the CIIP for $\bigcup_{l=b+1}^{a}\tilde{M}_l$ and M_b^k in the case task T_a is preempted by T_b. Here, $\tilde{M}_l$ is the MUMBS of task T_l. M_b^k is the set of memory blocks when task T_b runs along a particular path. We assume there are n tasks. In the case

$a = n$ and $b = 0$, $\bigcup_{l=b+1}^{a} \tilde{M}_l$ is the union of $n-1$ MUMBSes. $n-1$ is the largest number of MUMBSes that can appear in this union. Because the number of memory blocks in each $\tilde{M}_l$ is bounded by Ω, the number of memory blocks in $\bigcup_{l=b+1}^{a} \tilde{M}_l$ is bounded by $n\Omega$. The number of memory blocks in M_b^k is bounded by Ω. Thus, the computational complexity of CIIP calculation in Equation 15 is $O(n\Omega)$.

Further, we use the CIIP of two memory block sets, $\bigcup_{l=b+1}^{a} \tilde{M}_l$ and M_b^k, to calculate the number of cache conflicts between these two memory block sets as shown in Equation 15. The computational complexity problem for this calculation is generalized as follows. Suppose we have two memory block sets M_1 and M_2. The CIIP of M_1 is $\widehat{M}_1 = \{\widehat{m}_{10}, ..., \widehat{m}_{1,N-1}\}$, where N is the number of cache sets/lines in a cache. The CIIP of M_2 is $\widehat{M}_2 = \{\widehat{m}_{20}, ..., \widehat{m}_{2,N-1}\}$. The number of cache conflicts between M_1 and M_2 is given by $S(M_1, M_2) = \sum_{i=0}^{N-1} \{min(|\widehat{m}_{1i}|, |\widehat{m}_{2i}|, L)\}$, where L is the number of ways in a cache. For each individual calculation $min(|\widehat{m}_{1i}|, |\widehat{m}_{2i}|, L)$, the computation time is constant. Usually in a system, the cache size is already known before performing cache-related timing analysis. In other words, N is a constant. Thus, the computational complexity of calculating $S(M_1, M_2)$ is a constant. In other words, after CIIPs are calculated, the computational complexity of Equation 15 is constant. Thus, assuming each memory block set is already known, the total computational complexity of Equation 15 is determined by CIIP calculation, which is $O(n\Omega)$.

If we perform path analysis on the preempting task, Equation 16 shown in Step 4 in Section 6.3 is used. In Equation 16, M_b^k is the set of memory blocks accessed by T_b when T_b runs along a particular path. In order to find the Worst MUMBS Path (WMP), we have to perform path analysis on T_b. For path analysis performed on a task, the computational complexity is exponential in the number of nodes that have two outgoing edges (i.e., the nodes that contain branches) in the PCFG of the task (note that a node in a PCFG can only have up to two outgoing edges according to Definition 9). We assume the number of the nodes with two outgoing edges in a PCFG is bounded by Γ_b. The computational

complexity of path analysis on a task is $O(2^{\Gamma_b})$.

In summary, suppose we have n tasks, in order to estimate CRPD, we follow the steps below.

(i). We calculate MUMBS for every task except the task with the highest priority. Each MUMBS has a computational complexity of $O(\Gamma\Theta\Omega W)$ as given above. The total computational complexity for MUMBS calculation is $O(n\Gamma\Theta\Omega W)$.

(ii). We calculate CRPD for every pair of tasks. If we have n tasks, we need to calculate n^2 CRPDs. For each CRPD, if we do not perform path analysis, the computational complexity is $O(n\Omega)$. If we perform path analysis, the computational complexity is $O(n\Omega 2^{\Gamma_b})$. Thus, the computational complexity of total CRPD calculation is $O(n^3\Omega)$ if path analysis is not performed and $O(n^3\Omega 2^{\Gamma_b})$ if path analysis is performed.

Therefore, the computational complexity for CRPD calculation is $O(n\Gamma\Theta\Omega W + n^3\Omega)$ if path analysis is not performed. the computational complexity for CRPD calculation is $O(n\Gamma\Theta\Omega W + n^3\Omega 2^{\Gamma_b})$ if path analysis is performed.

Path analysis can reduce the CRPD estimate, however, increases computational complexity. Therefore, as discussed in Section 5.4, we can choose to use path analysis or not. If the control structure of a task is not complicated, in other words, the number of nodes with two edges in a task PCFG is not large, we can perform path analysis in order to refine CRPD.

Computational Complexity of Step 5.

In Step 5, WCRT is estimated by using CRPD calculations. We here analyze the computational complexity of our WCRT analysis approach. Here, we assume that all CRPD values have already been calculated. In short, we assume here that Steps 1 through 4 have already completed and thus are not part of computational complexity analysis for Step 5.

The computational complexity of our WCRT analysis approach is is given in a theorem. Then, we give a proof of this theorem.

Theorem 1: Assuming the CRPD of each type of preemption is already known (calculated), the computational complexity of our WCRT analysis by using the iterative equation, Equation 12, is $O(n)$, where n is the number of tasks.

Proof: From Equation 12, we can find the following:

$R_i^k - R_i^{k-1} \geq 0$ and $R_i^k - R_i^{k-1} = \sum_{j=0}^{i-1}(\lceil \frac{R_i^{k-1}}{P_j} \rceil - \lceil \frac{R_i^{k-2}}{P_j} \rceil) \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

If for all j, where $0 \leq j \leq i-1$, $\lceil \frac{R_i^{k-1}}{P_j} \rceil - \lceil \frac{R_i^{k-2}}{P_j} \rceil = 0$, we have $R_i^k = R_i^{k-1}$. In this case, we conclude that R_i converges and $R_i = R_i^k$. Otherwise, if there exists a j, where $0 \leq j \leq i-1$, such that $\lceil \frac{R_i^{k-1}}{P_j} \rceil - \lceil \frac{R_i^{k-2}}{P_j} \rceil \neq 0$, we have $\lceil \frac{R_i^{k-1}}{P_j} \rceil - \lceil \frac{R_i^{k-2}}{P_j} \rceil \geq 1$ (due to the fact that $\lceil x \rceil$ yields only integer values). In this case, we have the following,

$$\sum_{j=0}^{i-1}(\lceil \frac{R_i^{k-1}}{P_j} \rceil - \lceil \frac{R_i^{k-2}}{P_j} \rceil) \times (C_j + CRPD(T_i, T_j) + 2C_{cs}) \geq min_{j=0}^{j=i-1}(C_j)$$

In other words, $R_i^k - R_i^{k-1} \geq min_{j=0}^{j=i-1}(C_j)$. Therefore, R_i has to increase monotonically before the iteration terminates. R_i has to be increased by at least $min_{j=0}^{j=i-1}(C_j)$ in each iteration. On the other hand, R_i cannot exceed P_i. Thus, the number of iterations is limited by $\frac{P_i}{min_{j=0}^{j=i-1}(C_j)}$. This implies that the number of iterations has a constant upper bound when the periods and the WCET of tasks are determined. In other words, WCRT estimate for one task by using this iteration method has a constant time complexity. We need to perform such WCRT estimate for all tasks except the task with the highest priority. Thus, the total computational complexity is $O(n)$.□

Overall Computational Complexity of WI^3 approach.

Based on the computational analysis above, we can analyze the overall computational complexity of our WI^3 approach.

The overall computational complexity in our approach consists of two parts, the computational complexity of SYMTA and the computational complexity of our WCRT analysis approach. We have already given the computational complexity of SYMTA above. In SYMTA, the computational complexity of deriving WCETs for all SFP-PrSes is $O(|V|)$. The computational complexity of solving ILP equations is $O(2^{|V|})$ in the worst case. Thus,

the total computational complexity of SYMTA is $O(|V| + 2^{|V|}) = O(2^{|V|})$ in the worst case.

Excluding the first two steps calculated by SYMTA, WI^3 approach consists of CRPD estimation and WCRT estimation. By combining these two parts, we can conclude that our WCRT analysis approach has a computational complexity of $O(n\Gamma\Theta\Omega W + n^3\Omega + n) = O(n\Gamma\Theta\Omega W + n^3\Omega)$, if we do not perform path analysis on the preempting task. In other words, our WCRT analysis approach, excluding SYMTA, has a polynomial computational complexity if we do not perform path analysis on the preempting task. If we perform path analysis on the preempting task, the computational complexity of the overall CRPD and WCRT analysis approach is $O(n\Gamma\Theta\Omega W + n^3\Omega 2^{\Gamma_b} + n) = O(n\Gamma\Theta\Omega W + n^3\Omega 2^{\Gamma_b})$. We keep path analysis as an option that can be selected by users depending on the complexity of control structures in preempting tasks. Usually embedded applications have relatively simple control structures; thus, we can search the worst MUMBS path without requiring much computation time. By using path analysis, we can reduce CRPD and WCRT estimate significantly.

Note that in [21, 22], in order to estimate the WCRT for one task, all the preemption scenarios have to be investigated. The total number of preemption scenarios are exponential in the number of tasks; thus, the approach of [21, 22] has exponential complexity in WCRT estimation. Therefore, our overall CRPD and WCRT analysis approach without using path analysis and excluding computation time taken by SYMTA has only polynomial complexity and thus is more feasible and scalable than Lee et al. [21, 22] when there are a large amount of tasks in the system. (Please note that WI^3 without path analysis is equivalent to Lee et al. [19, 20] since Lee et al. do **not** perform path analysis on the preempting task.)

Finally, the overall complexity of WI^3 including computation time taken by SYMTA is $O(n\Gamma\Theta\Omega W + n^3\Omega + 2^{|V|})$ if path analysis is not performed. If path analysis is performed, the overall complexity of WI^3 including computation time taken by SYMTA is $O(n\Gamma\Theta\Omega W + n^3\Omega 2^{\Gamma_b} + 2^{|V|})$. However, please note that in practice, SYMTA requires little

computation time since, as claimed by Wolf [69], the ILP equations of SYMTA can be solved efficiently by ILP solvers for most practical real-time applications.

6.4 Summary

In this chapter, we incorporate CRPD into WCRT analysis. We propose a new WCRT analysis formula, which is the fourth contribution claimed in Section 1.3, **Contribution 4: A new WCRT estimate formula is proposed**. This formula can accommodate nested preemptions. As discussed in Section 6.3.2, with respect to the number of tasks *n*, our approach without using path analysis has a computational complexity as compared to the exponential complexity of the approach of Lee et al. In the next chapter, we illustrate prioritized cache design. A prioritized cache can reduce inter-task cache interference. We will also modify our WCRT analysis approach in this chapter to analyze the behavior of a prioritized cache.

CHAPTER VII

PRIORITIZED CACHE

The difficulties in estimating WCET/WCRT of a real-time task in a multi-tasking environment with caches lie in cache interference among tasks. Because cache resources are shared among tasks, cache lines used by one task may be evicted by another task when the former is suspended. Disabling cache sharing among tasks may help reduce cache interference. An intuitive way to solve this problem is to divide the cache into several partitions. Each task is assigned one or more partitions exclusively so that interference among tasks is eliminated. Now, we need to design the partitioning algorithm and determine how the partitions are assigned to tasks. Here, we propose a prioritized cache based on cache partitioning.

7.1 Hardware Design

A prioritized cache is a variant of a conventional set associative cache. In a multi-way set associative cache, each way is called a column [5, 13, 52]. For example, a 4-way set associative cache has four columns. The cache is partitioned at the granularity of columns in a prioritized cache. Cache partitions are then used by tasks according to their priorities. Inter-task cache conflicts are reduced. A prioritized cache is different from a conventional set associative cache in the cache replacement algorithm. We can control a prioritized cache directly by accessing control registers in the prioritized cache. Alternatively, we can slightly modify the scheduler function of an Operating System (OS) in order to support the prioritized cache. The prioritized cache is transparent to applications that run with an OS supporting a prioritized cache.

7.1.1 Cache Replacement Algorithm in a Prioritized Cache

A prioritized cache is partitioned at the granularity of columns. A column may be assigned to a particular task. When a column in the cache is assigned to a task, that task is called the owner of the column and the column is owned by the task. Not all columns need to be assigned to tasks. We can also set a column to the status of "shared" so that the column can be shared by tasks. Shared columns avoid the situation where some low priority tasks do not have any cache columns to use.

The basic idea behind the prioritized cache is to assign cache partitions to tasks according to their priorities. Priorities are widely used in task scheduling of real-time systems. Depending on the scheduling algorithm chosen, priorities of tasks may be fixed – e.g., Rate Monotonic Scheduling (RMS) – or dynamic – e.g., Earliest Deadline First (EDF) or Priority Ceiling Protocol (PCP). Usually, those tasks with strict timing constraints have higher priorities in using CPU resources. We need to notice that these existing scheduling algorithms (e.g., RMS, EDF and PCP as mentioned above) are almost always only used to allocate CPU resources. The priorities of tasks are not taken into account in conventional cache allocation. However, tasks with strict timing constraints should also have higher priorities in using other resources such as caches. With this intuition, we divide a cache into partitions and assign partitions to each task according to its priority. In this thesis, we do not address the problem of choosing priorities for tasks, but assume that each task has been assigned a unique priority (not assigned to any other task) with an existing priority-based scheduling algorithm such as RMS or EDF. We focus instead on how to assign cache partitions to tasks according to their priorities.

Now, we assume that priorities of tasks range from 0 to $p_l - 1$. 0 is the highest priority and p_l is the lowest priority. We also give each column a priority. At the beginning, the priority of every column in the cache is the lowest one ($p_l - 1$). When a task needs to use the cache, the cache controller compares the priority of the task with the priority of each column. To accomplish this, we add a special register to the cache controller as described

in the next section. Only when the priority of a task is equal to or higher than the priority of a column can the task use the column. In other words, a task with a higher priority can use all columns owned by tasks with lower priorities. When a column is used by a task, the priority of the column is upgraded to be equal to the priority of the task. After a task completes, it notifies the cache controller to release all columns the task owns. The cache controller does this by setting priorities of those columns to the lowest priority again. Let us consider an example as below.

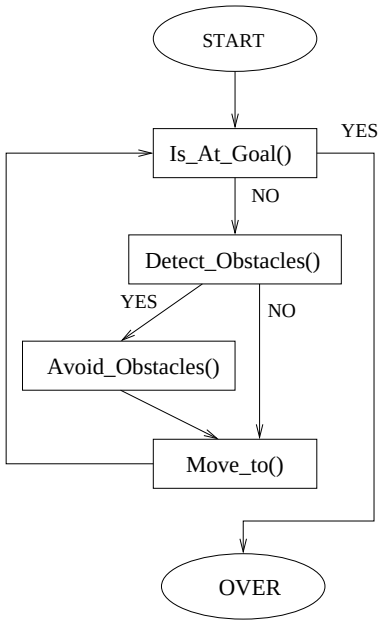


Figure 26: PCFG of MR

Example 34: Suppose we have two tasks, an MPEG decoder (MPEG for short) and a Mobile Robot Control program (MR for short). The MR application is derived from Missionlab, which is a mobile robot control software developed by the Georgia Tech Mobile Robot Lab [43]. The PCFG of MR is shown in Figure 26.

MPEG and MR are two different kinds of applications. MPEG is a data-processing application with soft real-time constraints. MR controls the behavior of a robot based on a small set of data such as the coordinates of the robot and the coordinates of obstacles. However, MR has a more strict timing requirement than MPEG.

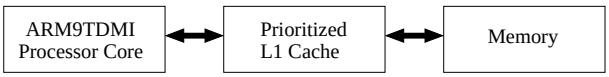


Figure 27: Architecture for Executing MPEG and MR

Thus, a tight WCET analysis for MR is needed. According to the PCFG shown in Figure 26, we can see that the worst case control path in MR is the path from *Is_At_Goal( )* to *Move_To( )* via *Detect_Obstacles( )* and *Avoid_Obstacles( )*.

We assume that MR has a higher priority than MPEG. In this example, there are 4 priorities, where 3 is the lowest and 0 is the highest priority. MR is given a priority of 1 and MPEG is given a priority of 2. We use a 4-way 16K set associative L1 cache for all instructions and data. Each column has 256 lines. The processor used in this example is ARM9TDMI. Figure 27 shows the architecture for the example. At the very beginning, all columns in the cache are empty and thus have the lowest priority of 3. When MPEG runs, it uses all four columns. The priorities of these four columns are upgraded to the priority of MPEG, i.e., to 2 as shown in Figure 28(b). Then, MPEG is suspended and MR begins to run. When there is a cache miss, a cache line is chosen to be replaced. If the priority of the column in which the cache line locates is lower than the priority of the task, the priority of this column is upgraded to the priority of the task. In this example, two columns used by MPEG are replaced by MR and the priorities of these two columns are upgraded to 1 as shown in Figure 28(c). So, next time when MPEG is executed, MPEG can only use the other two columns that still have priority 2. From this example, we can see that if there is no other task with an equal or higher priority than MR, MR can use the first two columns exclusively. In this manner, we can guarantee the usage of the cache by high priority tasks at a cost, however, of degrading the performance of lower priority tasks.

When a task is completely over, it releases all columns it owns. The cache controller sets the priorities of these columns to the lowest priority. In the example

above, we assume that MR is completed earlier than MPEG. When MR is over, it releases the first and the second column and sets the priorities of these two columns to 3, which is shown in Figure 28(d). Therefore, when MPEG is executed next time, it can use all four columns again, as shown in Figure 28(e). □

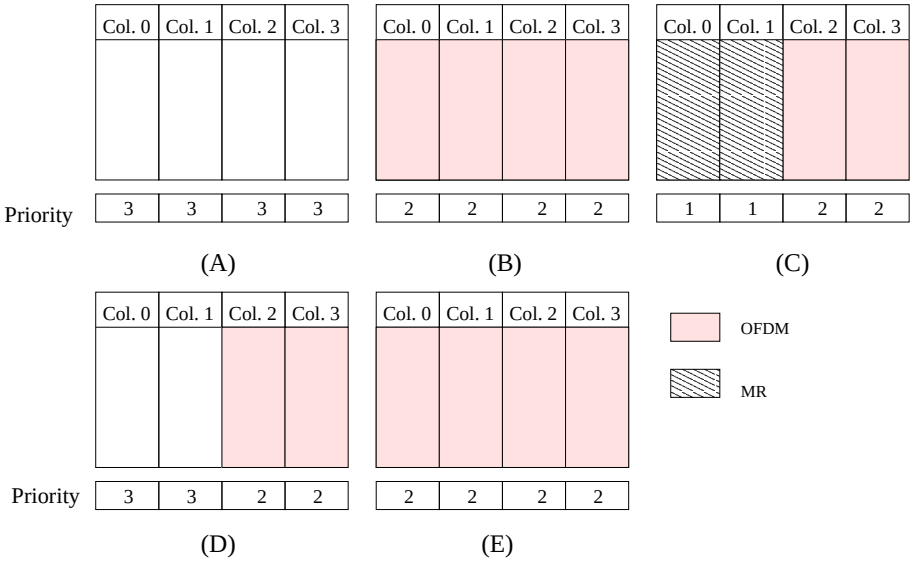


Figure 28: An Example of the Assignment Strategy in a Prioritized Cache

In this section, we explain the cache replacement algorithm in a prioritized cache. In the next section, we give details about implementation of such a cache replacement algorithm.

7.1.2 Cache Controller and Status Registers

The prioritized cache is a variant of a set associative cache. Figure 29 shows a prototype of a prioritized cache. The parts that are different from a conventional cache are enclosed by a square composed of a dashed line. The cache controller is modified and some registers are added to save and control the status of each column.

The prioritized cache uses a replacement algorithm different from any used by a conventional set associative cache. The replacement algorithm is implemented in the cache controller. We add two tables, the Column Priority Table (CPT) and the Column Owner Table (COT), and three registers, Current Task Register (CTR), Current Task Priority Register (CTPR) and Column Status Register (CSR), to the cache controller. Each column has

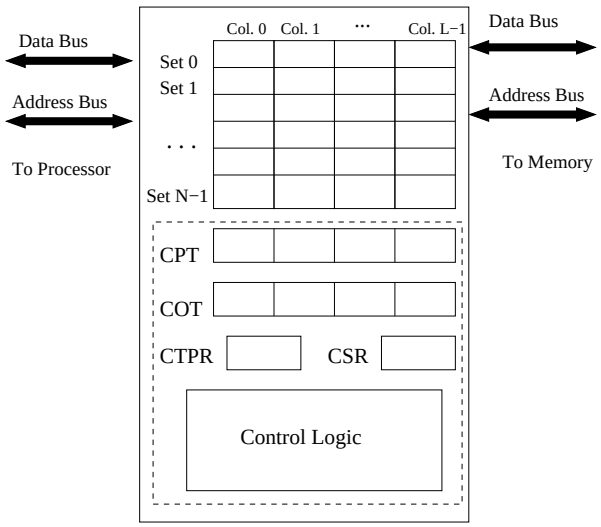


Figure 29: A prototype of a prioritized cache

an entry in CPT and COT. An entry records the priority and the owner of the column. CTR and CTPR are used to save the ID and the priority of the task which is currently running on the processor. CSR indicates if a column is shared. Setting a column as shared allows all tasks to use this column. Note that a shared column differs from a column with the lowest priority in that a shared column always has the lowest priority with this priority never being upgraded even if the column is used by a high priority task. Each column has one "share" bit in the CSR. If the bit is set, the corresponding column is shared.

A prioritized cache uses only a task's priority to allocate a column. The prioritized cache does not limit the number of tasks and priorities directly, except for the limitation imposed by the length of the CPT and COT registers. For example, if the CPT and COT entries each have 16 bits, up to 2^{16} tasks and 2^{16} different priorities can be supported, which is sufficient for many real-time systems. Suppose we have an L-way set associative cache, a maximum of 2^N tasks and a maximum of 2^K different priorities. Clearly, then, we have L entries each in the COT and CPT tables (i.e., L columns or ways). Each COT entry needs N bits, for a total usage of $L \times N$ bits. Each CPT entry needs K bits, for a total usage of $L \times K$ bits. The CTR register has N bits, while CTPR has K bits. Additionally, the CSR register needs L bits. Therefore, in total, we need $L \times N + L \times K + N + K + L = (L+1)(K+N) + L$

bits for the CPT and COT tables and the CTR, CTPR and CSR registers. Example 35 shows these extra tables and registers in a 4-way set associative prioritized cache. The prioritized cache does not require much more area than a normal cache. For a prioritized 16K 4-way cache which supports 64 tasks and 64 priorities, the area increases by less than 5% compared with a set-associative cache of the same size and associativity.

Example 35: Suppose we have a 16KB cache with 4 columns as shown in Figure 29. The lengths of the specialized registers in the cache – CPT, COT, CSR and CTR – are 16 bits each. Since the CPT and COT registers are each 16 bits long, this cache supports up to 2^{16} tasks and 2^{16} priorities. In this example, K=the number of bits in the CPT register=16, N=the number of bits in the COT register=6, L=the number of columns=4; thus, we need (L+1)(K+N)+L=(4+1)(16+16)+4=164 extra bits for the prioritized 16KB cache. □

In Example 35, the prioritized cache needs to be initialized before it is used (as shown in Figure 28(a)). We notice that most Operating Systems have an IDLE task which controls the CPU when there is no other tasks running. The IDLE task has the lowest priority. Therefore, we can assign the ID and the priority of the IDLE task to the corresponding registers in the prioritized cache for initialization. We assume that the IDLE task has an ID of 0 and a priority of $0xFFFF$ which is the lowest priority. Therefore, all of the CPT entries in Figure 30 are $0xFFFF$, and all of the COT entries are zero. Also, in order to allow low priority tasks to use the cache, we set the fourth column as shared by default. The initial settings of the registers are listed in Figure 30.

CPT	0XFFFF	0XFFFF	0XFFFF	0XFFFF
COT	0X0000	0X0000	0X0000	0X0000

CTR	0X0000

CTPR	0XFFFF

CSR	0001

Figure 30: Initialization of Registers in the Prioritized Cache

We give memory mapped addresses to the tables and the registers so that users can set the values in the tables and registers as needed. Alternatively, specialized assembly

instructions for accessing these tables and registers can be defined if the target instruction set has a sufficient number of undefined assembly instructions and if the processor can be redesigned to support the new specialized assembly instructions. Usually, we only need to set the value of CTR and CTPR. Every time when the task is switched, we write to the CTR and CTPR the task ID and the priority of the task to be executed, respectively.

When the cache hits, the prioritized cache works the same as a conventional cache. When there is a cache miss, a cache line has to be selected to hold the data/instructions loaded from the memory. We select the cache line from a cache column in the following order.

(1). An invalid cache line in a column owned by the current task.

(2). A cache line in a column that is not owned by any task (not the shared columns but columns that have not yet been allocated, i.e., whose owner is the IDLE task).

(3). A cache line in a column that is owned by a lower priority task (other than the IDLE task, which falls under (2) above).

(4). A cache line in a shared column.

If the cache line to be replaced is located in a column that is not owned by the current task and this column is not a shared column, the priority and the owner of the column are updated, that is, the values in CTR and CTPR are copied to the corresponding entries in COT and CPT. When a column is released, the priority of the released column is set to the lowest priority and the corresponding entry in COT is set to the ID – i.e., "0" – of the IDLE task. Example 36 shows how the registers in a prioritized cache are updated.

Example 36: Suppose we have two tasks, MR and MPEG, as described in Example 34 and Figure 28. We assume MPEG has an ID of 2 and MR has an ID of 7. The priority of MPEG is 2, while the priority of MR is 1. Column 3 is set as shared by default in order to allow low priority tasks to use the cache in the worst case that all other columns are allocated to high priority tasks. After MPEG runs, Columns 0, 1 and 2 are owned by MPEG as shown in Figure 28(B). MPEG also

Table 2: An Example of Cache Replacement in a Prioritized Cache

Line	CTR	CTPR	CSR	Priority of Columns (CPT)				Owner of Columns (CPT)			
				0	1	2	3	0	1	2	3
1	2	2	0001	2	2	2	3	2	2	2	0
2	7	1	0001	2	2	2	3	2	2	2	0
3	7	1	0001	1	1	2	3	7	7	2	0

uses the shared column, Column 3. The status of registers in the prioritized cache is shown in Line 1 of Table 2. Then, MR begins to run for the first time. MR writes its ID and priority to the CTR and CTPR registers, respectively, as shown in Line 2 of Table 2. MR needs to use cache lines to save its instructions and data in the cache. Since MPEG has a lower priority than MR, the first column, which is owned by MPEG, is selected and assigned to MR. For the same reason, the second column is also assigned to MR, as shown in Figure 28(c). The values in registers are changed as shown in Line 3 of Table 2. □

As presented in this section, additional registers are introduced in the cache controller of a prioritized cache. The cache replacement algorithm depends on the values in those registers. Each register in a prioritized cache is mapped to a memory address. By reading/writing these registers, users can control a prioritized cache using software. Notice that users of a prioritized cache do not need to handle memory-to-cache mapping directly. Thus, there is no need to customize TLB design in a system using our prioritized cache. Software interface details for a prioritized cache are introduced in the next section.

7.2 Software Interface

Registers in a prioritized cache are mapped to the memory address space. We provide some APIs to access these registers in order to provide users with control of a prioritized cache. However, application developers do not need to deal with the prioritized cache directly if the OS is modified to support the prioritized cache. Only the context switch function and

the task destruction function in an OS are required to be modified slightly.

7.2.1 APIs for Controlling a Prioritized Cache

The prioritized cache is software controllable. We provide APIs for users to configure the cache. API functions can change the values in COT, CPT, CTR, CSR and CTPR to assign or release the columns. We provide four APIs. *Set_tid_pri(tid,pri)* writes the priority and ID of the current task into CTR and CPTR, respectively. *Set_column_pri(col,pri)* sets the priority of a column. *Release_column(tid)* releases all columns owned by the task with an ID of tid. *Set_ column_shared(col)* sets a column to a status of shared. These APIs can be implemented as system calls in an OS. We give an example below to show how we use these APIs in the MPEG and MR applications.

```
1. Set_column_shared(3);
2. while(!MPEG_over()&&!MR_over()){
3.      Set_tid_pri(MPEG_ID,2);
4.      MPEG_decode_one_slice();
5.      Set_tid_pri(MR_ID,1);
6.      MR_move_one_step();
7.     if(MPEG_over())
8.          Release_column(MPEG_ID);
9.      if(MR_over())
10.         Release_column(MR_ID);
11. }
```

Figure 31: Code Example Using APIs

Example 37: Consider the example shown in Figure 28. If we set Column 3 to be shared and execute MPEG and MR alternatively, we can implement this example with the C code shown in Figure 31. Notice that, in order to explain the APIs with a simple example, we do not use an RTOS to schedule the tasks. Instead, the MPEG function and the MR function are switched manually. Table 3 shows how the CTR, CTPR, CSR, the priority and the owner of each column changes after each line of code in Figure 31 is executed. We give MPEG an ID of 2 and MR an

Table 3: Values in registers of the prioritized cache

Line	CTR	CTPR	CSR	Priority of Columns				Owner of Columns			
				0	1	2	3	0	1	2	3
Initial	0	0	0000	3	3	3	3	0	0	0	0
1	0	0	0001	3	3	3	3	0	0	0	0
3	2	2	0001	3	3	3	3	0	0	0	0
4	2	2	0001	2	2	2	3	2	2	2	0
5	7	1	0001	2	2	2	3	2	2	2	0
6	7	1	0001	1	1	2	3	7	7	2	0
8	7	1	0001	1	1	3	3	7	7	0	0
10	7	1	0001	3	3	3	3	0	0	0	0

ID of 7. After line 1 of Figure 31 is executed, Column 3 is set to be shared: thus, the value in CSR is changed to 0001 as can be seen in the second row of Table 3. *Set_tid_pri( )* is called in line 5 of Figure 31 in order to write the ID and priority of MR to CTR and CTPR. Thus, the values in CTR and CTPR are changed to 7 and 2 respectively as can be seen in the fifth row of Table 3. Then, MR starts to run. As described in Example 36, the first two columns of MPEG are assigned to MR. Thus, the register values are changed as shown in the sixth row of Table 3. In Line 8 of Figure 31, MPEG releases all of its columns if MPEG is over, which is the case in this example. Thus, the register values indicating column priorities and owners change as shown in the seventh row of Table 3. When MR is over, it also releases all of its columns. The last row of Table 3 gives the status of registers after MR is over. □

7.2.2 Embedding Prioritized Cache APIs in an OS Kernel

We provide APIs for users to configure the prioritized cache. However, users do not need to call these APIs directly; instead, the APIs can be embedded into the OS system calls. For example, we can insert *Set_tid_pri()* into the scheduling function so that every time a task is switched, the priority and the ID of the current task is written into CTR and CTPR, respectively, in the prioritized cache. We can also embed *Release_Column()* into the task destruction function so that when a task is completed, all the columns it owns are released.

Obviously, the changes needed to be made in the OS are minor. By embedding prioritized cache control APIs into the OS kernel, the details of the prioritized cache can be made transparent to users. Thus, users can focus on application development at a higher level of abstraction. Figure 32 shows the pseudo-code of the updated scheduling function in Atalanta RTOS [53]. The line in bold is added to support the prioritized cache.

```
void reschedule(void)
{
    disable_interrupt()
     ...
    old_stack=task[current_task].stack;
    new_stack=task[next_task].stack;
    contextswtich();
    current_task=next_task;
    Set_tid_pri(task[current_task].tid,task[current_task].pri);
    ...
    enable_interreupt();
}
```

Figure 32: The updated scheduling function in Atalanta RTOS

In this section, we introduce software support for a prioritized cache. We can use a prioritized cache by using APIs directly or by modifying OS. In the next section, we adapt our WCRT analysis approach for a prioritized cache.

7.3 WCRT Analysis for a Prioritized Cache

Customized caches can be demonstrated to be effective in eliminating inter-task cache conflicts by running benchmarks or evaluating average performance (e.g., cache miss rate). However, lack of worst case analysis is not acceptable in a real-time system. Benchmark applications may not cover all situations. Average performance cannot guarantee worst-case performance. Thus, in order to apply a customized cache in a real-time system safely, we need to analyze the WCET/WCRT formally.

We divide prioritized cache usage into two stages. In the first stage, the cache columns are allocated to tasks. Cache evictions may happen if columns used by low priority tasks are allocated to high priority tasks. In this stage, tasks run with a cold cache. Because a

task may have multiple feasible paths, the task may not execute all SFP-PrSes in one run. In other words, a task may request more cache columns after the first run. Thus, the cache columns owed by a task may change dynamically. A high priority task may acquire more cache columns and a lower priority task may lose cache columns subsequently. However, after all SFP-PrSes of a task are executed at least once, cache columns allocated to this task become stable. Therefore, for the purpose of WCRT analysis, we run each task one or more times with carefully selected input data so that every SFP-PrS in the task is executed at least once in the first stage. Tasks are executed in this way in the descending order of task priorities. After this stage, cache allocation is completed. So, in the second stage, all tasks are allocated a portion of columns. The prioritized cache works in the second stage for the rest of time. Thus, we call the first stage *transit stage* and the second stage *stable stage*.

We assume that we can use the behavior of a prioritized cache in the stable stage to determine the timing properties of tasks. Otherwise, to analyze the transit stage, the assumption of a cold cache can be used. In the stable stage, cache eviction only happens in the shared columns. Notice that we assume each task has a unique priority (see Section 2.1 for a full list of all of our assumptions).

Usually, WCET/WCRT analysis assumes a cold cache when a task starts to run. This assumption is reasonable in a system where the cache is shared by all tasks because the cache lines used by one task may be evicted by other tasks. In the worst case, a task has to reload all data and instructions from the memory to the cache after a context switch. This cache reload cost due to the cold cache assumption applies to all multi-tasking systems, either preemptive or non-preemptive. Thus, we call this type of cache reload cost *non-preemption related cache reload cost*. Non-preemption related cache reload cost is included in the WCET estimate for each task. Non-preemption related cache reload cost affects both WCET and WCRT of tasks. An example is given as below.

Example 38: Figure 33 shows a scenario in a preemptive multi-tasking system. A

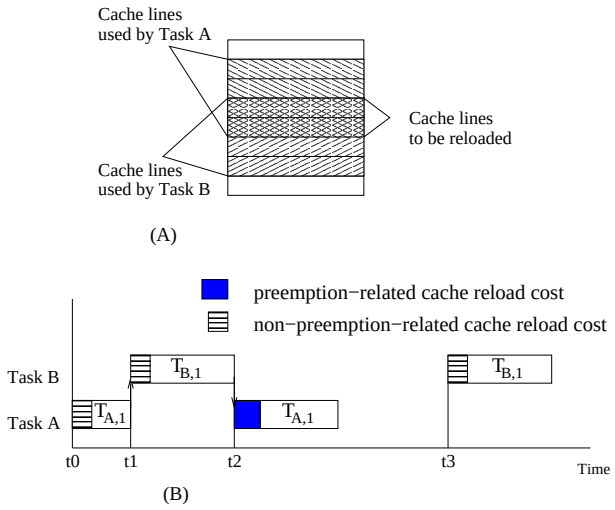


Figure 33: An example of inter-task cache eviction

low priority task, Task A, is preempted by a high priority task, Task B, at time t_1. During the preemption, Task B uses some cache lines that were used by Task A before the preemption. The memory blocks loaded to these cache lines by Task A are thus evicted. After Task A resumes at time t_2, it again needs to access some of the memory blocks evicted by Task B. Task A has to reload those memory blocks to the cache. Furthermore, after Task A resumes, Task A may evict some cache lines used earlier by Task B as well. When Task B is executed for the second time at time instant t_3, Task B also needs to reload some cache lines. Cache reload caused by inter-task interference extends the response times of Task A and Task B, as shown in Figure 33(B).□

Notice that, in a system where the cache is used by a task exclusively, the assumption of a cold cache is too conservative. After the first run of the task, the cache is filled with some data and instructions used by the task. Since the cache is not shared, no inter-task cache conflicts exist. When the task runs again later, the cache is already warmed up. In SYMTA [68], an iterative method is provided to calculate the minimum set of memory blocks that have resided in the cache because of previous executions.

As discussed earlier in the thesis, in a preemptive multi-tasking system, preemptions

can cause additional cache reload cost, which is called CRPD. This CRPD only happens in preemptive multi-tasking systems. Thus, we also call this type of cache reload cost *preemption-related cache reload cost*. Preemption-related cache reload cost (i.e., CRPD) of conventional caches are analyzed in Chapter 5. Now, we apply this approach to analyze the CRPD for the prioritized cache.

In a prioritized cache in the stable stage, tasks only conflict in the shared columns. The tasks that do not use shared columns do not have cache interference with other tasks (since we are in the stable stage). Thus, all preemptions related to these tasks do not incur CRPD. Since cache columns are allocated to tasks according to task priorities, high priority tasks have higher priority in using cache columns. Suppose we have a set of tasks T_i, $(0 \leq i < n)$, sorted in the descending order of their priorities. Here, n is the number of tasks. In other words, if $i < j$, T_i has a higher priority than T_j. In this case, if T_j can use some non-shared cache columns in the stable stage, T_i must own some non-shared cache columns as well. (Otherwise, T_i will occupy cache columns used by T_j since T_j has a lower priority.) Thus, in the stable stage, we divide tasks into two groups. In the first group, we have tasks T_0, T_1, ..., T_q, where $0 \leq q < n$. We use $\psi_1 = \{T_l | 0 \leq l \leq q\}$ to represent this group of tasks. All the tasks in this group do not use any shared columns. Thus, each task in ψ_1 does not conflict with any other task. In the second group, we have tasks T_{q+1}, ...,T_{n-1}. We use $\psi_2 = \{T_l | q < l < n\}$ to represent this second group of tasks. The tasks in ψ_2 use shared cache columns; thus, each task in ψ_2 may conflict with other tasks in ψ_2.

Suppose we have two tasks, T_a and T_b. T_b has a higher priority than T_a. In the stable stage, there are only two possibilities for $CRPD(T_a, T_b)$ as follows:

1. T_b is in the task group set ψ_1. In this case, T_b owns cache columns exclusively. As a result, the cache lines used by T_b do not overlap with the cache lines used by T_a; thus, $CRPD(T_a, T_b) = 0$.

Note that T_a has a lower priority than T_b. If T_a is in ψ_1, T_b must be also in ψ_1. Thus, we do not need to consider the case where T_a is in ψ_1, since the case of $T_a \in \psi_1$ is already

covered by Case 1 above.

2. Neither T_a nor T_b is in ψ_1. In other words, both T_a and T_b use shared columns. In this case, cache eviction only happens in the shared columns. We can modify the CRPD analysis equation, Equation 5 in Section 5.5, to estimate the CRPD after T_a resumes from a preemption. The necessary modifications are shown in Equation 17 below, which is based on Equation 5.

$$CRPD(T_a, T_b) = \begin{cases} 0 & T_b \in \psi_1 \\ S(\bigcup_{l=b+1}^{a} \tilde{M}_l, M_b^{WMP}) \times C_{miss} & T_b \in \psi_2 \end{cases} \quad (17)$$

We use Example 39 to explain how a prioritized cache can affect cache interference among tasks.

Example 39: Suppose we have three tasks: MR, ED and OFDM (these three tasks were explained in Example 14. Suppose further we have a 32KB 8-way prioritized cache. Each cache line has 16 bytes. Thus, each column has 256 cache lines. MR uses Column 0 to Column 4 and ED uses the rest of columns. The last two columns are set as shared. Thus, OFDM can only use the last two columns. OFDM has no conflicts with MR, but OFDM and ED may conflict in the shared columns in the prioritized cache. We can derive the memory footprints of ED and OFDM first, then use the CIIP based approach as given in Equation 4 to estimate the number of conflicts between OFDM and ED in the shared columns. In this example, the estimate of an upper bound on the number of cache conflicts between OFDM and ED is 160.

If we use a conventional cache, all three tasks, MR, ED and OFDM conflict with each other in the cache. Again, by using our CIIP based approach, we can derive the estimate of cache conflicts between MR and OFDM, which is 88. The estimate of the number of cache conflicts between ED and OFDM is 98.

Assuming cache miss penalty is 10 clock cycles, the CRPD caused MR preempting

OFDM is zero in a prioritized cache. As a comparison, the CRPD caused MR preempting OFDM is 880 clock cycles in a conventional set-associative cache. Therefore, a prioritized cache can be quite effective in preventing high priority tasks conflicting with low priority tasks. □

In Example 39, although the number of cache conflicts between ED, MR and OFDM is reduced, OFDM cannot use the full cache. Thus, the WCET of OFDM is expected to increase. We examine this impact in our experiments.

Now, let us consider how to adapt our WCRT analysis approach to a prioritized cache. Based on the CRPD given in Equation 17, we can modify our WCRT analysis approach for the prioritized cache as explained next.

For each task T_i, if $T_i \in \psi_1$, T_i does not conflict with any other tasks in the cache. For each preemption, we only need to consider the context switch cost and the WCETs of preempting tasks. Thus, the WCRT analysis formula introduced in Section 5.5 can be modified as follows.

$R_i^0 = C_i$;

$R_i^1 = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + 2C_{cs})$

...

$R_i^k = C_i + \sum_{j=0}^{i-1} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + 2C_{cs})$

On the other hand, if $T_i \in \psi_2$, T_i may conflict with any other task in the task group ψ_2. But T_i does not conflict with any task in ψ_1. Thus, if T_i is preempted by a task T_j in ψ_1, for overhead (i.e., over and above actual worst-case task execution time C_j of T_j) we need only consider the context switch costs. If T_i is preempted by a task in ψ_2, we need to consider both CRPD and the context switch costs. Therefore, we use the formula below to estimate the WCRT of T_i.

$R_i^0 = C_i$;

$R_i^1 = C_i + \sum_{j=0}^{q} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + 2C_{cs}) + \sum_{j=q+1}^{i-1} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

...

$R_i^k = C_i + \sum_{j=0}^{q} \lceil \frac{R_i^0}{P_j} \rceil \times (C_j + 2C_{cs}) + \sum_{j=q+1}^{i-1} \lceil \frac{R_i^{k-1}}{P_j} \rceil \times (C_j + CRPD(T_i, T_j) + 2C_{cs})$

In this section, we adapt our WCRT analysis approach to a prioritized cache. By using WCRT analysis, we can predict the worst case timing properties of a prioritized cache, which allows us to use a prioritized cache in a real-time system safely.

7.4 Summary

In this chapter, we present a prioritized cache, which is the fifth contribution claimed in Section 1.3, **Contribution 5: A novel "prioritized cache" design is presented to reduce CRPD**. By using a prioritized cache, we can reduce inter-task cache interference. As a result, WCRTs of tasks are tightened. The WCRT analysis approach in Chapter 6 is adapted to analyzed the behavior of a prioritized cache. Such a formal analysis allows us to use a prioritized cache safely in real-time systems. In the next chapter, we use some application to evaluate the performance of our WCRT analysis approach and the prioritized cache.

CHAPTER VIII

EXPERIMENT SETUP

In this thesis, we propose a WCRT analysis approach for preemptive multi-tasking systems. In order to tighten task WCRT by reducing cache interference among tasks, we design a prioritized cache. We further develop some real-time applications to evaluate the performance of our WCRT analysis and the prioritized cache. In this chapter, we explain the environment and the flow of experiments in detail; however, actual experimental results (as opposed to tools and methods) are reserved for the chapter after this one.

8.1 Experiment Flow

The flow of experiments performed in this thesis is shown in Figure 34. The flow consists of three main parts: (i) simulation, (ii) WCET analysis and (iii) WCRT analysis. The research presented in this thesis focuses on WCRT analysis, as shown in the shadowed block on the right-hand-side of Figure 34. In this section, we explain each part in the flow.

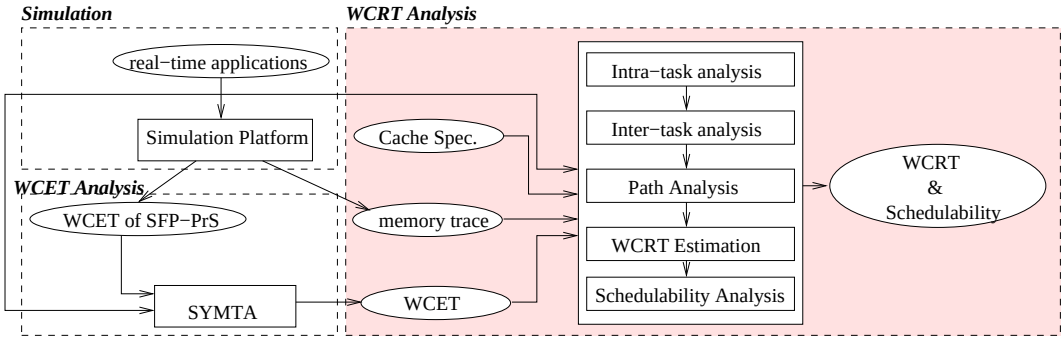


Figure 34: Experiment flow

8.1.1 Simulation

The objectives of simulation are (i) finding all memory addresses that can possibly accessed by a task and (ii) finding the WCET for each SFP-PrS in a task.

As mentioned in Section 2.3, a task consists of SFP-PrS nodes in a PCFG. An SFP-PrS behaves as a basic block. Since there is only one feasible path in an SFP-PrS, we can use simulation to find the memory footprint and the WCET of an SFP-PrS. Note that we assume that there are no dynamic memory allocations and that addresses of all data structures are fixed. Because branches exist in a task, we need to use different input data sets to make sure all SFP-PrSes are simulated. This is explained in Example 40.

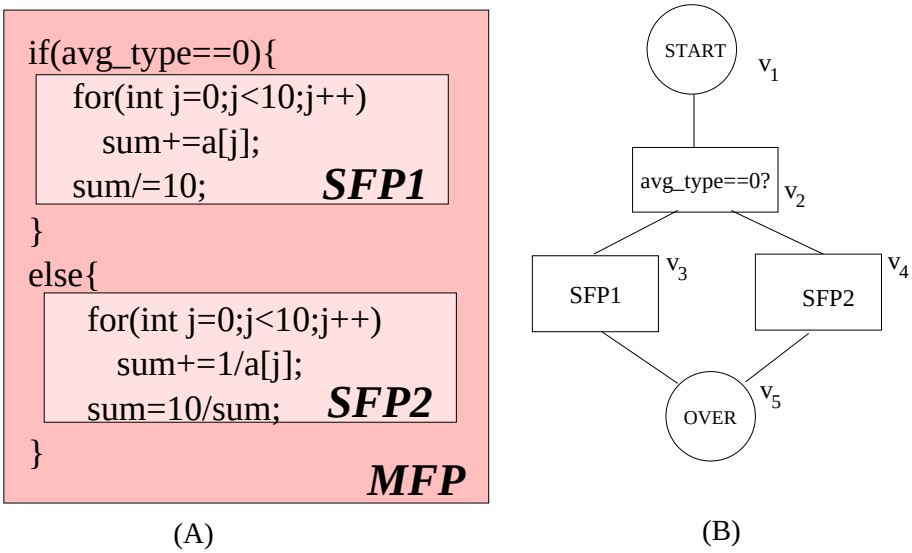


Figure 35: A Example of Simulation

Example 40: Consider the program shown in Figure 35. The value of *avg_type* determines which SFP-PrS to execute next. In order to find the WCET and memory footprint of SFP-PrS v_3, we first simulate the program with *avg_type* equal to 0. Then we simulate the program once again with *avg_type* equal to 1 to find the WCET and memory footprint of SFP-PrS v_4. Therefore, we need to simulate the program at lease twice in this case. □

Usually, a task has to be simulated more than once to collect WCETs and memory

footprints of all SFP-PrSes. WCETs of all SFP-PrSes in the task under consideration are then fed to SYMTA for WCET analysis.

8.1.2 WCET Analysis (SYMTA)

WCETs of tasks are required in our WCRT analysis. We use SYMTA for WCET analysis. SYMTA takes as input the program structure of a task and the WCETs of each SFP-PrS in the task. Based on these inputs, SYMTA builds ILP equations, which are then solved to find the WCET of the task. We introduced SYMTA briefly in Section 3.2. A detailed description of SYMTA can be found in [68, 70, 71, 72]. Note that we can also use other WCET analysis approaches to obtain WCETs of tasks. We only need to replace the WCET estimates in our WCRT analysis approach.

8.1.3 WCRT Analysis

Our WCRT analysis requires four inputs: (i) program structures of tasks, (ii) cache specification, (iii) WCETs of tasks and (iv) memory footprints of tasks. Cache specification gives the parameters of a cache which consists of cache size, the number of ways, the number of bytes in each cache line and the type (i.e., set-associative or prioritized).

First, we apply our intra-task cache analysis approach to calculate the MUMBS for every task except the task with the highest priority. Second, we use inter-task cache eviction analysis approach as proposed in this thesis to analyze cache interference between every pair of tasks with different priorities. Path analysis is exploited to tighten the analysis results. After these two steps, we can derive the CRPD estimate for every type of preemption. Next, we export the CRPD estimates to our WCRT analysis equation as shown in Section 6.3.1 to find the WCRT for each task. Based on the WCRT estimates, we can find if a feasible schedule can be found or not.

8.2 Experiment Environment

We develop some real-time applications to evaluate the performance of our WCRT analysis algorithm and our prioritized cache. The experiments are performed on a uniprocessor system with a unified L1 cache. The experimental environment is shown in Figure 36.

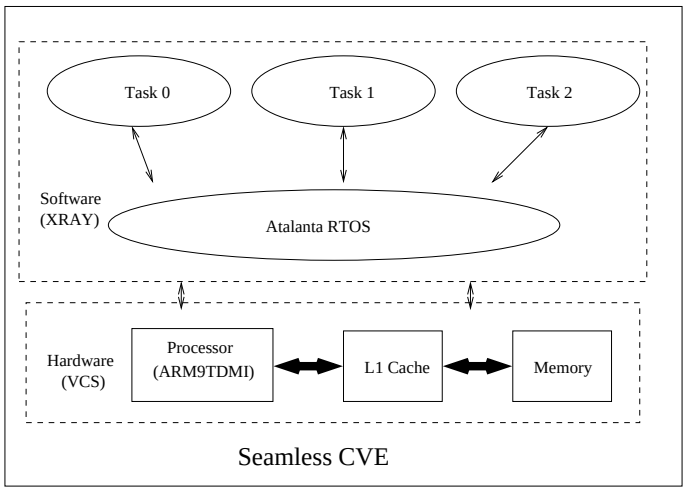


Figure 36: Simulation Architecture

The experiments require support of both software and hardware. Software and hardware setup are illustrated in this chapter respectively.

8.2.1 Software Setup

Our experiments are based on three groups of tasks. The first group of tasks, OFDM, ED and MR, are derived from a Mobile Robot application as described in Example 1. The tasks in the second group are Adaptive Differential Pulse Code Modulation Coder (ADPCMC), ADPCM Decoder (ADPCMD) and Inverse Discrete Cosine Transform (IDCT). ADPCMC and ADPCMD are taken from MediaBench [23, 39]. IDCT is extracted from an MPEG2 decoder. The third group of tasks are obtained from a paper of Lee et al. [19] including four tasks, FFT, LU Decomposition (LUD), Least Mean Square (LMS) and Finite Impulse Response filter (FIR).

An RTOS, Atalanta, developed at Georgia Tech, runs on the processor [53]. Atalanta provides multi-tasking supports with a Fixed Priority Scheduling algorithm. We assign task

priorities statically. Each task has a unique priority.

All software including the RTOS and applications are supported by XRAY [42], which is a debugger and an Instruction Set Simulator (ISS) provided by Mentor Graphics [40].

8.2.2 Hardware Setup

We build a simulation platform by using an ARM9TDMI processor core with a unified L1 cache. The L1 cache can be a set associative cache or a prioritized cache. The cache size is 32KB. The main memory size is 256MB. It turns out that none of our examples require more than 256MB, so we use only this simple two-level memory hierarchy. The ARM9 processor is simulated with ARM9 Processor Simulation Package (PSP) developed by Mentor Graphics. The caches (i.e., the set associative and the prioritized cache) are modeled with the Verilog Hardware Description Language (HDL). All hardware units including the processor, the cache and the memory are simulated with VCS [55], a Verilog simulator developed by Synopsys [54].

Hardware and software are co-simulated with Seamless CVE [41], which is a hardware/software co-simulation tool from Mentor Graphics [40].

8.3 Summary

In this chapter, we first illustrate the flow for experiments in Section 8.1. Then we explain the simulation environment used in experiments in Section 8.2. In Section 8.2, we also briefly introduce the tasks used in our experiments.

In the next chapter, Chapter 9, we elaborate our experiments and give experimental results plus analysis of those results.

CHAPTER IX

EXPERIMENTAL RESULTS

In this chapter, we develop some real-time applications to evaluate the performance our WCRT analysis approach and our prioritized cache. The experiments for the WCRT analysis approach are shown in Section 9.1. The experiments for the prioritized cache are explained in Section 9.2. Finally, Section 9.3 summarizes this experimental results chapter.

9.1 Experiments for the WCRT Analysis

We use three applications as described in Section 8.2.1 to evaluate the performance of our WCRT estimate approach. In addition, we use a fourth application provided in several publications by Lee et al. [19, 20, 21, 22]. In the experiments, we compare five approaches to estimate WCRT. All approaches take CRPD into consideration.

Approach 1 (A1): This is the approach proposed by Busquests-Mataix [4]. In this approach, all cache lines used by preempting tasks are assumed to be reloaded for a preemption.

Approach 2 (A2): Only lines in the intersection set of lines used by the preempting task and the preempted task are assumed to be reloaded after a preemption. Our inter-task cache eviction method proposed in 5.1 is used here.

Approach 3 (A3): Only useful memory blocks in the preempted task are used to estimate the CRPD. Intra-task cache access analysis for the preempted task proposed by Lee et al. in [19, 20] is used here.

Approach 4 (A4): Both inter-task cache eviction analysis and intra-task cache access analysis are used to estimate the cache reload cost. Path analysis is not used in this approach. Note that this approach is used by Lee et al. in [21, 22] to estimate the CRPD for

each preemption. ILPs as proposed in Lee's approach are constructed to estimate to WCRT in Approach 4.

Approach 5 (A5): Both inter-task cache eviction analysis and intra-task cache access analysis are used to estimate the cache reload cost. Also, path analysis proposed in Section 5.4 is applied to the preempting task. This approach – Approach 5 – is the WI^3 approach described in this thesis and summarized in Section 6.3.

9.1.1 Experiment I: a Mobile Robot Application

The tasks in the first experiment, OFDM, ED and MR, are described in Example 1. Table 4 lists the WCET estimate, period/deadline and priority for each task.

Table 4: Tasks

Task	WCET(us)	Period(us)	Priority
T_1(MR)	842	3,500	2
T_2(ED)	1892	6,500	3
T_3(OFDM)	2830	40,000	4

As shown in Table 4, MR has the highest priority and OFDM has the lowest priority. Three types of preemptions can happen in this application, MR preempting ED, ED preempting OFDM and MR preempting OFDM.

The estimates of the number of cache lines to be reloaded in each type of preemption derived with these five approaches are listed in Table 5.

Table 5: Number of cache lines to be reloaded

Preemptions	A1	A2	A3	A4	A5
OFDM by MR	245	134	187	118	88
OFDM by ED	254	172	187	135	98
ED by MR	245	87	106	85	81

Approach 1 assumes that all cache lines used by the preempting task will be accessed by the preempted task after the preempted task is resumed. Obviously, this may not be true. Some cache lines will never be used by the preempted task no matter which path the preempted task takes. Thus, by calculating the set of cache lines that can possibly be

accessed by both the preempting and the preempted task, we can further reduce the estimate of the number of cache lines to be reloaded by the preempted task, as shown in Approach 2.

Approach 3 calculates the maximum set of memory blocks in the preempted task that can potentially cause cache reload. This approach only relates to the memory access pattern of the preempted task. Thus, for a certain preempted task, the estimate of cache reload cost is always the same. Obviously, this approach ignores the differences among preempting tasks and only assumes that all "useful" memory blocks in the preempted task will be evicted by the preempting task, which might not be true.

Both inter- and intra-task cache eviction are considered in Approach 4. By considering the preempting tasks and incorporating inter-task cache eviction analysis, the estimate of the number of cache lines that need to be reloaded is significantly reduced, as shown in Table 5.

In Approach 5, we further use path analysis to tighten CRPD estimates. As shown in Table 5, the estimates of the number of cache conflicts are reduced the most in Approach 5.

The WCRT of OFDM and ED can be calculated based on the results shown in Table 5. Notice that MR has the highest priority so that it can never be preempted. So, the WCRT of MR is just equal to its WCET. We also vary C_{miss} from 10 cycles to 40 cycles to investigate the influence of cache miss penalty on the WCRT. The estimate results (Approach 1 thorough Approach 5) and the Actual Response Times (ART) which is the WCRT as observed in simulations are listed in Table 6. To calculate ART via simulation, we instrumented data values to maximize execution time (e.g., by iterating over maximum numbers of loops). However, please note that we did not instrument worst-case task arrivals (e.g., to maximize nested preemptions) in our ART calculation.

Approach 4 is the best approach, in this thesis author's opinion, of all prior work known to the thesis author. As we can see from Table 6, when the cache penalty is small (we vary from 10 to 80 cycles), CRPD does not have a large impact task WCRT. Thus, our WI^3 approach (A5) does not show much difference from the approach of Lee et al. (A4). However,

Table 6: Comparison of WCRT estimate in Experiment I

C_{miss}	Task	A1	A2	A3	A4	A5	ART
10	OFDM	9847	9771	9789	9764	9684	6113
	ED	2567	2409	2428	2407	2403	2382
20	OFDM	12510	12242	12378	10424	10264	6211
	ED	2812	2496	2534	2492	2484	2400
30	OFDM	23501	19249	17244	12468	12258	6255
	ED	3057	2583	2640	2577	2565	2426
40	OFDM	45216	31284	30532	16952	12966	6362
	ED	3302	2670	2746	2662	2646	2525

when the cache miss penalty is large enough, CRPD cannot be ignored in WCRT estimation. In this case, a tighter CRPD estimate can reduce the WCRT estimate significantly. For example, when the cache miss penalty is 40, our approach tightens the WCRT of OFDM by 24%.

In order to investigate the impact of CRPD on the WCRT estimate, we change the cache miss penalty from 10 cycles to 80 cycles and compare the WCRT estimates of OFDM derived with A4 and A5 respectively. The result is shown in Figure 37.

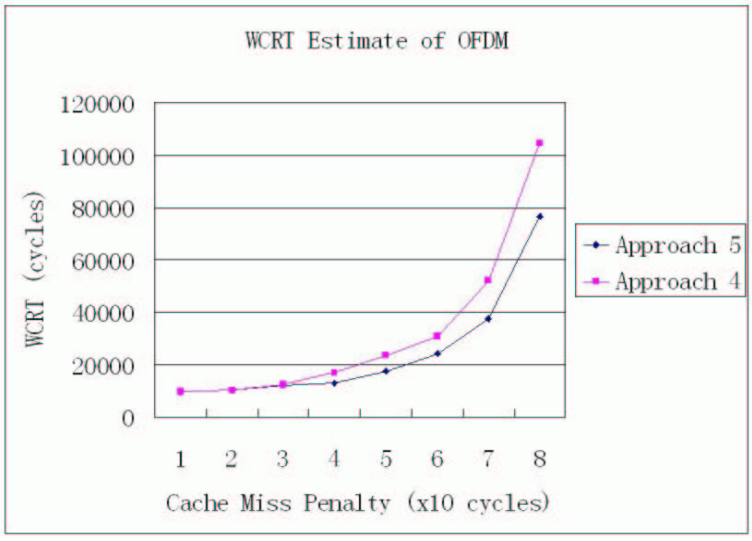


Figure 37: Comparison of the WCRT estimates derived with Approach 4 and Approach 5

As shown in Figure 37, when the cache miss penalty is small, there is not much difference between Approach 5 and Approach 4. However, when the cache miss penalty becomes

larger, our approach (Approach 5) performs better than Approach 4. When the cache miss penalty is 80 clock cycles, the WCRT estimate derived with Approach 5 is 28% less than the WCRT estimate derived from Approach 4.

Another reason that our approach (A5) outperforms the approach of Lee et al. is that our approach has a tighter estimate of the number of preemptions. We tie each preemption to an invocation of a preempting task. Instead, Lee et al. use ILP to estimate the number of preemptions. Without exploiting all constraints to build ILP equations, there may be overestimate in the number of preemptions. For example, when the cache penalty is 80 cycles, OFDM is preempted by MR 29 times as estimated with Approach 4. However, the number of times MR preempts OFDM is only 21 if estimated with Approach 5. The WCRT estimate is tightened in our approach due to reduction in the upper bound estimate of the number of preemptions.

9.1.2 Experiment II: a DSP Application

The tasks in the second experiment are Adaptive Differential Pulse Code Modulation Coder (ADPCMC), ADPCM Decoder (ADPCMD) and Inverse Discrete Cosine Transform (IDCT). ADPCMC and ADPCMD are taken from MediaBench [23, 39]. IDCT is extracted from MPEG2 decoder. The periods, priorities and WCETs of tasks in each experiment are listed in Table 7.

Table 7: Tasks

Task	WCET(us)	Period(us)	Priority
T_1(IDCT)	1580	4,500	2
T_2(ADPCMD)	2839	10,000	3
T_3(ADPCMC)	7675	50,000	4

Similar to Experiment I, there are three types of preemptions, IDCT preempting ADPCMD, IDCT preempting ADPCMC and ADPCMD preempting ADPCMC. The estimates of the number of cache lines to be reloaded in each type of preemption derived with these five approaches are listed in Table 8.

Table 8: Number of cache lines to be reloaded for each preemption in Experiment II

Preemptions	A1	A2	A3	A4	A5
ADPCMC by IDCT	249	68	98	64	56
ADPCMC by ADPCMD	220	114	98	92	64
ADPCMD by IDCT	183	58	89	55	46

Based on Table 8, we can estimate CRPD and use CRPD to derive WCRT estimate for each task. The WCRT estimates of the second experiment are listed in Table 9.

Table 9: Comparison of WCRT estimates for tasks in Experiment II

C_{miss}	Task	A1	A2	A3	A4	A5	ART
10	ADPCMC	35743	35701	35071	35027	34676	23512
	ADPCMD	6565	6315	6377	6309	6291	6190
20	ADPCMC	48528	38687	37987	35983	34967	23867
	ADPCMD	6931	6431	6555	6419	6383	6223
30	ADPCMC	88606	39555	39055	38911	38779	24101
	ADPCMD	7297	6547	6733	6529	6475	6278
40	ADPCMC	359239	48714	47722	39931	39755	24353
	ADPCMD	7663	6663	6911	6639	6567	6354

Because the number of cache lines to be reloaded for each preemption in this application is small, the CRPD is not large enough to affect the WCRT when the cache miss penalty is small. As shown in Table 9, although our approach (A5) shows a slightly better performance than the approach of Lee et al., the difference is not large. However, if the cache miss penalty increases, our approach significantly outperforms the approach of Lee et al. in estimating the WCRT of ADPCMC, as shown in Figure 38. The cache miss penalty is changed from 10 cycles to 80 cycles.

9.1.3 Experiment III: a Task Set with Six Tasks

Each application in Experiment I and Experiment II has only three tasks. The preemption scenarios are relatively simple. In this experiment, we use a lager number of tasks to investigate the performance of our WCRT analysis approach.

The third experiment contains six tasks, OFDM, ADPCMC, ADPCMD, IDCT, ED and MR. The priority and period of each task is listed in Table 10. Note that, in order to

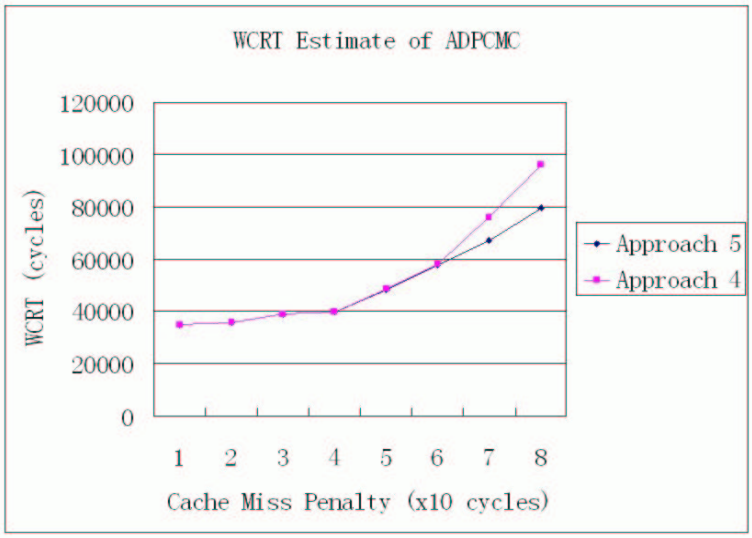


Figure 38: Comparison of the WCRT estimates of ADPCMC derived with Approach 4 and Approach 5

satisfy the necessary condition of schedulability of a real-time system (i.e., the total CPU utilization of all tasks must be less than 100% [28, 34]), we increase the periods of some tasks as compared to the same tasks in Experiment I and II. ADPCMC has the lowest priority and MR has the highest prority. The WCET of each task remains the same.

Table 10: Tasks in Experiment III

	T_1(MR)	T_2(IDCT)	T_3(ED)	T_4(ADPCMD)	T_5(OFDM)	T_6(ADPCMC)
Period(us)	7,000	9,000	13,000	20,000	40,000	50,000
Priority	2	3	4	5	6	7
WCET(us)	830	1580	1392	2839	2830	7675

We use the five different approaches described earlier to estimate the WCRT of the two tasks with the lowest priorities, OFDM and ADPCMC, which may be preempted more frequently than other tasks. Table 11 gives the WCRT estimates of OFDM and ADPCMC with different approaches.

Approach 5 and Approach 4 are compared in Table 12. By applying path analysis, the WCRT estimate is reduced by up to 32%. Thus, we demonstrate that our approach can tighten WCRT estimation results significantly by using path analysis technique, which is missing in the enhanced approach of Lee et al.[20]. Also, our WI^3 approach (A5) achieves

Table 11: WCRT estimates in Experiment III

WCRT estimates of ADPCMC					
C_{miss}	A1	A2	A3	A4	A5
10	51434	34163	34591	33893	33507
20	75201	51452	57850	38431	34685
30	232903	59482	74020	58099	38905
40	4070285	75073	114209	69495	58142
WCRT estimates of OFDM					
C_{miss}	A1	A2	A3	A4	A5
10	16901	16551	17050	16496	16330
20	25904	17199	17242	17001	16757
30	50831	17847	17750	17699	17184
40	116464	34694	27718	25615	17611

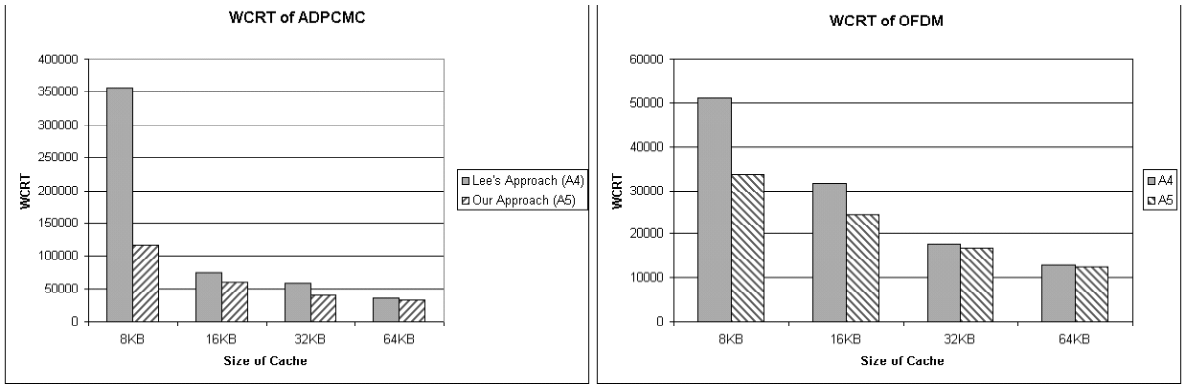
Table 12: Comparison of Approach 4 and Approach 5 for WCRT estimates

	C_{miss}			
Task	10	20	30	40
ADPCMC	1%	10%	32%	16%
OFDM	18%	3%	4%	31%

a tighter WCRT estimate because the estimate of the number of preemptions is tightened. Our WCRT estimate approach (Approach 5) binds each preemption with an invocation of a preempting task. As a comparison, Lee et al. use ILP to estimate of the number of preemptions. As we point out earlier, incomplete ILP constraints may cause overestimate in the number of preemptions, which increases the WCRT estimate.

We also run this experiment on caches with different sizes. Figure 39(A) and Figure 39(B) show the WCRT of ADPCMC and OFDM, respectively, when cache size changes from 8KB to 64KB. The cache miss penalty is 30 clock cycles here.

Usually, when the cache size is small, task WCRT is more sensitive to cache interference among tasks. In this case, the difference between our approach and Lee's approach is larger. For example, as we can see in Figure 39, when the cache size is 8KB, the WCRT of ADPCMC is reduced by 68% when comparing Approach 5 with Approach 4. However, when the cache size increases, the cache conflicts among tasks lessen; thus, for large caches, WCRT may not be heavily affected by cache interference. The difference between



(A) (B)

Figure 39: Comparison of WCRT with Different Cache Size

our approach and the approach of Lee et al. (Approach 4) also becomes smaller as cache size becomes larger. For example, when the cache size is 64KB, WCRT is deduced by only 4% as comparing Approach 5 with Approach 4.

From the comparison results as shown in Figure 39, our WCRT approach can reduce the WCRT estimates by from 4% to 68%, as compared to the approach of Lee et al. When the cache size increases, the conflicts among tasks occured in the cache become less. Thus, WCRT is not affected heavily by cache interference. The difference between our approach and Lee's approach also becomes smaller as cache size becomes larger.

9.1.4 Experiment IV: a Task Set Used in the Work of Lee et al.

Two factors affect the accuracy of WCRT estimates: (i) the estimate of the number of preemptions and (ii) the estimate of cache reload cost for each preemption. The approach of Lee et al. explores all preemption scenarios in order to tight the cache reload cost estimate for each preemption. They use ILP equations to estimate the number of preemptions. However, the estimate of the number of preemptions by using ILP depends on the constraints exploited in ILP equations. ILP equations without complete constraints may give an overestimate of the number of preemptions. As a comparison, we tie every CRPD to an invocation of a preempting task directly. Thus, each preemption is actually invoked by a preempting task. We can expect a tight estimate of the number of preemptions.

We execute one more experiment to show the effect of the estimate of the number of preemptions. For example, consider the following scenario based on the incomplete description of task specification of the experiment in Lee et al. [21] (the description is incomplete because we do not know have any information about the program structures and memory footprints of these tasks).

Four tasks as listed in Table 13 are used in the experiment in [21]. When the cache reload penalty is 100 cycles, the WCRT of FIR (i.e., the task with the lowest priority) given by the approach of Lee et al. is 5,323,620 cycles. However, the WCRT estimate resulting from the iteration we proposed in Section 6.2.3 is 3,778,075 cycles, which shows a reduction of 29%. Note that we use the preemption related cache reload cost as reported in [21]. Since we use the same cache reload cost for each preemption, the difference in WCRT estimate is caused by the the number of preemptions used in WCRT estimation. Apparently, the approach of Lee et al. overestimates the number of preemptions. Also, note that the cache reload cost here is derived without applying path analysis on the preempting task. We can expect additional reduction in WCRT estimation results if the path analysis technique proposed in our approach were to be used.

Table 13: Tasks in the paper of Lee et al. [20]

Task	Period	WCET
FFT	$320,000$	$60,234 + 280 \times C_{miss}$
LUD	$1,120,000$	$255,998 + 364 \times C_{miss}$
LMS	$1,920,000$	$365,893 + 474 \times C_{miss}$
FIR	$25,600,000$	$557,589 + 405 \times C_{miss}$

9.2 Experiments for a Prioritized Cache

We present a prioritized cache in this thesis. Our prioritized cache is a variant of a set associative cache. The cache is partitioned at the granularity of columns. After the cache partitions are allocated to tasks, the prioritized cache behaves as a group of small size set associative caches. Thus, the WCRT analysis approach designed for conventional set

associative caches can be adapted easily to analyze the behavior of the prioritized cache. In this section, we estimate the WCRT of tasks in three applications running with a prioritized cache. The results are compared with a set associative cache of the same size.

9.2.1 Experiment I: a Mobile Robot Application

In this experiment, we use the same tasks as in Experiment I in Section 9.1. As stated in Chapter 7, the prioritized cache usage is divided into two stages, the transit stage and the stable stage. For many real-time applications, the performance of the prioritized cache is mainly determined by the stable stage. Thus, in our experiments, we only investigate the performance of the prioritized cache in the stable stage.

Table 14 compares the WCET of each task in the set associative cache and the prioritized cache.

Table 14: WCET with different caches

Tasks	MR	ED	OFDM
WCET in SA	842	1892	2830
WCET in PC	626	1676	4210

Three types of preemptions can happen in this system, MR preempting ED, MR preempting OFDM and ED preempting OFDM. The number of cache lines to be reloaded in these three preemptions are estimated in Table 15. Table 15 shows that no cache conflicts occur between ED and MR in the prioritized cache. This is also true for OFDM and MR. This means MR is assigned a partition of the prioritized cache exclusively. Because MR has the highest priority, OFDM and ED cannot use the cache assigned to MR. Thus, there are no cache conflicts among MR and other tasks (i.e., ED and OFDM). In this experiment, three columns are assigned to MR exclusively. It turns out that all cache lines required by MR fit into these columns. MR uses 80% of the SRAM available in these three cache columns. The other three non-shared columns are assigned to ED exclusively. ED also uses the two shared columns, which are also used by OFDM. Thus, there are cache conflicts between ED and OFDM. ED uses over 90% of the cache columns which ED can access (i.e.,

the three columns owned by ED and the two shared columns). OFDM uses 100% of the shared columns.

Table 15: Estimate of cache lines to be reloaded

Preemptions	ED by MR	OFDM by MR	OFDM by ED
In SA	81	88	98
In PC	0	0	160

Based on the WCET and the number of cache lines to be reloaded, we can apply the WCRT approach as proposed in [57, 56]. Because the impact of cache on the WCRT depends on not only the number of cache conflicts but also the cache miss penalty, we change the cache miss penalty from 10 cycles to 40 cycles. The comparison of WCRT of OFDM running with the set associative cache and the prioritized cache is shown in Table 16.

Table 16: WCRT of OFDM with different caches

C_{miss}	10	20	30	40
WCRT in PC	10260	11306	11626	11946
WCRT in SA	9684	10264	12558	12966

Two facts in these experimental results show the advantages of the prioritized cache as compared to the conventional cache. First, non-preemption related cache reload costs in high priority tasks are reduced. High priority tasks such as MR do not share any cache resources with other tasks; thus, we do not need to assume a cold cache for WCET analysis of these tasks after the first execution of the task upon startup/reboot. As a result, the WCET estimates of high priority tasks are tightened. For example, the WCET of MR – which is never preempted since MR has the highest priority – is reduced by 26% according to the WCET estimate results in Table 14. In short, by using the prioritized cache, non-preemption related cache reload costs of high priority tasks are reduced. However, as we notice, the WCET of the low priority task, OFDM, is extended significantly because OFDM is restricted to use a limited portion of the cache; in our case, OFDM can only use the shared columns of the hot cache (i.e., after initial runs of tasks due to startup/reboot).

Second, the prioritized cache can also tighten the WCRT estimates of tasks because preemption-related cache reload overhead is minimized. As we can see from Table 15, there are no cache conflicts between ED and MR; neither are there any cache conflicts between OFDM and MR. Thus, CRPD caused by MR preempting ED and CRPD caused by MR preempting OFDM are both zero. However, both ED and OFDM use the shared columns. Thus, there is still CRPD caused by ED preempting OFDM.

9.2.2 Experiment II: a DSP Application

The same tasks in Experiment II of Section 9.1 are used in this experiment. The WCETs of tasks running with the prioritized cache and the set associative are compared in Table 17.

Table 17: Tasks in Experiment II

Tasks	IDCT	ADPCMD	ADPCMC
Periods(us)	4500	10000	50000
Priority	2	3	4
WCET in SA	1580	2839	7675
WCET in PC	1498	2830	11182

As shown in Table 17, the WCET of high priority task is reduced as well, as a result of eliminating cache sharing. For instance, the WCET of IDCT is reduced by 5%. The number of cache lines to be reloaded in each cache is listed in Table 18.

Table 18: Estimate of cache lines to be reloaded

Preemptions	ADPCMD	ADPCMC	ADPCMC
	by IDCT	by ADPCMD	by IDCT
In SA	46	64	56
In PC	0	0	0

Table 19: WCRT of ADPCMC with different caches

C_{miss}	10	20	30	40
WCRT in PC	35686	35873	36001	36349
WCRT in SA	34676	34967	38779	39775

The number of cache lines to be reloaded in each cache is listed in Table 18. Recall, as stated earlier, that we use caches with eight “ways” or “columns.” In this experiment, IDCT

uses two columns. ADPCMD uses three columns and ADPCMC uses the remaining three columns available, two of which the user preset as shared. IDCT uses 70% of the available memory in the two cache columns used by IDCT. ADPCMD and ADPCMC use 90% of the memory available in the cache columns each uses. In this application, there are no cache conflicts among tasks in the prioritized cache. This means both IDCT and ADPCMD use cache columns exclusively and do not require any shared columns. Only ADPCMC uses shared columns. The overall result is that there are no cache conflicts among these three tasks. The WCRT estimate of ADPCMC is shown in Table 19.

In this application, because all inter-task cache conflicts in the prioritized are eliminated, the preemption-related cache reload cost is zero. Thus, the WCRT of ADPCMC is not affected by cache miss penalty. As it turns out, when the cache miss penalty is big enough so that the preemption-related cache reload cost cannot be ignored in the conventional cache, the prioritized cache shows better performance in the WCRT even of low priority tasks. For example, when the cache miss penalty is 40, the WCRT of ADPCMC with the prioritized cache is reduced by 8% as compared to an equivalent set-associative cache.

9.2.3 Experiment III: a Task Set with Six Tasks

The third experiment contains six tasks, OFDM, ADPCMC, ADPCMD, IDCT, ED and MR, which are the same as tasks in Experiment III in Section 9.1. The priority and period of each task is listed in Table 10. ADPCMC has the lowest priority and MR has the highest priority.

In this experiment, we set the cache miss penalty to 30 clock cycles. Figure 40 compares the WCRT of each task with a set associative cache and a prioritized cache.

Apparently, by using a prioritized cache, the WCRT of high priority tasks can be reduced because high priority tasks are allocated cache columns exclusively. On the contrary,

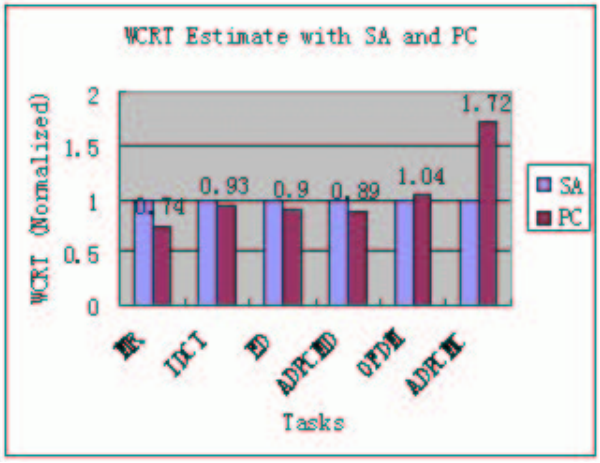


Figure 40: Comparison of task WCRT with a SA and a PC

low priority tasks have to use shared columns. Thus, a prioritized cache improves the performance of high priority tasks at the cost of the performance of low priority tasks. As shown in Figure 40, the WCRTs of MR, IDCT and ED are reduced by between 7% and 26%. However, the WCRT of ADPCMC is increased by nearly 70%.

9.3 Summary

In this chapter, we use some applications to verify the performance of our WCRT analysis approach and the prioritized cache. The experimental results demonstrate that the WCRT estimate can be tightened significantly by using our WCRT analysis approach. When using a 32KB, four way set-associative cache, we can achieve a reduction up to 32% in WCRT estimate in our experiments. When the cache size is small – say, 8KB – WCRT estimates can be reduced by up to 68%. Furthermore, the prioritized cache reduce or eliminate inter-task cache interference. A reduction up to 26% can be achieved in WCRT estimates by using a prioritized cache in our experiments. The next chapter concludes this thesis.

CHAPTER X

CONCLUSION

This thesis presents our work in cache related WCRT analysis for preemptive multi-tasking real-time systems. A novel WCRT analysis approach is proposed. The impact of cache behavior is considered in our approach. In order to reduce cache interference among tasks, we design a prioritized cache. A prioritized cache has a more predictable cache behavior in multi-tasking systems.

First, we propose a cache-related WCRT analysis approach. Cache Relate Preemption Delay (CRPD) is considered in our approach. A new algorithm based on Cache Index Induced Partition (CIIP) is presented to calculate the interference among tasks in the cache. CIIP gives an abstract mapping from the memory to the cache without requiring knowledge of the replacement algorithms used in the cache. The method can be uniformly applied to the all types of caches (e.g., direct-mapped, set associative and full associative caches), although we focus our efforts on unified, set associative caches. Inter-task cache eviction analysis is then combined with useful memory block analysis of the preempted task as proposed by Lee et al. [19, 20, 21, 22]. By calculating the intersection of cache lines used by the preempting task and the preempted task, the estimate of CRPD is tightened. Moreover, we apply path analysis to the preempting task. Path analysis reduces the CRPD estimate further. Based on the CRPD analysis, we propose a new WCRT estimate formula. We call our overall analysis approach *WCRT Integrating Inter- and Intra-task cache timing analysis approach* or WI^3 for short. WI^3 utilizes significant portions of SYMTA [68, 70, 71, 72] and Lee et al. [19, 20, 21, 22]. The WI^3 analysis approach can be used to analyze task WCRT in a multi-tasking preemptive real-time system and give a tighter WCRT estimate because of exploiting new techniques such as CIIP and exploiting known techniques such as path

analysis in a novel way.

We use four groups of tasks to evaluate the performance of our WCRT analysis approach. The tasks are derived from typical embedded systems such as mobile robot control systems and DSP applications. The experiment shows that for a 32KB four way set associative cache, our approach can reduce the estimate of WCRT by up to 32% when compared with prior approaches.

The formal WCRT analysis allows designers to evaluate trade-offs in cache design when designers are focused on the effect of cache type selection on WCET/WCRT estimation. For example, in multi-tasking systems, WCRT is worsened by inter-task cache interference. In order to reduce inter-task cache interference, we design a prioritized cache. In a prioritized cache, the cache is partitioned at the granularity of columns. Cache partitions are then assigned to tasks according to the task priorities. Inter-task cache interference is avoided except in the shared columns. We also apply our WCRT analysis approach to analyze the behavior of the prioritized cache. A formal WCRT analysis for the prioritized cache enables the prioritized cache to be used safely in a real-time system. Two applications are used to compare the performance of the prioritized cache and the set associative cache. The experimental results demonstrate a reduction of up to 26% in WCRT estimate by using our prioritized cache versus a set-associative cache of equal size and associativity.

In conclusion, we make five major contributions in this thesis, which are listed as below.

Contribution 1: A novel approach is proposed to analyze inter-task cache interference.

Contribution 2: Inter-task cache eviction analysis is integrated with intra-task cache eviction analysis.

Contribution 3: Path analysis is used to improve cache interference analysis.

Contribution 4: A new WCRT estimate formula is proposed.

Contribution 5: A novel "prioritized cache" design is presented to reduce CRPD.

REFERENCES

[1] AKGUL, B. E. S. and MOONEY, V. J., "The system-on-a-chip lock cache," *International Journal of Design Automation for Embedded Systems*, vol. 7, pp. 139–174, September 2002.

[2] ALT, M., FERDINAND, C., MARTIN, F., and WILHELM, R., "Cache behavior prediction by abstract interpretation," *Proceedings of Static Analysis Symposium (SAS'96)*, pp. 52–66, September 1996.

[3] BALASUNDARAM, A., PEREIRA, A., PARK, J., and MOONEY, V., "Golay and wavelet error control codes in vlsi," *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'04)*, pp. 563–564, January 2004.

[4] BUSQUETS-MATAIX, J., SERRANO, J., ORS, R., GIL, P., and WELLINGS, A., "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," *Real-Time Technology and Applications Symposium*, pp. 204–212, June 1996.

[5] CHIOU, D., JAIN, P., DEVADAS, S., and RUDOLPH, L., "Application specific memory management for embedded systems using software-controlled caches," *Proceedings of the 37th Design Automation Conference*, pp. 416–419, June 2000.

[6] Definition of Partition, http://www.nist.gov/dads/HTML/partition.html.

[7] DROPSO, S., "RISC processor worst-case execution time penalties," Tech. Rep. TR-95-110, University of Massachusetts, Amherst, October 1995.

[8] DROPSO, S., "Comparing caching techniques for multitasking real-time systems," Tech. Rep. UM-CS-1997-065, University of Massachusetts, Amherst, November 1997.

[9] ERMERAHL, A., STAPPERT, F., and ENGBLOM, J., "Clustered calculation of worst-case execution times," *Proceedings of Compilers, Architecture and Synthesis for Embedded Systems*, pp. 51–62, October 2003.

[10] FERDINAND, C., HECKMANN, R., LANGENBACH, M., MARTIN, F., SCHMIDT, M., THEILING, H., THESING, S., and WILHELM, R., "Reliable and precise WCET determination for a real-life processor," *Proceedings of the First International Workshop on Embedded Software, (EMSOFT 2001) Volume 2211 of LNCS, Springer-Verlag (2001)*, pp. 469–485, 2001.

[11] HENNESSY, J. and PATTERSON, D., *Computer Architecture: A Quantitative Approach (3rd edition)*. Morgan Kaufmann Publishers, Menlo Park, CA, 2002.

[12] Intel IXP2800 Network Processor Hardware Reference Manual, Intel Corporation, 2004, Available HTTP: http://www.intel.com/network/manuals/.

[13] JAIN, P., DEVADAS, S., ENGELS, D., and RUDOLPH, L., "Software-assisted cache replacement mechanisms for embedded systems," in *Proceedings of the Int'l Conference on Computer-Aided Design*, pp. 119–126, November 2001.

[14] JOSEPH, M. and PANDYA, P., "Finding response times in a real-time system," *BCS Computer Journal*, vol. 29, pp. 390–395, October 1986.

[15] JUAN, T., ROYO, D., and NAVARRO, J., "Dynamic cache splitting," *Proceedings of the XV International Conference of the Chilean Computer Society*, November 1995.

[16] KENNETH, A., and CHARLES, W., *Discrete Mathematics (5th edition)*. McGraw Hill, Boston, MA, 2004.

[17] KIRK, D., "SMART (Strategic Memory Allocation for Real-Time) cache design," *Proceedings of IEEE 10th Real-Time System Symposium*, pp. 229–237, December 1989.

[18] KIRK, D., STROSNIDER, J., and J.SASINOWSKI, "Allocating SMART cache segments for schedulability," *Proceedings of Euromicro '91 Workshop on Real Time Systems*, pp. 41–50, June 1991.

[19] LEE, C., HAHN, J., SEO, Y.-M., MIN, S., HA, R., HONG, S., PARK, C., LEE, M., and KIM, C., "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *Proceedings of the Seventeenth IEEE Real-Time Systems Symposium*, pp. 264–274, December 1996.

[20] LEE, C., HAHN, J., SEO, Y.-M., MIN, S., HA, R., HONG, S., PARK, C., LEE, M., and KIM, C., "Bounding cache-related preemption delay for real-time systems," Tech. Rep. SNU-CE-AN-97-002, Department of Computer Engineering, Seoul National University, November 1997.

[21] LEE, C., HAHN, J., SEO, Y., MIN, S., HA, R., HONG, S., PARK, C., LEE, M., and KIM, C., "Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *Proceedings of IEEE Real-Time Systems Symposium*, pp. 187–198, December 1997.

[22] LEE, C., HAHN, J., SEO, Y., MIN, S., HA, R., HONG, S., PARK, C., LEE, M., and KIM, C., "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700–713, 1998.

[23] LEE, C., POTKONJAK, M., and MANGIONE-SMITH, W., "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," *Proceedings of International Symposium on Microarchitecture*, pp. 330–335, December 1997.

[24] LEE, J. and MOONEY, V., "A novel deadlock avoidance algorithm and its hardware implementation," *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS'04)*, pp. 200–205, September 2004.

[25] LEE, J. and MOONEY, V., "Hardware/software partitioning of operating systems: Focus on deadlock detection and avoidance," *To appear in IEE Transactions on Computers and Digital Techniques*, 2005.

[26] LEE, J., MOONEY, V., DALEBY, A., INGSTROM, K., KLEVIN, T., and LINDH, L., "A comparison of the RTU hardware RTOS with a hardware/software RTOS," *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'03)*, pp. 683–688, January 2003.

[27] LEHOCZKY, J., SHA, L., and DING, Y., "The rate monotonic scheduling algorithm: Exact characterisation and average case behavior," in *Proceedings of 10th IEEE Real-Time Systems Synposium*, pp. 166–171, December 1989.

[28] LEHOCZKY, J., SHA, L., and DING, Y., "The rate monotonic scheduling algorithm:exact characterization and average case behavior," in *Proc. IEEE 10th Real-Time System Symposium*, pp. 166–171, 1989.

[29] LEHOCZKY, J., SHA, L., and DING, Y., "Fixed priority scheduling of periodic task sets with arbitrary deadlines," *Proceedings of 11th IEEE Real-Time Systems Synposium*, pp. 201–209, December 1990.

[30] LI, Y. and MALIK, S., *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[31] LI, Y., MALIK, S., and A.WOLFE, "Efficient microarchitecture modeling and path analysis for real-time software," *Proceedings of IEEE Real-Time Systems Symposium*, pp. 298–397, December 1995.

[32] LI, Y., MALIK, S., and WOLFE, A., "Performance estimation of embedded software with instruction cache modeling," *ACM Transaction on Design Automation of Embedded Systems*, vol. 4, pp. 257–279, July 1999.

[33] LIEDTKE, J., HARTIG, H., and HOHMUTH, M., "OS-controlled cache predictability for real-time systems," *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pp. 213–227, June 1997.

[34] LIU, C. and LAYLAND, J., "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of ACM*, vol. 20, pp. 26–61, January 1973.

[35] LOSER, J. and HARTIG, H., "Cache influence on worst case execution time of network stacks," Tech. Rep. TUD-FI02-07, TU Dresden, July 2002.

[36] MAKI, N., HOSON, K., and ISHIDA, A., "A data-replace-controlled cache memory system and its performance evaluations," *Proceedings of IEEE Region 10 Conference*, pp. 471–474, April 1999.

[37] MAY, D., IRWIN, J., MULLER, H., and PAGE, D., "Effective caching for multithreaded processors," *Communicating Process Architectures 2000*, pp. 145–154, September 2000.

[38] MAY, D., PAGE, D., IRWIN, J., and MULLER, H., "Microcache," *6th International Conference On High Performance Computing*, pp. 21–27, December 1999.

[39] MediaBench, Available HTTP: http//cares.icsl.ucla.edu/MediaBench/.

[40] Mentor Graphics Corp. Avaliable HTTP: http://www.mentor.com/.

[41] Mentor Graphics, Hardware/Software Co-Verification: Seamless. Avaliable HTTP: http://www.mentor.com/seamless/.

[42] Mentor Graphics, XRAY® Debugger. Avaliable HTTP: http://www.mentor.com/xray/.

[43] Mobile Robot Lab at Georgia, http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/.

[44] MOONEY, V. and BLOUGH, D., "A hardware-software real-time operating system framework for SoCs," *IEEE Design and Test of Computers*, pp. 44–51, November-December 2002.

[45] MPEG Homepage, Avaliable HTTP: http://www.chiariglione.org/mpeg/.

[46] MULLER, F., "Compiler support for software-based cache partitioning," *Proceedings of ACM SIGPLAN Workshop on Languages, Compliers and Tools for Real-Time Systems*, pp. 125–133, June 1995.

[47] NEGI, H., MITRA, T., and ROYCHOUDHURY, A., "Accurate estimation of cache-related preemption delay," *Proceedings of ACM Joint Symposium CODES+ISSS*, pp. 201–206, October 2003.

[48] SHALAN, M., *Dynamic Memory Management for Embeded Real-Time Multiprocessor System on a Chip*. PhD thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology, November 2003.

[49] SHALAN, M. and MOONEY, V., "A dynamic memory management unit for embedded real-time system-on-a-chip," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'00)*, pp. 180–186, November 2000.

[50] SHIU, P., TAN, Y., and MOONEY, V., "A novel parallel deadlock detection algorithm and architecture," *9th International Workshop on Hardware/Software Co-Design (CODES'01)*, pp. 30–36, April 2001.

[51] SUH, G., DEVADAS, S., and RUDOLPH, L., "Cache models with applications to cache partitioning," *Proceedings of the 15th international conference on Supercomputing*, pp. 1–12, June 2001.

[52] SUH, G., RUDOLPH, L., and DEVADAS, S., "Dynamic cache partitioning for simultaneous multithreading systems," *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 116–127, September 2001.

[53] SUN, D., BLOUGH, D., and MOONEY, V., "Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications," Tech. Rep. GIT-CC-02-19, Georgia Institute of Technology, April 2002. Available HTTP: http://www.cc.gatech.edu/tech_reports/index.03.html.

[54] Synopsys, Inc. Avaliable HTTP: http://www.synopsys.com/.

[55] Synopsys, VCS^{TM} Verilog Simulator. Avaliable HTTP: http://www.synopsys.com/products/simulation/simulation.html.

[56] TAN, Y. and MOONEY, V., "Integrate inter- and intra- cache eviction anlaysis for preemptive multi-tasking real-time systems," *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES'04)*, pp. 200–206, September 2004.

[57] TAN, Y. and MOONEY, V., "Timing analysis for preemptive multi-tasking real-time systems," *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pp. 1034–1039, February 2004.

[58] THEILING, H., FERDINAND, C., and WILHELM, R., "Fast and precise WCET prediction by seperate cache and path analyses," *Real-Time Systems*, vol. 18, May 2000.

[59] TINDELL, K., "Using offset information to analyze static priority preemptively scheduled task sets," tech. rep., Department of Computer Science, University of York, August 1992.

[60] TINDELL, K., BURNS, A., and WELLINGS, A., "An extendible approach for analyzing fixed priority hard real-time tasks," *Real-Time Systems*, vol. 6, pp. 133–151, March 1994.

[61] TOMIYAMA, H. and DUTT, N., "Program path analysis to bound cache-related preemption delay in preemptive real-time systems," *Proceedings of the Eighth International Workshop on Hardware/software Codesign*, pp. 67–71, May 2000.

[62] Vela Research LP, Avaliable HTTP: http://www.vela.com.

[63] VERA, X., LISPER, B., and XUE, J., "Data cache locking for higher program predictability," in *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, pp. 272–282, June 2003.

[64] WAGNER, R., "Compiler-controlled cache mapping rules," Tech. Rep. DUKE–TR–1995–31, Duke University, 1995.

[65] WHITE, R., MUELLER, F., HEALY, C., WHALLEY, D., and HARMON, M., "Timing analysis for data caches and set-associative caches," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 192–202, June 1997.

[66] WHITE, R., MUELLER, F., HEALY, C., WHALLEY, D., and HARMON, M., "Timing analysis for data and wrap-around fill caches," *Real-Time Systems*, vol. 17, no. 2-3, pp. 209–233, 1999.

[67] WHITE, R., WHALLEY, D., and HARMON, M., "Bounding worst-case data cache performance," *IEEE Real-Time Systems Symposium*, December 1996.

[68] WOLF, F., *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers, 2002.

[69] "IBID., PAGE 113"

[70] WOLF, F., ERNST, R., and YE, W., "Path clustering in software timing analysis," *IEEE Transactions on VLSI Systems*, vol. 9, December 2001.

[71] WOLF, F., STASCHULAT, J., and ERNST, R., "Associative caches in formal software timing analysis," *Proceedings of the IEEE/ACM Design Automation Conference*, June 2002.

[72] WOLF, F., STASCHULAT, J., and ERNST, R., "Hybrid cache analysis in running time verification of embedded software," *Design Automation for Embedded Systems*, pp. 271–295, October 2002.

[73] WOLFE, A., "Software-based cache partitioning for real-time applications," *Proceedings of the 3rd Workshop on Responsive Computer Systems*, September 1993.

[74] WOLFE, A., "Software-based cache partitioning for real-time applications," *Journal of Computer and Software Engineering*, vol. 2, no. 3, pp. 315 – 327, 1994.

PUBLICATIONS

This dissertation is based on and extends the work and results presented in the following publications:

[1] TAN, Y. and MOONEY, V., "WCRT Analysis for a Uniprocessor with a Unified Prioritized Cache," Accepted by Languages, Compilers, and Tools for Embedded Systems (LCTES'05), June 2005.

[2] TAN, Y. and MOONEY, V., "Integrate inter- and intra- cache eviction anlaysis for preemptive multi-tasking real-time systems," in *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES'04)*, pp. 200–206, September 2004.

[3] TAN, Y. and MOONEY, V., "Timing analysis for preemptive multi-tasking real-time systems," in *Proceedings of Design, Automation and Test in Europe (DATE'04)*, pp. 1034–1039, February 2004.

[4] TAN, Y. and MOONEY, V., "A prioritized cache for multi-tasking real-time systems," in *Proceedings of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI'03)*, pp. 168–175, April 2003.

[5] SHIU, P., TAN, Y., and MOONEY, V., "A novel parallel deadlock detection algorithm and architecture," in *9th International Workshop on Hardware/Software Co-Design (CODES'01)*, pp. 30–36, April 2001.

[6] TAN, Y. and MOONEY, V., "Timing analysis for preemptive multi-tasking real-time systems with caches," Tech. Rep. GIT-CC-04-02, Georgia Institute of Technology, February 2004.

[7] TAN, Y. and MOONEY, V., "Cache-related timing analysis for multi-tasking real-time systems with nested preemptions," Tech. Rep. GIT-CC-04-11, Georgia Institute of Technology, November 2004.