WCET-Aware Scratchpad Memory Management for Hard Real-Time Systems

by

Yooseong Kim

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved January 2017 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
David Broman
Georgios Fainekos
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

May 2017

ABSTRACT

Cyber-physical systems and hard real-time systems have strict timing constraints that specify deadlines until which tasks must finish their execution. Missing a deadline can cause unexpected outcome or endanger human lives in safety-critical applications, such as automotive or aeronautical systems. It is, therefore, of utmost importance to obtain and optimize a safe upper bound of each task's execution time or the worst-case execution time (WCET), to guarantee the absence of any missed deadline. Unfortunately, conventional microarchitectural components, such as caches and branch predictors, are only optimized for average-case performance and often make WCET analysis complicated and pessimistic. Caches especially have a large impact on the worst-case performance due to expensive off-chip memory accesses involved in cache miss handling. In this regard, software-controlled scratchpad memories (SPMs) have become a promising alternative to caches. An SPM is a raw SRAM, controlled only by executing data movement instructions explicitly at runtime, and such explicit control facilitates static analyses to obtain safe and tight upper bounds of WCETs. SPM management techniques, used in compilers targeting an SPM-based processor, determine how to use a given SPM space by deciding where to insert data movement instructions and what operations to perform at those program locations. This dissertation presents several management techniques for program code and stack data, which aim to optimize the WCETs of a given program. The proposed code management techniques include optimal allocation algorithms and a polynomial-time heuristic for allocating functions to the SPM space, with or without the use of abstraction of SPM regions, and a heuristic for splitting functions into smaller partitions. The proposed stack data management technique, on the other hand, finds an optimal set of program locations to evict and restore stack frames to avoid stack overflows, when the call stack resides in a size-limited SPM. In the evaluation, the WCETs of various benchmarks including real-world automotive applications are statically calculated for SPMs and caches in several different memory configurations.

ACKNOWLEDGMENTS

My Ph.D. studies were truly a life changing experience. I first started with excitement and enthusiasm, but prolonged challenges with research and multifaceted life problems soon filled me with frustration and shame. Now I am finishing my 8 years of studies with great humility and gratitude. I would not have been able to finish, or even keep my sanity, without the help and support of many people. I wish I could present them this thesis as a masterpiece in a way to show my sincere gratitude, but I can only realize my limitations. I am afraid to thank everyone with mere words instead.

Prof. Aviral Shrivastava was the initiator and enabler of all these. He not only gave me opportunities to start the studies, but also sustained me and helped me go through myriads of problems and difficulties. I truly enjoyed working with him and learned from him a different kind of warm leadership that has patience and consistency in being constructive toward making progress. He always rescued me from self-hatred and lack of confidence and finally turned me into a proud student of him. He is my best advisor, mentor, teacher, researcher, and even a good friend and counsellor to me.

Meeting Prof. David Broman was a turning point in my life. He opened my eyes to research with the depth and gravitas that I had not even realized. I will cherish the rewarding discussions and the memories during the time at UC Berkeley. I learned from him invaluable lessons about the researcher's responsibility and ethics, and also perfectionism. It is with my regret that I could and might have done better to make this relationship more fruitful. I would like to sincerely thank Prof. David Broman for all his support and help.

Without the valuable inspiration and advice from Prof. Georgios Fainekos and Prof. Carole-Jean Wu, I would not have been able to finish this thesis. Having them in my committee broadened my view of the problem, and I could significantly improve this thesis

with their valuable comments and suggestions, from the early stage of this work at thesis proposal to the final stage of thesis defense.

During the most time of my Ph.D. studies, I lived in California, away from campus, working remotely. Because of the difficulties and lonesomeness I experienced during this time, I truly understand the value of our comradeship and friendship. Each and every conversation and discussion, short or long, with fellow graduate students helped me get through difficulties at the moment and gave me a new lesson. I would especially like to thank all past and current members of Compiler-Microarchitecture Lab and Korean Student Fellowship in CS department.

I would like to show my deepest gratitude to my wife Soyoun. Without her love, support, encouragement, and endurance, I would never have been able to hold out until this very moment. Also, without my dear daughter Olivia, I could not have realized the truth of life and the value of family. I am grateful to have this wonderful family in my life and to be able to start a new chapter of our life together as family. Last but not least, I would like to thank God for sustaining me so far and for His countless blessings. I trust in the Lord, my good shepherd, and would like to declare that the sovereign God is in control of all things in my life and the world.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Hard real-time systems (Buttazzo, 2011) or cyber-physical systems (Lee, 2008) are subject to strict timing constraints that require tasks to finish execution before their specified *deadlines*. Any failure to meet a timing constraint can lead to unexpected outcomes, thus timing is a key factor in functional correctness, not only a performance measure in these systems. Particularly, in safety-critical applications, such as automobiles or aircrafts, a missed deadline may cause devastating and even life-threatening consequences. It is, therefore, of utmost importance to guarantee that all timing constraints are satisfied.

To check the presence of missed deadlines, testing has been widely used as a traditional approach in industry in the context of system validation. It is, however, very difficult to cover all possible test cases and to identify representative worst-cases. The sheer number of test cases in modern real-time systems makes exhaustive testing infeasible. Moreover, it is often difficult to identify the execution scenarios that lead to the worst-case execution times (WCETs) of tasks. These worst-case scenarios are often counterintuitive due to timing anomalies (Reineke *et al.*, 2006; Lundqvist and Stenström, 1999), commonly present in highly-optimized modern real-time systems. It is not possible to guarantee that the actual WCETs have been observed in testing, and therefore testing is not a safe way of validating timing correctness of hard real-time systems.

The absence of missed deadlines can only be guaranteed by a static analysis (Wilhelm *et al.*, 2008). A static WCET analysis estimates or calculates a *safe upper bound* of the WCETs of each task without actually executing them. As these estimates are safe upper bounds, a system that is validated using a static analysis is guaranteed to have timing correctness at any circumstances.

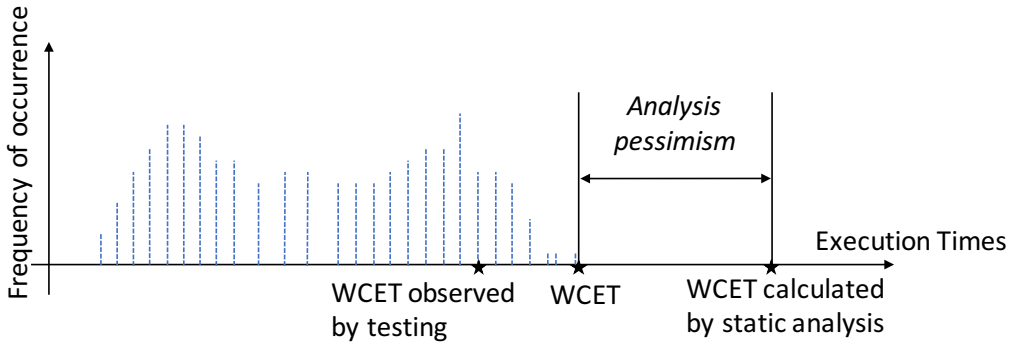
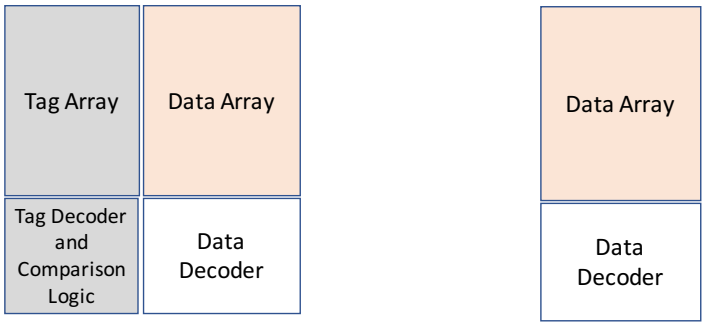


Figure 1.1: Worst-case execution time (WCET) observed by testing may not be the actual WCET, and only by static analyses, we can obtain a safe upper bound of the WCET. The pessimism in the analysis can, however, limit the practical usability of the results.

Although the safety of the results is not of critical importance here, the practical usability of an analysis can be limited because of the tightness of its results. Figure 1.1 illustrates the tightness of the results or pessimism in the analysis, as the gap between the actual WCET and the calculated WCET. For example, assuming a cache miss for every memory access is certainly a safe way to estimate the worst-case memory access times, but it can be so pessimistic that tasks may not be scheduled without significantly upgrading system hardware or changing the design. The tightness of the analysis results are largely affected by the timing predictability (Schoeberl, 2012) of the target processor architecture [1]. Unfortunately, traditional approaches for improving the average-case performance, e.g. caches and speculative execution, often complicate static analyses and make results pessimistic (Axer *et al.*, 2014).

In particular, caches are difficult to analyze statically, and an imprecise cache analysis is an important source of pessimism (Reineke, 2009; Cazorla *et al.*, 2013). The contents of a cache during the execution of a task depend on previous execution history, so all execution

[1] The uncertainties in software, such as unknown loop bounds, recursion depths or pointer values, have a critical impact on the timing predictability, too. They can, however, be controlled and reduced by code annotations and strict coding guidelines. For instance, safety standards for avionics (Radio Technical Commission for Aeronautics Special Committee (152), 1992) or coding guidelines for safety-critical systems (Montgomery, 2013) forbid or strictly limit the usage of recursion or dynamic memory allocation.



(a) Cache (b) Scratchpad Memory (SPM)

Figure 1.2: Scratchpad memories are a directly-addressable raw memory whose data movement is controlled only by software, not by hardware logic as in caches.

paths of the task and all higher-priority tasks including interrupt service routines must be analyzed, which is quite computationally intensive. Notice that for shared caches, all tasks running on all cores must be considered as well. In the presence of sporadic preemptions or under dynamic scheduling policies, accurately predicting the dynamic behavior of caches is nearly impossible because it is not clear when preemptions occur by which tasks. Thus, caches are not as effective in improving worst-case performance as they are in average-case performance. As the complexity and the size of hard real-time applications keep increasing, the difficulties in cache analysis can become a stumble stone in designing correctness-guaranteed hard real-time systems.

1.1 Scratchpad Memory (SPM) and Its Management

Scratchpad memories (SPMs) are a promising alternative to caches in hard real-time systems, for their time-predictable characteristics. SPM is a raw memory that is directly addressable by software. Its simpler hardware, as shown in Figure 1.2, provides less overhead than caches in terms of die area and and per-access power consumption (Banakar *et al.*, 2002a; Redd *et al.*, 2014), which is much appreciated in embedded systems. Data movement in an SPM is explicitly controlled only by executing direct memory access (DMA) instructions whereas that in a cache is implicitly controlled by the addresses of

memory accesses and its replacement policy. Since the updates in the contents are made explicitly, SPMs are more time-predictable and thus facilitate static analyses (Suhendra *et al.*, 2005; Liu *et al.*, 2012). Also, explicit management enables various optimizations in a form of memory space allocation and access scheduling to reduce the WCETs. One key strategy, for example, to optimize the WCET of a task is to avoid memory space sharing among software objects frequently accessed on the worst-case execution path of the task. Moreover, partitioning SPM space to tasks and interrupt service routines can greatly simplify schedulability analysis that calculates the worst-case response times (WCRTs) based on the WCETs and scheduling policy in use. If the memory accesses from each task are completely localized to its partition, preemptions or interrupts do not cause any side effects on the SPM states. This means that *cache-related preemption delay* (CRPD) analysis (Altmeyer *et al.*, 2011; Chattopadhyay and Roychoudhury, 2014; Altmeyer and Burguière, 2011), which tend to be pessimistic, can be completely removed in the design process. Of course, cache locking (Plazar *et al.*, 2012; Ding *et al.*, 2014) and partitioning (Liu *et al.*, 2010; Suhendra and Mitra, 2008) can be used to lower WCETs by reducing conflict misses, but the granularity of these techniques is limited by blocks, lines or ways, which may cause a waste of cache space (Whitham and Audsley, 2009). Link-time optimizations such as code positioning techniques can be used to avoid conflict misses among instructions of different functions and reduce WCETs (Falk and Kotthaus, 2011; Um and Kim, 2003; Li *et al.*, 2015), but the amount of reduction is not significant because these techniques are again, limited by the degree of control provided by caches; for example, line sizes and associativity cannot be changed.

Using an SPM requires a modification in a program; explicit data movement instructions need to be inserted into the program. This modification is usually done by a compiler. Before a compiler performs such a code transform, it needs to make number of decisions. It first needs to decide where in the code to insert a data movement operation. And for each

data movement operation, it needs to decide what to transfer from where and to where. The vast literature on SPM management techniques answers these questions in one way or another with different goals. As techniques to be used in a compiler, they typically employ static analysis and try to optimize reducing power consumptions (Steinke *et al.*, 2002a; Verma *et al.*, 2004; Steinke *et al.*, 2002b), improving average-case performance (Kandemir and Choudhary, 2002; Jung *et al.*, 2010; Bai *et al.*, 2013; Egger *et al.*, 2006; Pabalkar *et al.*, 2008; Baker *et al.*, 2010; Lu *et al.*, 2013; Nguyen *et al.*, 2009; Udayakumaran *et al.*, 2006; Avissar *et al.*, 2002), or worst-case performance (Falk and Kleinsorge, 2009; Puaut and Pais, 2007; Wu *et al.*, 2010; Prakash and Patel, 2012; Suhendra *et al.*, 2005; Wan *et al.*, 2012; Deverge and Puaut, 2007). In this thesis, we present present management techniques to reduce the WCET of a given program and particularly focus on the management of program code and stack data.

SPM management techniques can be either static or dynamic. Any usage of SPM creates a mapping between the main memory contents and the SPM contents. This applies to any on-chip memories; for example, the contents in a cache are a temporary copy or an alias of selected data in the main memory. This mapping between SPM addresses and main memory addresses can be one to one or one to many. In static management, it is one to one such that only selected data can make use of the SPM. The total size of data that can benefit of the SPM cannot be greater than the size of the SPM. For instance, when managing read-only data such as code, the selected instructions are loaded into the SPM at loading time, and the contents of the SPM does not change during runtime. The instructions that are not loaded into the SPM must be fetched directly from the main memory. In dynamic management, on the other hand, the mapping is one to many such that an SPM address can be mapped to many different main memory addresses. When the code of a program is managed in a dynamic way, the contents of the SPM are updated as the program executes by executing DMA loading operations at runtime. Many small applications can benefit

from static management as there is no runtime overhead, but for large applications, the overhead of accessing the slow main memory can outweigh. Dynamic management is inherently more advanced as it can exploit the locality of large applications. All management techniques proposed in this thesis perform dynamic management.

1.2 Overview of This Thesis

This thesis presents dynamic SPM management techniques that optimize the WCET of a given program, particularly focusing on managing program code and stack data. For managing program code, our work is based on function-level overlaying mechanism (Pabalkar *et al.*, 2008; Baker *et al.*, 2010) where a whole function is loaded to the SPM before it is called. Functions are loaded and evicted according a mapping of functions to SPM addresses, which are decided at compile-time. As loading a function using a direct memory access (DMA) transfer is a long-latency operation, it is critical to intelligently allocate SPM space to functions. In Chapter 2, we describe an optimal SPM space allocation technique to minimize the WCET. We further describe a heuristic to split functions into smaller partitions and discuss the impact of management granularities on the WCETs in Chapter 3. Our work for managing stack data is based on the dynamic management mechanism by Lu *et al.* (2013). In this mechanism, while the program stack is kept in the SPM, stack frames are evicted to and restored from the main memory at a call site to prevent the stack from growing larger than the SPM size. Since the size (depth) of the stack and the frequency of management operations for transferring stack frames would be different at different call sites, it is important to decide when to perform such operations. This problem can be seen as scheduling of stack management operations, and we describe an optimal technique for finding call sites to perform the management operations in Chapter 4.

1.3 Contributions

The ultimate goal of this work is to enable correct-by-construction timing of hard real-time systems, which can guarantee the correct timing behavior at design-time or compile-time, without exhaustive testing. As stated earlier in this chapter, the pessimistic static analysis for existing cache-based architectures was a stumble stone in this regard. The significant reduction in the WCET estimates by the proposed techniques, compared to caches, can be seen that this is a step forward toward the goal of correct-by-construction timing of hard real-time systems. Other main contributions of this thesis are as follows.

- **Finding optimal solutions in dynamic management**: All techniques presented in this thesis perform dynamic management and can find optimal solution to minimize the WCET of a given program. All previously published techniques, regardless for allocating program code or for scheduling stack management operations, can only find optimal solutions for static management (Falk and Kleinsorge, 2009; Suhendra *et al.*, 2005) or are optimized for the average-case performance (Avissar *et al.*, 2002). Though claimed to be optimal for the worst-case, some work solves a very limited subset of a problem like selecting instructions in non-nested loops (Wu *et al.*, 2010) or cannot really find an optimal solution as it takes one nominal input assumed to be the worst-case scenario (Whitham and Audsley, 2012; Liu and Zhang, 2015).

- **Extensive evaluations**: The effectiveness of the proposed managed techniques are thoroughly verified by extensive evaluations. We use not only several benchmarks from the Mälardalen WCET suite (Gustafsson *et al.*, 2010a) and MiBench suite (Guthaus *et al.*, 2001), but also three proprietary real-world applications from industry, for automotive powertrain control. In chapter 3, we compare against static analysis results for various previous techniques and caches, with several different architectural configurations.

1.4 Related Publications

The main body of this thesis is composed of three chapters, each of which is a paper under review for publication.

Chapter 2 is an extended version of a published paper [2]. It is under review as of January 2017 at journal, ACM Transactions on Embedded Computing Systems (TECS), with a title of "WCET-Aware Function-level Dynamic Code Management on Scratchpad Memory". I invented all technical parts of this work, including input representation, analysis techniques, integer linear programming formulation, and the heuristic algorithm. All evaluations are also designed and performed by myself. The contributions of co-authors include, but not limited to, the formalism in writing of input representation and algorithm descriptions (David Broman) and the implementation of software infrastructure for constructing control-flow graphs and an early version of cache analysis technique (Jian Cai).

Chapter 3 is under review as of January 2017 at ACM Transactions on Architecture and Code Optimization (TACO), with a tile of "A Comparison of WCET-Centric Dynamic Code Management Techniques for Scratchpads". I am the sole contributor of this work.

Chapter 4 is under review as of January 2017 for publication in the proceedings of the 54th IEEE/ACM Design Automation Conference (DAC). I am the sole technical contributor in this work as well. I developed and implemented all algorithms. I also designed and performed all evaluations. A co-author, Jian Cai, helped with implementing a previous work for comparison, with extensive discussions.

[2] ©2014 IEEE Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastava, "WCET-aware dynamic code management on scratchpads for Software-Managed Multicores", In Proceedings of Real-Time and Embedded Technology and Applications Symposium (RTAS), Apr., 2014. The publisher (IEEE) does not require individuals working on a thesis to obtain a formal reuse license.

Chapter 2

WCET-AWARE FUNCTION-LEVEL DYNAMIC CODE MANAGEMENT [1]

SPM has time-predictable characteristics since its data movement between the SPM and the main memory is entirely managed by software. One way of such management is dynamic management. In dynamic management of instruction SPMs, code blocks are dynamically copied from the main memory to the SPM at runtime by executing direct memory access (DMA) instructions. Code management techniques try to minimize the overhead of DMA operations by finding an allocation scheme that leads to efficient utilization. In this chapter, we present three function-level code management techniques. These techniques perform allocation at the granularity of functions, with the objective of minimizing the impact of DMA overhead to the worst-case execution time (WCET) of a given program.

2.1 Introduction

In this chapter, we describe techniques that allocate code blocks to SPM, called code management techniques, and the objective of the techniques is to reduce the WCET of a given program. Compared to previous code management techniques that try to reduce WCET (Falk and Kleinsorge, 2009; Prakash and Patel, 2012; Puaut and Pais, 2007; Wu *et al.*, 2010), our techniques have two distinct characteristics.

The first difference is at the granularity of management. Code management techniques perform management at various levels of granularity, such as basic blocks (Steinke *et al.*, 2002a; Janapsatya *et al.*, 2006; Puaut and Pais, 2007; Wu *et al.*, 2010), groups of basic blocks on a straight-line path (Verma *et al.*, 2004), or fixed-size pages (Egger *et al.*, 2006).

[1] This chapter extends a published paper, ©2014 IEEE, Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastava, "WCET-aware dynamic code management on scratchpads for Software-Managed Multicores", In Proceedings of Real-Time and Embedded Technology and Applications Symposium (RTAS), Apr., 2014. The publisher (IEEE) does not require individuals working on a thesis to obtain a formal reuse license.

In this chapter, we focus on function-level code management techniques (Baker *et al.*, 2010; Pabalkar *et al.*, 2008; Jung *et al.*, 2010; Bai *et al.*, 2013), which load code blocks at the granularity of functions.

The second difference also comes from the function-level management. In function-level management, a function is loaded as a whole, and instructions are always fetched from the SPM, not the main memory [2]. Other management schemes, on the other hand, allocate only part of the instructions to the SPM and leave the rest in the main memory. The accesses for the instructions left in the main memory are assumed to be uncached and slow or to be cached, which can be less time-predictable.

All previous function-level code management techniques aim to optimize *average-case execution time* (ACET), by reducing overall DMA operation overhead, but none of them considers WCET. We present the first and only function-level code management techniques that optimize the WCET of a given program. Also, all previous techniques use *function-to-region mappings* (Pabalkar *et al.*, 2008) to allocate SPM space to functions. The proposed techniques can not only find an optimal function-to-region mapping for WCET but can also find an optimal *region-free mapping* that maps functions directly to SPM addresses, not regions, which can lead to a lower WCET than the optimal function-to-region mapping. We evaluate our approach using several benchmarks from Mälardalen suite (Gustafsson *et al.*, 2010a), MiBench suite (Guthaus *et al.*, 2001), and proprietary automotive control applications from industry. The evaluation results show that our techniques can effectively reduce the WCETs of programs.

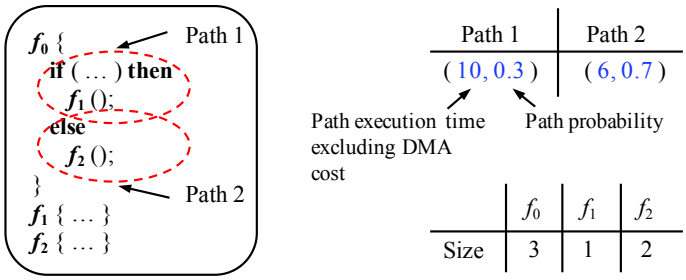
[2] This makes function-level code management techniques only viable options in software-managed multicore architectures (Bai *et al.*, 2013), like IBM Cell processor (Kahle *et al.*, 2005), in which cores cannot directly access the main memory.

2.2 Background: Function-level Dynamic Code Management

When the SPM is large enough to store all instructions, running a program is very simple; the whole program can be loaded at loading time before execution. Dynamic code management, however, becomes necessary once the code size becomes larger than the SPM size. In dynamic management, compiler inserts DMA instructions for loading operations so that the contents stored in the SPM can be updated with different instructions at runtime. Management techniques coordinate DMA operations in order to avoid the overhead of such DMA operations.

Function-level code management (Pabalkar *et al.*, 2008) loads instructions at the granularity of functions around each call site. Since it is assumed that the core fetches instructions only from the SPM, a whole function must be loaded in the SPM before executing the function. This imposes a limitation that the largest function in a program must fit in the SPM in order to be executable using function-level code management techniques. Where to load each function is decided at compile time, and in all previous approaches, such decisions are represented by function-to-region mappings. A function-to-region mapping is a surjective map from all functions in the program to all regions in the SPM.

Code management using function-to-region mappings is analogous to a direct-mapped cache. A region corresponds to a cache line. As memory addresses are mapped to cache lines, functions are mapped to regions. A function is always loaded to the starting address of its region, so loading a function always replaces any previously loaded function in the region. At a call (return), the compiler-inserted code looks up the state of the region to check if the callee (caller) function is loaded in the region. If not, the function is loaded by a DMA operation, and the core waits until it finishes before proceeding to execute the function. This process is analogous to tag comparison and cache miss handling in caches.



(a) An example program

		Path 1	Path 2
Region 1	f_0, f_1	load f_1	load f_2
Region 2	f_2	load f_0	
		10+1+3=**14**	6+2=**8**

(b) Mapping A

		Path 1	Path 2
Region 1	f_0, f_2	load f_1	load f_2
Region 2	f_1		load f_0
		10+1=**11**	6+2+3=**11**

(c) Mapping B

	Mapping size	ACET	WCET
A	$max(3,1)+2=5$	$14*0.3+8*0.7=9.8$	$max(14,8)=14$
B	$max(3,2)+1=4$	$11*0.3+11*0.7=11$	$max(11,11)=11$

(d) ACET and WCET comparison

Figure 2.1: Mapping A optimizes the more frequently executed path (Path 2), achieving a better ACET than mapping B. In terms of WCET, however, mapping B is a better solution.

2.3 Motivating Examples

In this section, we use a simple motivating example to demonstrate the difference between the mapping optimized for ACET and the mapping optimized for WCET. Then, we show another motivating example to explain the benefit of mapping functions directly to addresses, instead of regions.

2.3.1 Why Do We Need a New Technique for Optimizing WCET?

Figure 2.1(a) shows an example program with three functions: f_0, f_1, and f_2. The main function f_0 has two paths, calling functions f_1 on Path 1 and f_2 on Path 2. The probability of the program to take each path is determined by the branch probability of the `if`-statement in f_0. The execution time of each path excluding the waiting time for DMA operations and path probabilities are also shown in the figure. The cost for loading each function is assumed to be the same as the size of the function.

Let us assume the size of the SPM is 5. Since the sum of all function sizes is larger than the SPM size, not all functions can have a private region. Here, we consider two feasible mapping solutions: mapping f_0 and f_1 to the same region (Mapping A) and mapping f_0 and f_2 to the same region (Mapping B). Figure 2.1(b) and 2.1(c) compare the sequence of DMA operations on each path for each mapping choice. For instance, with mapping A, f_0 must be loaded again when f_1 returns because f_0 was evicted by f_1.

Figure 2.1(d) shows the ACET and the WCET for each mapping. Considering path probabilities, mapping A achieves a better ACET than mapping B. The overall amount of DMA transfers is less with mapping A because it can avoid evicting the largest function, f_0, on the more frequently executed path, Path 2. The WCET of the program is, however, better with mapping B [3].

All previous approaches use *interference cost* to model the cost of mapping two functions into the same region. For example, Bai *et al.* (2013) calculates the calories cost of mapping two functions A and B into one region as $p_A \times p_B \times min(n_A, n_B) \times (s_A + s_B)$, where p_f, n_f, and s_f denote the execution probability, the iteration count, and the size of function f. Thus, the interference cost of mapping f_0 and f_1 into one region is $1 \times 0.3 \times min(1,1) \times (3 + 1) = 1.2$. Similarly, the cost of mapping f_0 and f_2 into the same region

[3] In fact, the best mapping for both the ACET and the WCET would be mapping f_1 and f_2 into the same region and leaving f_0 in a private region, but we only consider mapping A and B for illustrative purposes.

is $1 \times 0.7 \times min(1,1) \times (3+2) = 3.5$. Trying to minimize the interference cost, the technique from Bai *et al.* would always try to map f_0 and f_2 to different regions, and any other previous approaches would work similarly.

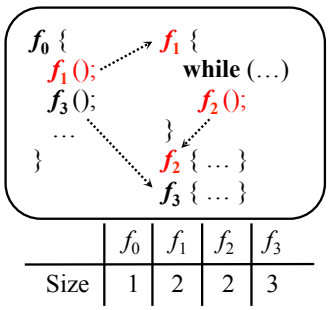
This example shows that optimizing for the ACET may not always result in a good WCET. Previous mapping techniques only try to optimize the ACET and are therefore not suitable for systems with strict timing constraints.

2.3.2 Why Do We Need Region-Free Mapping?

Figure 2.2(a) shows an example program with four functions, f_0, f_1, f_2, and f_4. f_0 first calls f_1, and then f_1 calls f_2 in a loop. After f_1 returns, f_0 calls f_3. The execution sequence of the functions is $f_0 f_1 (f_2 f_1)^n f_0 f_3 f_0$, where *n* is the number of iterations of the loop in which f_2 is called. We assume f_0 is preloaded before execution.

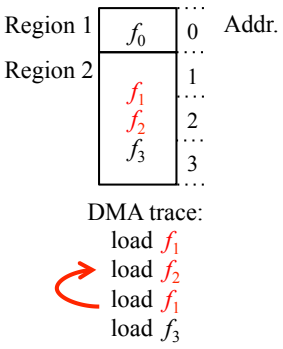
Let us assume the SPM size is 4. When a function-to-region mapping is used, it is not possible to assign separate regions to f_1 and f_2. This is because the size of the largest function, f_3, is 3, so at least one region has to be as large as 3. The remaining SPM space is only 1, and the only function that can fit in a region whose size is 1 is f_0. Thus, the optimal function-to-region mapping, shown in Figure 2.2(b), is to map f_0 in one region of size 1, and all the rest to the other region of size 3. With this mapping, f_0 is kept loaded in a separate region, so it is not reloaded again when other functions return. This mapping, however, causes f_1 and f_2 to replace each other repetitively in the loop, causing DMA operations in every iteration. This is a significant overhead and can greatly increase the WCET of the program.

If we can map each function directly to an address range, not a region, this problem can be solved. As shown in Figure 2.2(c), f_1 and f_2 can be mapped to disjoint address ranges, from 0 to 1 and from 2 to 3, respectively. This can greatly improve the WCET because B and C can stay loaded after their initial loadings. This mapping causes f_1 to be reloaded

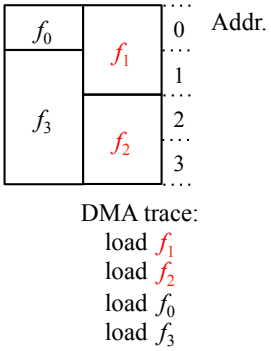


	f_0	f_1	f_2	f_3
Size	1	2	2	3

(a) An example program



(b) Optimal function-to-region mapping



(c) Optimal region-free mapping

Figure 2.2: Even with the optimal function-to-region mapping, f_1 and f_2 replace each other repetitively in the loop. Region-free mapping load them to disjoint address ranges, keeping them loaded after initial loadings.

when f_1 returns back to f_0 because their allocated SPM spaces overlap, but it happens only once. When f_3 returns, f_0 does not need to be reloaded.

2.4 WCET Analysis

In order to find a mapping that can optimize the WCET of a program, we first need to be able to estimate the WCET of the program for a mapping—which can be either a function-to-region mapping or a region-free mapping. Figure 2.3 shows an overview of our WCET analysis framework. Given a graph representation of the program, we need to perform two analyses to obtain necessary information about the program. Using this information, along

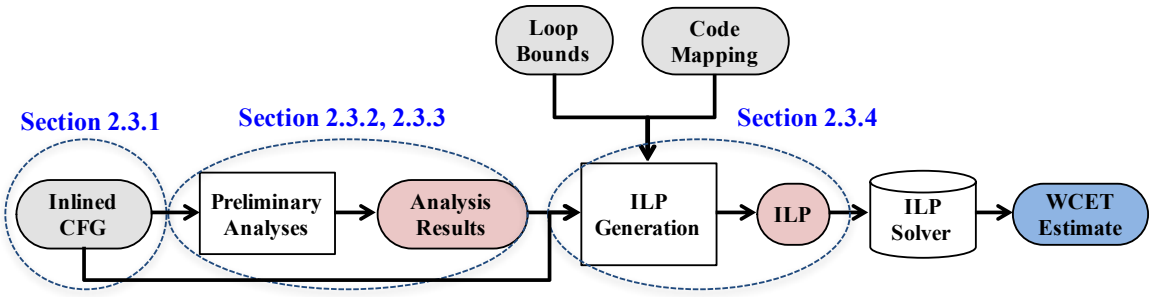


Figure 2.3: The flow of our WCET analysis for function-level dynamic code management

with a mapping, and loop bounds, we formulate an integer linear programming (ILP) to compute a safe upper bound of the WCET.

2.4.1 Inlined Control Flow Graph

We use a variant of control flow graph (CFGs), called *inlined* CFGs, to represent a given program. An inlined CFG is a CFG of a whole program, not just one function, whose edges represent not only control flows within a function but also function calls and returns. An example program is depicted in Figure 2.4. In this example, like the example from Figure 2.1(a), the main function, f_0 has one branch and calls f_1 and f_2. We assume that both f_1 and f_2 consist of a single basic block. When f_0 calls f_1 at v_1, the CFG of f_1 is inlined as v_3, and similarly the CFG of f_2 is inlined as v_4. The notable benefit with this representation is that context information or call stack trace is explicit at any node in a graph, which avoids pessimism regarding uncertainties with call history in static analysis. One limitation is that recursive functions cannot be represented, which can be acceptable in the context of real-time embedded applications. Note that this is only a program representation for analysis. Any program can be represented in an lined CFG without actual inlining or any other modification.

Let $G = (V, E, v_s, v_t, F, fn)$ be an inlined CFG. V is the set of vertices, each of which is a basic block. The set of edges is defined as $E = \{(v, w) |$ there is a direct path from v to w due to control flow, a function call or a return, where $v, w \in V.\}$. Unlike basic blocks

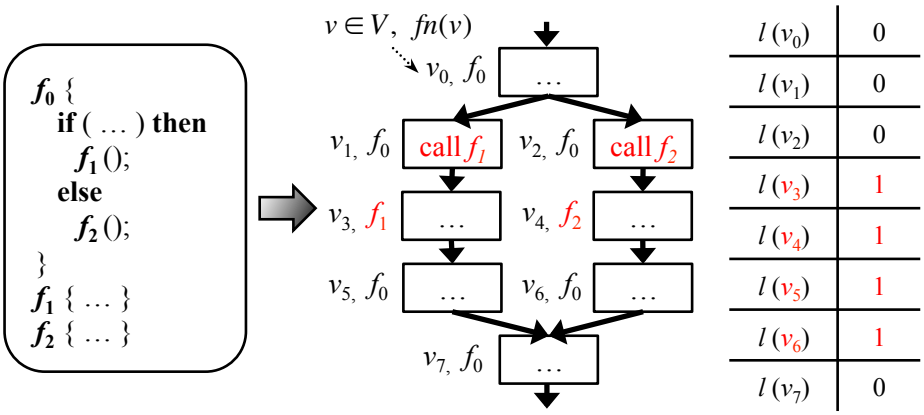


Figure 2.4: Inlined CFG represents the global call sequence and the control flow by inlining the CFG of the callee function at each function call.

in conventional CFGs, function call instructions are always at the end of a basic block and cannot be in the middle of a basic block. Vertices v_s and v_t represent the starting basic block and the terminal basic block. F is the set of functions in the program, and $fn : V \rightarrow F$ is a mapping stating that $fn(v)$ is the function that v belongs to.

A mapping $l : V \rightarrow \{0,1\}$ identifies *loading points* of functions. For a vertex v, $l(v)$ is 1 only when there is an immediate predecessor u such that $fn(u) \neq fn(v)$, which means there is an incoming edge from another function. Figure 2.4 illustrates $fn(v)$ and $l(v)$.

We also define the concept of paths and related notations as follows. A path is a finite sequence of vertices $p = p_1, p_2, \cdots, p_k$ such that $\forall 1 \leq i \leq k$, $p_i \in V$ and $\forall 1 \leq i < k$, $\exists (v_i, v_{i+1}) \in E$. The i-th vertex on p is denoted by p_i, and the length of a path p is denoted by $len(p)$. A vertex can appear multiple times on a path for the presence of loops. Given a vertex v, $P(v)$ denotes the set of all paths that start from v_s and end with an immediate predecessor of v. Thus, v itself is not included in $P(v)$. For a path p and a function f, $last(p, f) \in V \cup \{\perp\}$ denotes the last occurrence of f on p. Thus, if we let $last(p, f) = p_i$, then $fn(p_i) = f$ and $fn(p_j) \neq f$, $i < j \leq len(p)$. When f does not appear on p, $last(p, f) = \perp$.

2.4.2 Finding Initial Loading Points

A function needs to be loaded at least once when it is called for the first time, which is analogous to cold misses in caches. For vertex v that is a loading point of $fn(v)$, if there is any execution path from v_s to v on which $fn(v)$ has never been executed, we call v an initial loading point of $fn(v)$. We have to assume that a DMA operation must take place at least once for initial loading in such a case.

We define a binary mapping $il : V \rightarrow \{0, 1\}$ to identify initial loading points of functions. For a vertex $v \in V$, $il(v)$ is 1 only when v is an initial loading point of $fn(v)$, which is determined using traditional dominance analysis (Khedker *et al.*, 2009) as follows.

$$il(v) = \begin{cases} 0 & \exists d \in SDOM(v),\ fn(d) = fn(v) \\ 1 & \text{otherwise.} \end{cases} \tag{2.1}$$

where $SDOM(v)$ denotes the set of strict dominators of v. If there is any strict dominator d whose $fn(d)$ value is the same as $fn(v)$, function $fn(v)$ can be safely assumed to have been loaded before executing v. Otherwise, v is an initial loading point. In the example program in Figure 2.4, v_3 is a potential loading point of f_1, and its strict dominators are v_0 and v_1. Since both v_0 and v_1 belong to f_0, not f_1, v_3 is an initial loading point of f_1, thus $il(v_3) = 1$. Similarly, v_4 is also an initial loading point of f_2.

2.4.3 Finding the Interference among Functions

At a loading point v that is not an initial loading point, $fn(v)$ is guaranteed to have been loaded before control reaches v. To determine if $fn(v)$ is still loaded at v, we make a conservative assumption as follows. If there exists a function $g \neq fn(v)$ that satisfy the following two conditions, we assume that $fn(v)$ has been evicted from the SPM:

1. g and $fn(v)$ share SPM space (Their allocated SPM spaces overlap.).

2. There exists a path $p \in P(v)$, on which g is executed between $last(p, fn(v))$ and v.

Satisfying two conditions means, in other words, that $fn(v)$ could have been evicted by g on a path from $last(p, fn(v))$ to v. The first condition cannot be checked because the SPM addresses of functions are not decided before code mapping stage (Section 2.5). The second condition, however, can be checked by analyzing the given CFG.

If the second condition satisfies, $fn(v)$ and g have *interference* at loading point v. Our interference analysis [4] finds the set of all functions that potentially interfere with f at all loading points v, namely *interference set*, defined as below.

Definition 1 (Interference Set) *Let $G = (V, E, v_s, v_t, F, fn)$ be an inlined CFG. For a vertex $v \in V$ and a function $f \in F$, the interference set $IS[v, f] \subseteq F \setminus \{f\}$ is the set of all functions that appear between the path between $last(p, f)$ and v, excluding $last(p, f)$ and v, for all paths $p \in P(v)$.*

When $last(p, f)$ is $\perp$ for all path $p \in P(v)$, $IS[v, f] = \emptyset$. The following equation restates the above definition.

$$\forall v \in V, f \in F, \quad IS[v, f] = \bigcup_{\forall p \in P(v)} \{fn(p_j) \mid i < j \leq len(p),\ p_i = last(p, f)\} \tag{2.2}$$

Table 2.1 shows interference sets for the example in Figure 2.4. To help follow how interference sets are calculated at each vertex v, the table also shows the set of $last(p, f)$ for all paths $p \in P(v)$ on the right three columns.

In other words, interference set $IS[v, f]$ is the set of functions that *could* evict f from the SPM before f is executed at v. The eviction can actually occur if any function in

[4] The term "interference analysis" has been used in the context of compiler optimization, such as in register allocation or in optimizing parallel programs. Our interference analysis is different from any of those, but similar in the sense that the results are used to predict any side-effect of compiler decision. For example, allocating a register to a variable may cause additional spills of other interfering variables, and mapping a function to an SPM address may cause additional DMA overhead for reloading other functions interfering with the function.

Table 2.1: Interference sets for the example program in Figure 2.4

	$IS[v, f_0]$	$IS[v, f_1]$	$IS[v, f_2]$	$\bigcup_{\forall p \in P(v)}\{last(p, f)\}$		
				$f = f_0$	$f = f_1$	$f = f_2$
$v_0, v_1, v_2,$ v_3, v_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$ or has only immediate predecessor of v.		
v_5	$\{f_1\}$	$\emptyset$	$\emptyset$	$\{v_1\}$	$\{v_3\}$	$\emptyset$
v_6	$\{f_2\}$	$\emptyset$	$\emptyset$	$\{v_2\}$	$\emptyset$	$\{v_4\}$
v_7	$\emptyset$	$\{f_0\}$	$\{f_0\}$	$\{v_5, v_6\}$	$\{v_3\}$	$\{v_4\}$

$IS[v, f]$ is assigned an SPM space that overlaps with the SPM space assigned for f. Since a loading point v loads $fn(v)$, only $IS[v, fn(v)]$ is meaningful in estimating DMA costs. Nevertheless, the interference sets are calculated for all functions at each vertex to pass down the information to successor vertices.

We can safely assume that $fn(v)$ is still loaded in the SPM only if: i) v is not an initial loading point of $fn(v)$ ($fn(v) \notin A[v]$), and ii) none of the functions in $IS[v, fn(v)]$ shares SPM space with $fn(v)$. Otherwise, we have to assume a DMA transfer will take place at v to load $fn(v)$.

Interference sets can be calculated by a form of forward data-flow analysis, using the following data-flow equations. Algorithm 1 shows the procedure of using the equations. Let IN$[v, f]$ and OUT$[v, f]$ be the interference sets $IS[v, f]$ before and after executing v,

Algorithm 1: Interference analysis

Input: Inlined CFG (G)

Output: The interferense sets (IS)

1 **foreach** $(v, f) \in V \times F$ **do** $\text{IN}[v, f] \leftarrow \emptyset$

2 **repeat**

3 **foreach** $(v, f) \in V \times F$ **do**
Evaluate Equations (2.3) and (2.4)

until *$IN[v, f]$ and $OUT[v, f]$ stay unchanged for all $v \in V$ and $f \in F$*

4 **foreach** $(v, f) \in V \times F$ **do**
$IS[v, f] = \text{IN}[v, f] - \{f\}$

respectively.

$$\text{IN}[v, f] = \bigcup_{(u,v)\in E} \text{OUT}[u, f] \tag{2.3}$$

$$\text{OUT}[v, f] = \begin{cases} \emptyset & \text{if } f \neq fn(v) \wedge \text{IN}[v, f] = \emptyset \\ \{fn(v)\} & \text{if } f = fn(v) \\ \text{IN}[v, f] \cup \{fn(v)\} & \text{otherwise.} \end{cases} \tag{2.4}$$

Input value, $\text{IN}[v, f]$, is the union of output values from all predecessors, and there are three different cases regarding how output value, $\text{OUT}[v, f]$, is updated. First, when f is not $fn(v)$, $\text{OUT}[v, f]$ remains empty unless $\text{IN}[v, f]$ has any function in it. $\text{IN}[v, f]$ can become a non-empty set only when f has been executed previously, which is done by the second condition. The second condition says that when f is $fn(v)$, any collected execution history in $\text{IN}[v, f]$ is reset and the output value contains only $fn(v)$. Once this happens, starting from the successors u of v, $\text{IN}[u, f]$ will not be an empty set, and the function execution history can be recorded by taking a union of the input value and $fn(u)$, as seen in the third condition. Notice that Algorithm 1 sets $IS[v, f]$ to be $\text{IN}[v, f] - \{f\}$ at line 4, after all the

data-flow values converge, to make the final results comply to the definition that $IS[v, f]$ does not contain f.

2.4.4 ILP Formulation for WCET Analysis

We describe an integer linear programming (ILP) formulation to find a safe upper bound of the WCET of a given program, when a particular function-to-region mapping for the program is given as input. Variables in the following ILP are written in capital letters, and constants are in small letters. The formulation requires that the input inlined CFG G to be acyclic, so we require G to be reducible and remove all back edges first.

The high-level structure of our formulation is similar to the one from the previous work (Suhendra *et al.*, 2005; Falk and Kleinsorge, 2009) in two aspects: i) a WCET estimate is obtained by accumulating the cost of each basic block backward from the end to the start of the program (Equation (4.2)), and ii) the objective is to minimize the WCET (Equation (4.1)). There are, however, significant differences in the rest of the formulation as we model the function loading cost at each vertex (Equation (2.13)).

Let W_v be a WCET estimate from v to the end of the program. Thus, W_{v_s} is a WCET estimate for the whole program. The objective is to get a safe-yet-tight estimate of the WCET of the program as follows.

$$minimize \quad W_{v_s} \tag{2.5}$$

Each vertex v can contribute to the WCET with the sum of its computation cost C_v and its loading cost L_v. C_v is the time to execute all instructions in v, which excludes the time to execute DMA instructions, which is L_v. For each successor w of v, W_v is greater than or equal to the sum of the cost of v and the cost of w. This makes W_v be a safe upper bound of the WCET from v to the end of the program. The terminal basic block does not have any

successor, so W_{v_t} is the cost of itself.

$$\begin{aligned} \forall (v,w) \in E, \quad & W_v \geq W_w + C_v + L_v \\ & W_{v_t} = C_{v_t} + L_{v_t} \end{aligned} \tag{2.6}$$

The computation cost C_v is a product of the number of times v is executed in the worst-case (n_v) and the worst-case estimation of the time it takes to execute the instructions in v for once (c_v).

$$C_v = n_v \cdot c_v \tag{2.7}$$

For loading cost L_v to exist, v must be a loading point, i.e., $l(v) = 1$. To employ the value of $l(v)$ in the formulation, $l(v)$ is imported as a constant l_v as below. Similarly, the information regarding initial loading point, $il(v)$, is imported as a constant il_v.

$$l_v = l(v) \tag{2.8}$$

$$il_v = il(v) \tag{2.9}$$

The mapping information is taken into account as follows. For a pair of functions, f and g, a binary constant $o_{f,g}$ is only one when their allocated SPM address ranges overlap. When the mapping is function-to-region mapping, this means that both functions are mapped to the same region. With a region-free mapping, this is calculated using the mapped address and the size of each function.

$$o_{f,g} = \begin{cases} 1 & \text{if the allocated SPM spaces for } f \text{ and } g \text{ overlap} \\ o & \text{otherwise.} \end{cases} \tag{2.10}$$

Let d_f denotes the time it takes to load function f by a DMA operation plus the overhead of executing DMA instructions. The loading cost L_v is modeled as follows. Table 2.2 shows different scenarios in which loading cost L_v can exist. If there exists any interfering

Table 2.2: Categorization of function loading cost at vertex v

	Initial loading point ($il_v = 1$)	Non-initial loading point ($il_v = 0$)
$fn(v)$ shares SPM space with an interfering function f ($\exists f \in IS[v, fn(v)],\ o_{fn(v),f} = 1$)	Always-Miss (load n_v times)	
No space sharing ($\forall f \in IS[v, fn(v)],\ o_{fn(v),f} = 0$)	First-Miss (load only once)	No loading

function whose allocated SPM space overlaps with that of $fn(v)$, $fn(v)$ needs to be reloaded every time v is executed. AM_v (Always-Miss) models the loading cost in this case.

$$\forall f \in IS[v, fn(v)], \quad AM_v \geq n_v \cdot d_{fn(v)} \cdot o_{fn(v),f} \tag{2.11}$$

Consider an initial loading point v that is executed more than once in a loop. If there is no interfering function or none of the interfering function shares SPM space with $fn(v)$, $fn(v)$ needs to be loaded only once. FM_v (First-Miss) models the loading cost in this case. The value of FM_v is $d_{fn(v)}$ as $fn(v)$ is loaded only once. If, however, any interfering function shares SPM space with $fn(v)$, it becomes Always-Miss, and the value of FM_v should be the same as AM_v. The difference in AM_v and $d_{fn(v)}$ is compensated by adding $(n_v - 1) \cdot d_{fn(v)}$ to $d_{fn(v)}$ as follows.

$$\forall f \in IS[v, fn(v)], \quad FM_v \geq d_{fn(v)} + (n_v - 1) \cdot d_{fn(v)} \cdot o_{fn(v),f} \tag{2.12}$$

Finally, since the loading cost is present only when l_v is 1, and its value is either FM_v or AM_v, it is modeled by the following constraint.

$$L_v = l_v \cdot (il_v \cdot FM_v \ + \ (1 - il_v) \cdot AM_v) \tag{2.13}$$

After solving the above ILP, the objective value W_{v_s} is a safe WCET estimate of the given program running on an SPM-based architecture with dynamic code management using the given code mapping.

2.5 WCET Optimization

In this section, we present three techniques of finding a code mapping for optimizing the WCET. The first one is optimal and based on ILP, and the second one is a polynomial-time heuristic, but sub-optimal. These two techniques find a function-to-region mapping, whereas the third technique finds an optimal region-free mapping using ILP.

2.5.1 ILP Formulation for Optimal Function-to-region Mapping

We extend the ILP formulation in Section 2.4.4 to *explore all mapping solutions* instead of taking a fixed mapping solution as input. A function-to-region mapping solution is represented by the following binary variables.

$$\forall f \in F, r \in R, \quad M_{f,r} = \begin{cases} 1 & \text{if } f \text{ is mapped to } r \\ 0 & \text{Otherwise} \end{cases} \tag{2.14}$$

The number of regions used in a mapping solution can vary for different solutions. For example, all functions can be mapped to one region (there is only one region.), or each function can be mapped to a unique region (the number of regions is the same as the number of functions.). To handle various number of regions, we let the set of all regions, R, be a set of integers ranging from 1 to $|F|$, each of which represents a unique region. If a mapping solution uses only $n < |F|$ regions, there will be $(|F| - n)$ regions that do not have any functions mapped to them.

The following constraints ensure the feasibility of mapping solutions that the solver will explore. Firstly, every function is mapped to exactly one region.

$$\forall f \in F, \quad \sum_{r \in R} M_{f,r} = 1 \tag{2.15}$$

Secondly, the sum of the region sizes is not greater than the SPM size.

$$\begin{gathered} \forall f \in F, r \in R, \quad S_r \geq M_{f,r} \cdot s_f \\ \text{SPMSIZE} \geq \sum_{r \in R} S_r \end{gathered} \tag{2.16}$$

where SPMSIZE is the size of the SPM, and s_f is the size of function f. S_r is a variable that represents the size of the largest function mapped to r.

For a pair of functions f and g, and a region r, we use a binary variable $M_{f,g,r}$ that is 1 only when f and g are both mapped to r. This is represented by the following logical condition between variables. If both f and g are mapped to r, only the constraints in Set 1 should satisfy but not the constraints in Set 2, and vice versa.

$$\begin{array}{ll} \text{Set 1: } M_{f,r} + M_{g,r} > 1 & \text{Set 2: } M_{f,r} + M_{g,r} \leq 1 \\ M_{f,g,r} = 1 & M_{f,g,r} = 0 \end{array}$$

The above logical constraints can be integer-programmed using the standard way of formulating logical constraints (Bradley *et al.*, 1977) as follows.

$$\begin{gathered} \forall f, g \in F, r \in R, \quad M_{f,r} + M_{g,r} + B \cdot (1 - M_{f,g,r}) > 1 \\ M_{f,r} + M_{g,r} \leq 1 + B \cdot M_{f,g,r} \end{gathered} \tag{2.17}$$

where B is a constant chosen to be large enough so that regardless of the value of $M_{f,g,r}$, both constraints should satisfy at the same time. In this case, B should be at least 2 to make $M_{f,r} + M_{g,r} + B \cdot (1 - M_{f,g,r}) > 1$ satisfiable when $M_{f,g,r}$ is 0.

Then, the constraints for FM_v and AM_v from the constraints in Equation (2.11) and Equation (2.12) need to be rewritten using $M_{fn(v),f,r}$ as follows.

$$\forall f \in IS[v, fn(v)], r \in R, \quad AM_v \geq n_v \cdot d_{fn(v)} \cdot M_{fn(v),f,r} \tag{2.18}$$

$$FM_v \geq d_{fn(v)} + (n_v - 1) \cdot d_{fn(v)} \cdot M_{fn(v),f,r} \tag{2.19}$$

The solution of this ILP formulation is an optimal function-to-region mapping represented by the set of variables $M_{f,r}$ for all function $f \in F$ and region $r \in R$. The final objective value W_{v_s} is the WCET estimate for the found mapping solution.

2.5.2 WMP: A Heuristic Alternative to the ILP Formulation

The ILP-based technique from the previous section can find an optimal solution, but it can take a long time for an ILP solver to find one. As each function needs to be mapped to each region, and the number of regions can be as many as the number of functions, the solution space of the ILP grows exponentially with the number of functions. In our experiments with benchmarks in Section 2.6, it takes less than a second for a solver to find an optimal solution for 'cnt' which has 6 functions, but for '1REG' which has 28 functions, the solver cannot find an optimal solution in 3 hours.

To solve this problem, we present *WCET-aware Merging and Partitioning* (WMP), a polynomial-time heuristic technique which builds upon the ways of searching the solution space of our previous techniques, *function mapping by updating and merging* (FMUM) and *function mapping by updating and partitioning* (FMUP) (Jung *et al.*, 2010). As the name suggests, FMUM starts with assigning a separate region to every function and tries to merge regions so that the mapping can fit in the SPM and the cost of the mapping decreases, whereas FMUP starts with having only one region and iteratively partitions a region into two regions. While the cost function in these techniques estimates the overall amount of DMA transfers, we introduce a new cost function that estimates the WCET of the program.

Algorithm 2: WMP: a heuristic to find a function-to-region mapping for WCET

Input: Inlined CFG (G), SPM size (S) ($S \geq \max_{f \in F} s_f$)

Output: A feasible function-to-region mapping (M) ($\texttt{Size}(M) \leq S$)

function $\texttt{WMP}(G, S)$

1 remove all back edges in G

2 $T \leftarrow$ Topologically sorted vertices of G

3 $M_m \leftarrow \texttt{Merge}(G, T, S)$

4 $M_p \leftarrow \texttt{Partition}(G, T, S)$

5 **if** $\texttt{Cost}(M_m, T) < \texttt{Cost}(M_p, T)$ **then**

6 $M \leftarrow M_m$

else

7 $M \leftarrow M_p$

8 **return** M

Before discussing the details of the WMP algorithm, we would like to point out that the ILP-based technique can also be used as a heuristic with a time limit set to the solver, as we do later in this chapter (Section 2.6). This makes the solver to output the best solution found by the time limit, which may not be optimal. In our experiments with 3-hour time limit, this ILP-based technique could always find solutions that are better (meaning that the resulting WCET is smaller) or at least as good as the solutions found by WMP. This, however, brings up another problem of choosing a time limit that is long enough to find good solutions. For example, in our experiments with benchmark '1REG', the ILP solver needed at least 50 seconds to find a solution as good as the solution found by WMP and at least 20 minutes to find a solution better than WMP's solution. On the other hand, WMP could find a solution within a second for all benchmarks, and the increase in WCETs compared to the ILP with 3-hour time limit is not greater than 6.5%. In this sense, WMP is still a reasonable and scalable alternative to the ILP.

Algorithm 3: Cost function that estimates the WCET for a given mapping M

Input: Function-to-region mapping (M), Topological-sorted vertex list (T)

Output: The WCET estimate ($c[v_t]$)

function `Cost(`M,T`)`

```
1   initialize c[v] to 0 for all v ∈ V
2   for v in T from head to tail do
3       c ← n_v · c_v
4       if l(v) = 1 then
5           if ∃f ∈ IS[v, fn(v)] such that M[fn(v)] = M[f] then
6               c ← c + n_v · d_fn(v)
            else
7               if il(v) = 1 then
8                   c ← c + d_fn(v)
9       c[v] ← c + max_{(u,v)∈E} c[u]
10  return c[v_t]
```

Algorithm 2 shows the pseudocode. Given an inlined CFG, the interference sets and the size of SPM, it returns a function-to-region mapping M. A mapping solution, M, is represented by an integer array whose size is the same as the number of functions. The ID of a function is represented by an array index, and its mapped region is the value of the array element. For example, if function 1 is mapped to region 2, $M[1] = 2$. It finds two mapping solutions by merging and partitioning (line 3-4), whose algorithms are shown in Algorithm 4. The one with a lesser cost is selected (line 5-7).

The cost of a mapping is the WCET estimate of it. We take the longest path in the input inlined CFG as an estimation of the WCET, and to find the longest path, we first remove all back edges from the graph and topologically sort the vertices at line 2-3 in Algorithm 2.

The cost function, `Cost`, shown in Algorithm 3 visits each vertex in topological order and calculates the computation cost (line 3) and the loading cost (line 4-8) for the given mapping. At each vertex v, the final cost of the vertex $c[v]$ is the sum of its own cost and the maximum cost among the costs of all predecessors (line 9). Thus, the cost of the terminal vertex $c[v_t]$ becomes the longest path length.

Algorithm 4 shows two heuristics, `Merge` and `Partition`. `Merge` starts with mapping each function to a separate region (line 1). In every iteration of the while loop at line 2-14, we take every pair of two regions (line 4-5) to merge and create a temporary mapping M' where two regions are merged (line 6-7). We check the cost of M' and keep a record of the best pair of regions to be merged and its cost (line 8-11). After trying all combinations, we change the original mapping M by merging the best pair of regions (line 12). The loop repeats until the mapping can fit in the SPM, i.e. the sum of the sizes of regions is smaller than *SPMSIZE* (line 2). `Partition` starts with mapping all functions to one region (line 14). Variable nr represents the current number of regions. Again, we create a duplicate of M and move each function f to a different region r, creating another region $nr + 1$ (line 18-20). We keep a record of the best combination of f and r, and its cost (line 21-24). After trying all functions, we move function bf to region br (line 25). The loop repeats until the number of regions is not greater than $|F|$ (line 15) or until the number of regions stops increasing (line 26).

Function `Size` is defined in Algorithm 5. It calculates the memory size requirement of a given mapping M by summing up the size of the largest function in each region.

The while loop in `Merge` takes at most $|F| - 1$ times because the number of regions decreases by one at every iteration. The for-loop nest at line 5-7 takes $|F|^2$ times at most. Merging two regions requires checking every array elements at least once to find all functions mapped to one region and move it to another region, which takes $O(|F|)$ time complexity. The time complexity of `Cost` is $O(|V| \cdot |F - 1|)$ since it visits every vertex only

Algorithm 4: Search feasible mapping solutions by merging and partitioning

Input: Inlined CFG (G), Topologically sorted vertex list (T), SPM size (S) ($S \geq \max_{f \in F} s_f$)

Output: A feasible function-to-region mapping (M) (`Size(`M`)` $\leq S$)

function `Merge(`G,T,S`)`

```
1  initialize M[f] to f for 1 ≤ f ≤ |F|
2  while Size(M) > S do
3      bc ← ∞
4      for r1 = 1 to |F| − 1 do
5          for r2 = r1 + 1 to |F| do
6              M′ ← a duplicate of M
7              merge r1 and r2 in M′
8              nc ← Cost(M′,T)
9              if nc < bc then
10                 br1 ← r1, br2 ← r2
11                 bc ← nc
12     merge br1 and br2 in M
13 return M
```

function `Partition(`G,T,S`)`

```
14 M[f] ← 1 for 1 ≤ f ≤ |F|, nr ← 1
15 while nr ≤ |F| do
16     bc ← Cost(M,T)
17     for f = 1 to |F| do
18         M′ ← a duplicate of M
19         for r = 1 to min(nr + 1, |F|) do
20             M′[f] ← r
21             nc ← Cost(M′,T)
22             if nc < bc ∧ Size(M′) ≤ S
               then
23                 bf ← f, br ← r
24                 bc ← nc
25     if bc < Cost(M,T) then
           M[bf] ← br
26     if br = nr + 1 then
           nr ← nr + 1
       else break
27 return M
```

once, and the number of functions in the interference set $IS[v, fn(v)]$ can be at most $|F|-1$ because $fn(v)$ is excluded in the set. Thus, the time complexity of function `Merge` is $O(|F|^4 \cdot |V|)$. Similarly, `Partition` has the same time complexity because the while loop and for loops at line 16, 18 and 20 iterate at most $|F|$ times.

Algorithm 5: Find the SPM size requirement of a given function-to-region mapping

Input: Inlined CFG(G), function-to-region Mapping (M), Function sizes ($s_f, \forall f \in F$)

Output: The SPM size that the given mapping M requires.

```
function Size(M)
1   initialize S[r] to 0 for 1 ≤ r ≤ |F|
2   for each f ∈ F do
3       if S[M[f]] < s_f then
            S[M[f]] ← s_f
4   return Σ_{r∈R} S[r]
```

WMP algorithm always terminates. In `Merge`, the SPM size S is greater than or equal to the size of the largest function by assertion at the beginning, and the size of mapping, `Size`(M), is reduced after every iteration of while loop (line 2) by merging two regions (line 12). All for loops in `Merge` (line 4-5) have finite loop bound $|F|$, too. Function `Cost` finishes in a finite number of iterations because the vertex list has finite length $|V|$ (line 10) and the interference sets can have at most $|F|-1$ vertices (line 13). Thus, `Merge` terminates in a finite number of steps. Similarly, `Partition` also finishes in a finite number of steps because at the end of every iteration, either the number of regions nr increases or the loop terminates (line 26).

WMP algorithm is sound and complete in that it always finds a solution which is a feasible mapping can fit in the SPM. `Merge` always returns a feasible mapping because in every iteration, two regions are merged in line 12 and the loop termination condition in line 2 ensures the feasibility of the mapping. The initial solution of `Partition` is mapping all functions into one region, which is certainly feasible because of the precondition: $S \geq \max_{f \in F} s_f$. During the execution of the algorithm, the mapping changes only in a way that the resulting mapping fits in the SPM (line 22).

WMP algorithm is, however, not optimal because it does not explore the entire solution space. As a heuristic, WMP trades optimality for speed. For example, `Merge` only con-

siders merging two regions at time in a greedy fashion. Once two regions are merged, the functions in the regions have to be mapped to the same region until the end of the algorithm.

2.5.3 Optimizing WCET using Region-free Mappings

Similarly to the Section 2.5.1, we extend the ILP for WCET analysis from Section 2.4.4 to explore all feasible region-free mapping solutions and find an optimal one.

Variable A_f represents region-free mapping of f, the address at which function f will be loaded, and it should be in the following range.

$$0 \leq A_f \leq \text{SPMSIZE} - s_f \tag{2.20}$$

where s_f denotes the size of function f. Then, the following constraints compare the mapped addresses of two functions and represent their relations. For a pair of functions f and g, binary variable $G_{f,g}$ is 1 if A_f is greater than A_g, and 0 otherwise.

$$\begin{gathered} \forall f, g \in F \text{ such that } f \neq g, \qquad -\mathcal{M}(1 - G_{f,g}) \leq A_f - A_g \leq \mathcal{M} \cdot G_{f,g} \\ G_{f,g} + G_{g,f} = 1 \end{gathered} \tag{2.21}$$

where $\mathcal{M}$ is a sufficiently large integer, used to linearize the *if* conditions. In our formulation, *SPMSIZE* can be safely used as $\mathcal{M}$. For example, if $G_{f,g}$ is 1, the above constraints become $0 \leq A_f - A_g \leq \mathcal{M}$ and $G_{g,f} = 0$, so A_f have to be greater than or equal to A_g. If $G_{f,g}$ is 0, the above constraints become $-\mathcal{M} \leq A_f - A_g \leq 0$ and $G_{g,f} = 1$, so A_f have to be less than or equal to A_g.

The address range that is allocated to function f is $[A_f + 0, A_f + 1, \ldots, A_f + s_f - 1]$, where s_f is the size of f. For a pair of functions f and g, to make sure that their addresses do not overlap, either one of the two constraints should be satisfied: $A_f + s_f - 1 < A_g$ when $A_f < A_g$ ($G_{f,g} = 1$), or $A_g + s_g - 1 < A_f$ when $A_g > A_f$ ($G_{g,f} = 1$). Built on this idea, the following constraints make binary variable $O_{f,g}$ to be 1 if the address ranges of f and

g overlap, and 0 otherwise.

$$\begin{aligned}
\forall f, g \in F \text{ such that } f \neq g, \quad & A_f + s_f < A_g{+}1 + \mathcal{M} \cdot G_{f,g} + \mathcal{M} \cdot O_{f,g} \\
& \mathcal{M} \cdot (1 - O_{f,g}) + A_f + s_f \geq A_g{+}1 + \mathcal{M} \cdot O_{f,g} \\
& A_g + s_g < A_f{+}1 + \mathcal{M} \cdot G_{g,f} + \mathcal{M} \cdot O_{f,g} \\
& \mathcal{M} \cdot (1 - O_{f,g}) + A_g + s_g \geq A_f{+}1 + \mathcal{M} \cdot O_{f,g} \\
& O_{f,g} = O_{g,f}
\end{aligned} \tag{2.22}$$

For example, if $A_f > A_g$ ($G_{f,g} = 1$), the first four lines in the above become as follows.

$$\begin{aligned}
& A_f + s_f < A_g{+}1 + \mathcal{M} + \mathcal{M} \cdot O_{f,g} \\
& \mathcal{M} \cdot (1 - O_{f,g}) + A_f + s_f \geq A_g{+}1 + \mathcal{M} \cdot O_{f,g} \\
& A_g + s_g < A_f{+}1 + \mathcal{M} \cdot O_{f,g} \\
& \mathcal{M} \cdot (1 - O_{f,g}) + A_g + s_g \geq A_f{+}1 + \mathcal{M} \cdot O_{f,g}
\end{aligned}$$

The first line becomes meaningless because of the third term, $\mathcal{M}$, on the right hand side, and the second line also becomes meaningless regardless of the value of $O_{f,g}$ because A_f is greater than A_g. When the address ranges of f and g do overlap ($A_g + s_g \geq A_f + 1$), the third line ensures that $O_{f,g}$ becomes 1, and the fourth line becomes meaningless—it satisfies regardless of the value of $O_{f,g}$. When the address ranges do not overlap ($A_g + s_g < A_f + 1$), the third line becomes meaningless, but the fourth line ensures that $O_{f,g}$ becomes 0.

We rewrite the Equation (2.11) and Equation (2.12) with $O_{f,g}$ variables as below.

$$\forall f \in IS[v, fn(v)], \quad AM_v \geq n_v \cdot d_{fn(v)} \cdot O_{fn(v),f} \tag{2.23}$$

$$FM_v \geq d_{fn(v)} + (n_v - 1) \cdot d_{fn(v)} \cdot O_{fn(v),f} \tag{2.24}$$

The final objective value W_{v_s} after solving this ILP is a WCET estimate, and the A_f variables represent the optimal mapping of functions to SPM addresses.

2.6 Experimental Results

To evaluate our code management techniques, we use various benchmarks from the Mälardalen WCET benchmark suite (Gustafsson *et al.*, 2010a) and MiBench suite (Guthaus *et al.*, 2001), together with three real-world proprietary automotive powertrain control applications from industry. Among 31 benchmarks in Mälardalen suite and 29 benchmarks in MiBench suite, we exclude the ones with recursion or that have less than 6 functions. From Mälardalen suite, we use all 8 benchmarks that have at least six functions and do not have recursion. MiBench suite is in general much larger in size and more complicated, and we were not able to generate the inlined CFGs for 6 benchmarks due to the presence of recursion or function pointers, and 19 due to the complexity of compiled binaries [5]. We use all of the remaining 4 benchmarks from MiBench suite [6].

Table 2.3 shows the benchmarks used in the evaluation. The sizes shown in the table are the sizes after management code is inserted into the code. Only functions in the user code are considered, and library function calls are considered to take the same cycles as normal arithmetic instructions. Benchmarks are compiled for ARM v4 ISA, and inlined CFGs are generated from their disassemblies.

We assume the cost of loading x bytes into SPM by DMA to be $(L - B/W) + (\lceil x/W \rceil)$ cycles, where L is cache miss latency, B is cache block size of the system in comparison, and W is the word size, as it is modeled by Whitham *et al.* (Whitham and Audsley, 2009). The first term is the setup time that takes in every transfer regardless of the transfer size, and the second is the transfer time that corresponds to the transfer size. As in Whitham's work, we use 50, 16, 4 for L,B, and W, respectively. We observed that over a large set

[5] This was only caused by a technical limitation in our implementation regarding makefile build system and not a fundamental limitation.

[6] 'dijkstra' actually has one recursive function call for printing. We commented it out without changing the core algorithm.

Table 2.3: Benchmarks used in our evaluation

	Total code size	Largest function size (B)	Number of functions	Source
cnt	948	332	6	Mälardalen
matmult	1064	304	7	Mälardalen
dijkstra	1644	744	6	MiBench
compress	2892	872	9	Mälardalen
sha	2420	1040	7	MiBench
fft1	3404	1984	6	Mälardalen
lms	3804	980	8	Mälardalen
edn	4624	1924	9	Mälardalen
adpcm	8468	2272	17	Mälardalen
rijndael	9448	3128	7	MiBench
statemate	10580	3520	8	Mälardalen
1REG	27736	7748	28	Proprietary
DAP1	36400	27860	17	Proprietary
susan	51672	10504	19	MiBench
DAP3	56748	43004	28	Proprietary

of different parameters, there was no significant difference in results in terms of relative performance comparison.

To simplify computing WCET estimates, we assume that every instruction takes one cycle as it is on processors designed for timing predictability, such as PRET (Liu *et al.*, 2012; Zimmer *et al.*, 2014). Thus, the worst-case execution time of a basic block in number of cycles is assumed to be the same as the number of instructions in it, unless it has DMA

instructions. All data accesses are from a separate data SPM, without any contention with the accesses to the main memory or the instruction SPM. These assumptions are only for simplifying the evaluation and not limitations of our approach. We can extend our work by combining any microarchitecture analysis work to consider other timing effects such as pipeline hazards, but it is outside the scope of this work.

Loop bounds are found by profiling, except for the powertrain control applications which have infinite loops in the main scheduler. For such benchmarks, loop bounds are set to be a power of 10 according to the level of nesting. We use the Gurobi optimizer 6.5 [7] to solve the ILPs. All experiments are run on 2.2 Ghz Core i7 processor with 16GB of RAM. We set a time limit of 3 hours for the ILP solver so that if it cannot find an optimal solution within 3 hours, we use the best solution found up to that point.

The correctness of our WCET estimation is verified by running selected benchmarks on gem5 simulator (Binkert *et al.*, 2011). We modified the simulator so that it maintains a state machine of an SPM that is updated by DMA operations. Every instruction takes one cycle, and at function calls and returns, additional cycles are taken according to the state of the SPM, for executing management code and DMA operations. The number of cycles each benchmark took on the simulator was always less than or equal to the WCET estimates we obtained by analysis. The whole evaluation setup—the tools for generating inlined CFGs, performing cache analysis, finding mappings, and the simulator—is publicly available for download [8].

We use two SPM sizes, A and B, for each benchmark. A and B are $l + (t - l) * 0.1$ and $l + (t - 1) * 0.3$ respectively, where l is the size of the largest function—at least the largest function should fit in the SPM—, and t is the total code size. We picked these two values, 0.1 and 0.3, to stress test the mapping techniques' capabilities. With too large SPM

[7] Gurobi Optimization, Inc. `http://www.gurobi.com`

[8] `https://github.com/yooseongkim/SPMCodeManagement`

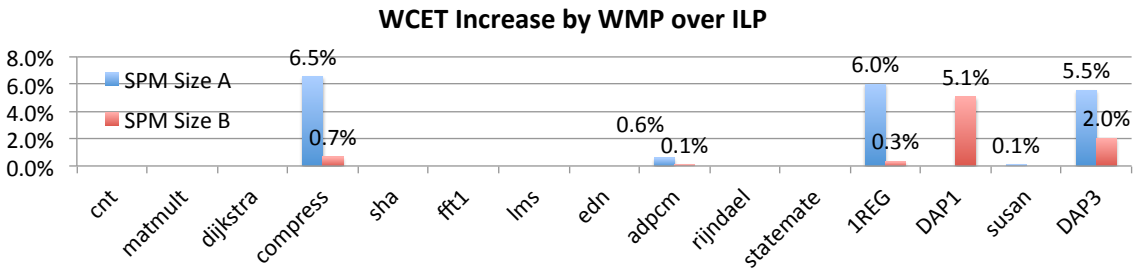


Figure 2.5: The increase in the WCETs by using WMP over ILP is limited within 6.5%.

sizes (values closer to 1), any mapping techniques are likely to allocate separate regions to functions which will generally be beneficial for both ACET and WCET. Likewise, too small SPM sizes (values closer to 0) can be too restrictive to compare mapping techniques effectively.

2.6.1 ILP vs. WMP Heuristic

WMP is a greedy heuristic that may not always be able to find an optimal solution. Figure 2.5 shows the increase in the WCET estimates when the mappings found by WMP are used, compared to the case in which optimal mappings found by the ILP are used. On x-axis, there are two cases for each benchmark, showing two different SPM sizes. In most cases, WMP heuristic can find the same optimal solutions as the ILP-based technique does. Even for the cases where it cannot, the WCET estimates are increased at maximum 6.5%.

Table 2.6.1 shows the algorithm execution times of both ILP-based mapping technique and the heuristic and the resulting WCET estimates of benchmarks. The algorithm execution times include the times for running all analyses. The ILP-based technique can find an optimal solution within seconds for most cases, but for larger benchmarks like '1REG' and 'DAP3', it cannot finish within the time limit. In contrast, WMP can finish under a second for all benchmarks.

A point worth noting here is that as Figure 2.5 shows, the WCETs resulting from using the ILP-based technique with the 3-hour time limit are always lower than or at least the

	SPM	Execution time (sec.)				SPM	Execution time (sec.)		
	Size	rbILP	rfILP	WMP		Size	rbILP	rfILP	WMP
cnt	432	0.01	0.001	$< 10^{-3}$	adpcm	2896	19.41	11.23	$< 10^{-3}$
	528	0.07	0.001	$< 10^{-3}$		4144	8890.3	235.5	0.05
matmult	384	0.01	0.01	$< 10^{-3}$	rijndael	3760	0.02	0.001	$< 10^{-3}$
	544	0.1	0.001	$< 10^{-3}$		5024	0.46	0.14	$< 10^{-3}$
dijkstra	848	0.05	0.01	$< 10^{-3}$	statemate	4240	0.06	0.02	0.002
	1024	0.16	0.01	$< 10^{-3}$		5648	0.34	0.18	0.003
compress	1088	0.3	0.09	0.001	1REG	9760	> 3 hrs	10223.9	0.29
	1488	0.5	0.18	0.001		13760	> 3 hrs	> 3 hrs	0.35
sha	1184	0.04	0.01	0.001	DAP1	28720	12.37	5.27	0.11
	1456	0.55	0.17	0.001		30432	8.4	3.81	0.10
fft1	2128	0.18	0.05	$< 10^{-3}$	susan	14624	1.08	0.68	0.05
	2416	0.2	0.09	0.002		22864	0.34	0.27	0.05
lms	1264	0.14	0.08	0.003	DAP3	44384	> 3 hrs	9938.1	0.83
	1840	0.35	0.09	0.003		47136	54.3	39.22	0.98
edn	2208	0.02	0.001	0.001					
	2736	0.04	0.001	0.001					

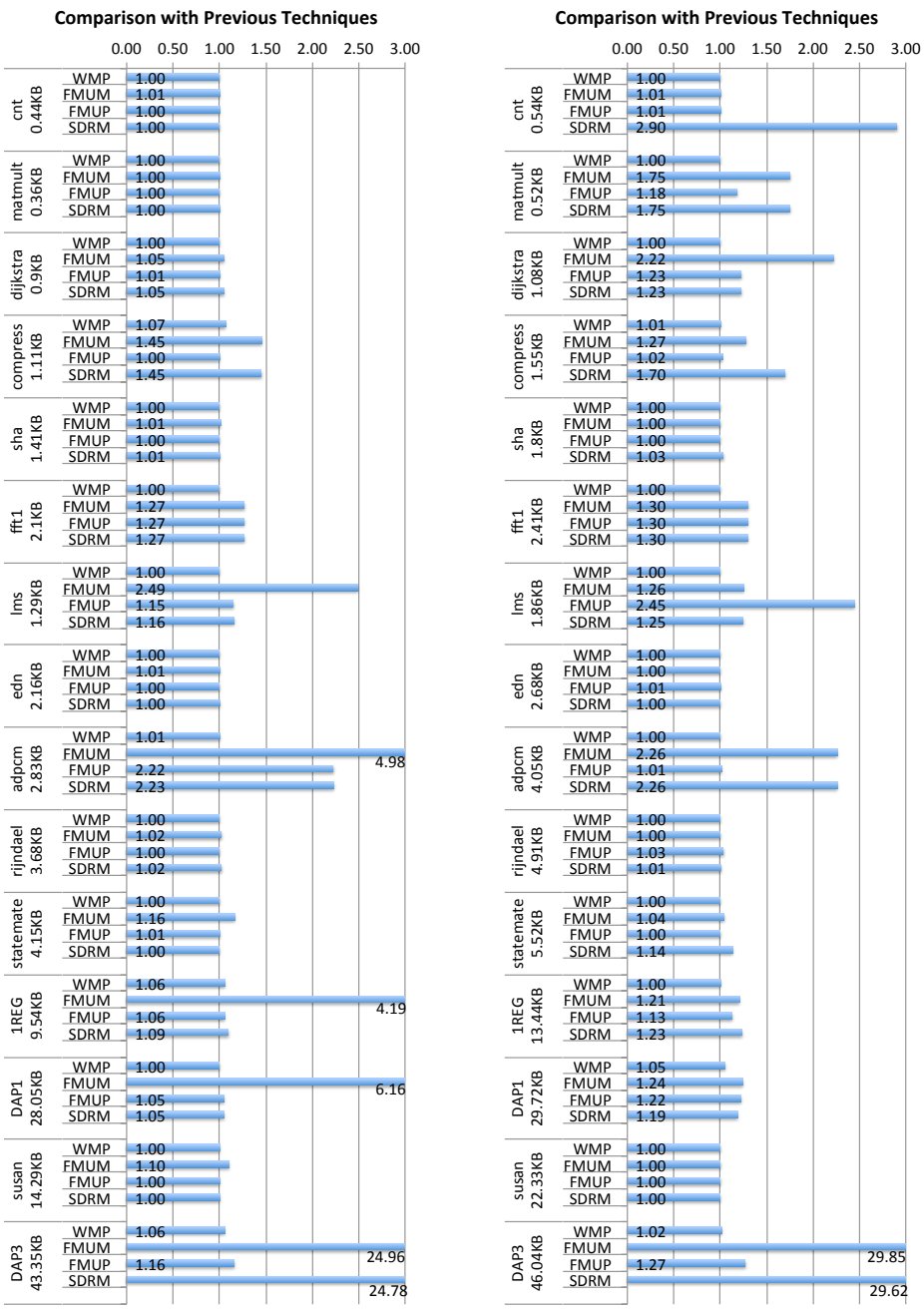
same as the WCETs resulting from using the heuristic. In our experiments, the qualities of all solutions found by the time limit are within 3% of optimality. We observed that even for the cases in which the ILP cannot finish within the time limit, the best objective value found by the solver does not significantly improve any more after 100 seconds. This means that solving the ILP with a reasonable time limit (e.g. 100 seconds) or an optimality range can be a good heuristic itself.

2.6.2 Comparison with Previous Techniques

We evaluate our mapping techniques in comparison with three previous function-level techniques. The two function-level techniques, namely FMUM and FMUP, are proposed by Jung *et al.* (2010). These two take iterative approaches like WMP, but are designed to optimize ACET as their cost function estimates the overall amount of DMA transfers. Another technique called *simultaneous determination of regions and mapping* (SDRM) (Pabalkar *et al.*, 2008) calculates the cost for each function, which is the product of the function size and the number of execution, and iteratively assigns a separate region starting from the function with the highest cost. We first obtained mapping solutions for each benchmark using previous techniques, and then the resulting WCET estimates for the obtained mappings were calculated using the ILP from Section 2.4.

Figure 2.6 compares the WCET estimates for WMP, FMUM, FMUP, and SDRM. These WCET estimates are normalized to the WCET estimates obtained with optimal mappings found by our ILP-based technique. There are four bars for each benchmark, each of which represents WMP heuristic and three previous techniques, respectively. The less the value is, the closer to the optimal solution it is, and the value of 1.00 means it is exactly the same optimal solution as the solution from the ILP. The maximum value of the x-axis is set to 3, and the values of the bars that go beyond 3 are explicitly marked.

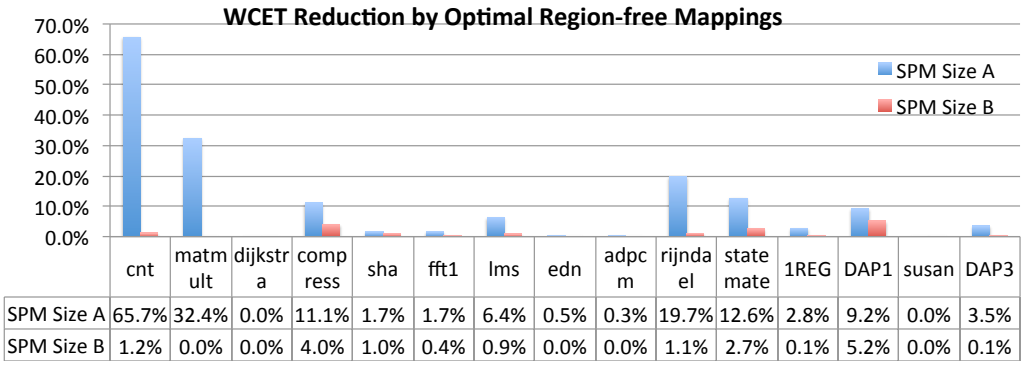
The normalized WCET estimates are always greater than or equal to 1, so no technique outperforms our ILP. Even for the time-limited ILPs used for '1REG' and 'DAP3', the ILP outperforms all other techniques. WMP does not underperform any previous techniques, except 'compress' with SPM size A, in which FMUP happens to find an optimal solution. Since all previous techniques do not have any notion of the WCET, their performance in terms of WCET is rather unpredictable. In fact, we observed counterintuitive results in which the WCET estimate with a larger SPM size B is greater than the WCET estimate



(a) SPM size A (b) SPM size B

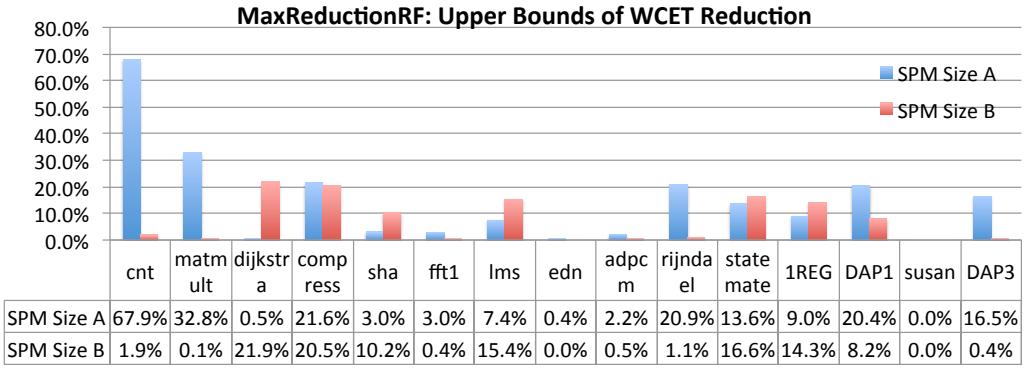
Figure 2.6: Previous function-level techniques are not optimized for the WCET and cannot always find a good solution for the WCET. The normalized WCET estimates using their solutions range from 1 to over 29.

with a smaller SPM size A. For example, in case of FMUP, the WCET estimate for 'lms' with SPM size A is almost twice as high as the WCET estimate with SPM size B, which



	cnt	matmult	dijkstra	compress	sha	fft1	lms	edn	adpcm	rijndael	statemate	1REG	DAP1	susan	DAP3
SPM Size A	65.7%	32.4%	0.0%	11.1%	1.7%	1.7%	6.4%	0.5%	0.3%	19.7%	12.6%	2.8%	9.2%	0.0%	3.5%
SPM Size B	1.2%	0.0%	0.0%	4.0%	1.0%	0.4%	0.9%	0.0%	0.0%	1.1%	2.7%	0.1%	5.2%	0.0%	0.1%

(a) WCET reduction over function-to-region mappings



	cnt	matmult	dijkstra	compress	sha	fft1	lms	edn	adpcm	rijndael	statemate	1REG	DAP1	susan	DAP3
SPM Size A	67.9%	32.8%	0.5%	21.6%	3.0%	3.0%	7.4%	0.4%	2.2%	20.9%	13.6%	9.0%	20.4%	0.0%	16.5%
SPM Size B	1.9%	0.1%	21.9%	20.5%	10.2%	0.4%	15.4%	0.0%	0.5%	1.1%	16.6%	14.3%	8.2%	0.0%	0.4%

(b) Upper bounds of WCET reduction

Figure 2.7: Region-free mapping can help reduce the WCET when interfering functions have to share a region even with an optimal function-to-region mapping.

is not directly shown in the figures due to the normalization. On the other hand, WMP is relatively consistent in reducing the WCET, in that the normalized WCET estimates are kept under 1.07, thanks to its WCET-awareness.

2.6.3 Function-to-region Mappings vs. Region-free Mappings

Figure 2.7(a) shows the reduction in WCET estimates by using region-free mappings compared to the function-to-region mappings found by the ILP. Overall, the WCET reduction ranges from 0% to 65.6%. This reduction comes from the fact that interfering functions can be mapped to disjoint address ranges with region-free mappings, as depicted in an ex-

ample in Figure 2.2. If the SPM size is large enough to accommodate separate regions for all interfering functions, an optimal function-to-region mapping can already find a good enough solution, and RF cannot reduce the WCET significantly. This is why the WCET reduction is higher in SPM size A than B.

In many cases, the reduction is less than 1%. The reason is as follows. For region-free mapping to be able to successfully avoid the reloading at a loading point v, the sum of the size of $fn(v)$ and the size of the largest function in the interference set $IS[v, fn(v)]$ should not be greater than the size of the SPM. If not, the largest function has to share SPM space with $fn(v)$, so region-free mapping cannot do anything. If at least these two functions can be assigned disjoint address ranges, all the inference can be removed by letting all functions in $IS[v, fn(v)]$ share SPM space with each other, but not with $fn(v)$. Although this may increase interference for those functions in $IS[v, fn(v)]$ at other locations, the additional loading costs are calculated in the ILP, so the solver will find an optimal allocation for reducing WCET.

Based on the above insight, we calculate the upper bound of WCET reduction achievable by region-free mapping as follows. We first find the WCEP when the optimal function-to-region mapping is used. Then, we take find all loading points v on the WCEP that satisfy these two conditions: i) v is classified as Always-Miss, and ii) the sum of the size of $fn(v)$ and the size of the largest function in $IS[v, fn(v)]$ is smaller than the SPM size. These are the only loading points where region-free mapping can remove interference and reduce the WCET. We introduce a new value, **MaxReductionRF**, that is the sum of the loading costs at these vertices, which is a rough upper bound of WCET reduction possible by region-free mapping. Figure 2.7(b) shows MaxReductionRF normalized to the WCET in percentage. We can see that the WCET reductions in Figure 2.7(a) have a strong resemblance to the upper bounds in Figure 2.7(b) in many cases such as ‘cnt’ or ‘matmult’. For ‘edn’ and ‘susan’, there is no room to reduce the WCET with region-free mappings.

Even though region-free mapping has larger solution space compared to function-to-region mapping, the ILP for region-free mapping has much less number of constraints than that for function-to-region mapping, which is shown in Table 2.6.1. We observed that linearizing the concept of binary variables in function-to-region mapping using Equation (2.15) and (2.17) causes the number of constraints to increase exponentially as the number of functions (and regions) increases. This difference translates to great reduction in the ILP solving times with large benchmarks.

2.6.4 WCET Reduction over Caches

We evaluate our techniques in comparison with 4-way set associative caches with LRU replacement policy of the same size as the SPMs. We set the associativity to 4 like in many processors in embedded application, such as Renesas V850 or various ARM Cortex series, but we did not observe significant difference in results with different associativity numbers. Although LRU replacement policy is not commercially popular, it is considered to be the the most predictable replacement policy (Guan *et al.*, 2012).

The cache or SPM sizes for each benchmark are chosen as $2^{\lceil \log_2(l) \rceil}$ and $2^{\lceil \log_2(l) \rceil + 1}$, where l is the size of the largest function. The first is the smallest power of 2, greater than the largest function size, and the second is the next power of 2. A cache miss latency takes L (50) cycles (see the beginning of Section 2.6). Although the cache sizes for small benchmarks are much smaller than the instruction cache sizes in modern processors, this makes our experimental setup closer to the real-world situation where code sizes are usually much larger than the instruction cache size.

We implemented the cache analysis algorithm by Cullmann (Cullmann, 2013), which is currently the state-of-the-art and fixes an error of the traditional cache analysis used in industry-leading aiT tool (Ferdinand, 2004). We run the must and may analyses (Ferdinand and Wilhelm, 1999), and new persistence analysis, based on abstract interpretation, on our

generated inlined CFGs. We do not perform virtual loop unrolling (Ferdinand and Wilhelm, 1999), so the first iterations of loops are treated the same as the rest of the iterations. Although as Huber *et al* (2014) discuss, considering local execution scopes, e.g. a function or a loop, may help identifying more number of first-misses, we consider only global execution scope (the whole program) as the persistence analysis algorithm itself does not discuss how to set persistence scopes. We use the same inlined CFGs for both caches and SPMs for fair comparison.

Figure 2.8 compares the WCET estimates. rbILP represents the region-based ILP from Section 2.5.1, and rfILP is the region-free ILP from Section 2.5.3. The cache/SPM size is shown after the name of each benchmark, and data values represent the WCET estimates normalized to the WCET estimates for caches (Thus it is always 1 for caches.). A WCET estimate for caches consists of C (instruction execution time) and L (cache miss penalties), and for SPMs, it consists of C, L (DMA transfer time), and M (management code execution overhead).

The WCET reduction is significant for most of benchmarks in which cache miss handling overhead (L) is very large. One main reason is the lack of link-time optimizations for caches. Instruction addresses are determined in linking stage. Unless a WCET-aware code positioning technique (Falk and Kotthaus, 2011; Um and Kim, 2003; Li *et al.*, 2015) is used, the linker is generally not aware of the impact on the WCET of its decisions or the cache configuration in the target system. For this reasons, function calls may cause many conflict misses in caches, whereas such side effects are actively avoided by code mapping in SPMs. When nested function calls exist in loops, the effect of cache conflict misses can be pronounced. Also, DMA operations for large functions take advantage of burst transfers (Huber *et al.*, 2014). In SPMs, a whole function is loaded at once with only one setup cost, whereas in caches, a cache miss penalty that includes the setup cost is incurred at every cache block boundary. In our experiments with larger cache block sizes such as 32

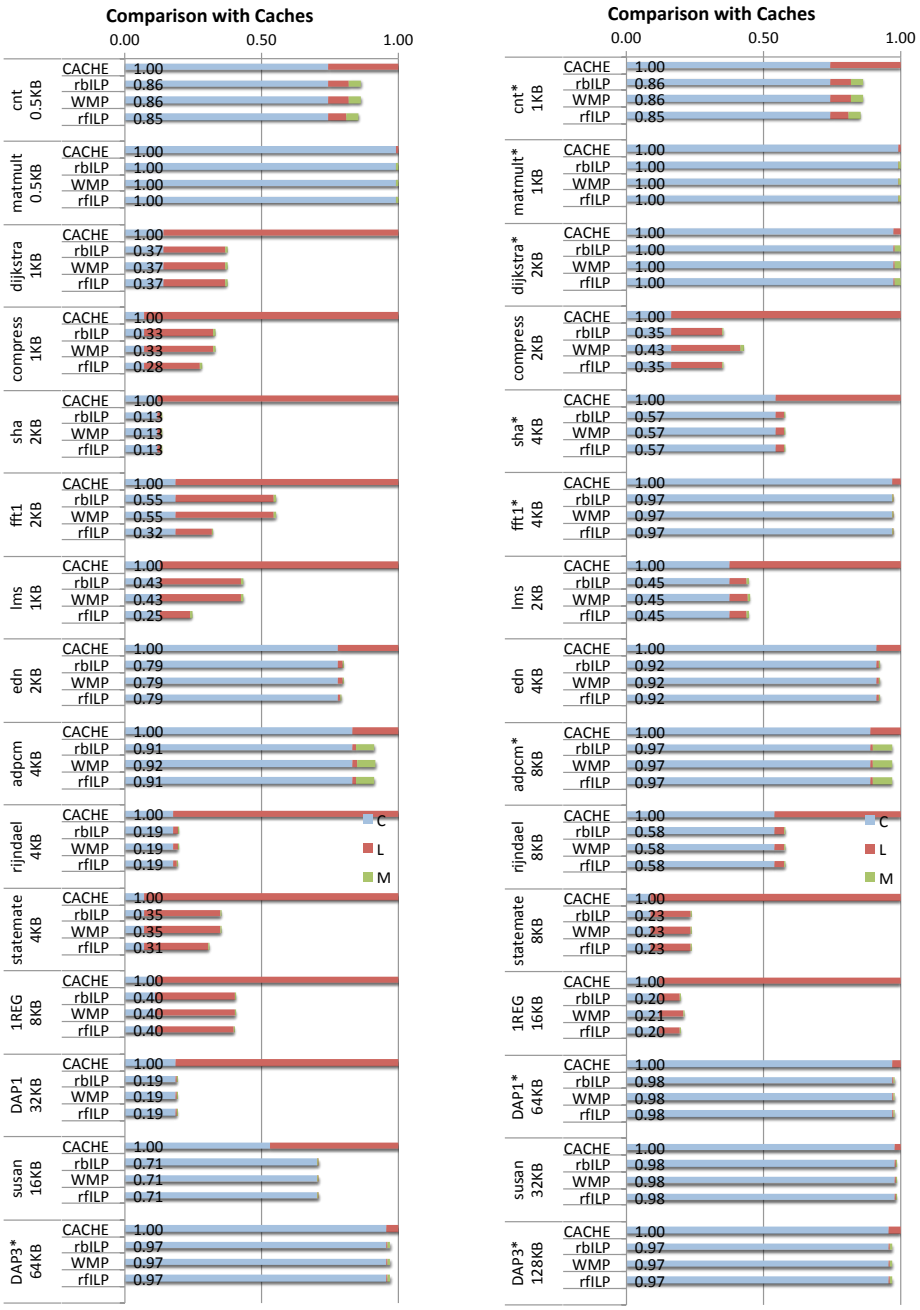


Figure 2.8: Our approaches can significantly reduce the WCETs when the cache suffers from a large overhead for cache miss handling.

bytes and 64 bytes, we did not observe significant differences in the overall trends of the results.

Note that the cache size is larger than the total code size for many cases, e.g., 65KB is larger than the total code size of 'DAP3' [9]. Such cases are marked with * after the name of the benchmark. In these cases, caches show very little cache miss penalties, so the WCET reduction by our techniques is not significant. Nevertheless, WCET estimates with our techniques are always less than or equal to the WCET estimates with caches.

2.7 Related Work

Scratchpad memory (SPM) first gained its popularity in embedded processors mostly due to its advantages over caches in terms of energy consumption and die area (Banakar *et al.*, 2002a). To utilize such benefits of SPM, many researchers initially proposed static management techniques, in which selected data or code is loaded into the SPM once before execution and remains in the SPM during the entire execution. Avissar *et al.* (2002) present an algorithm that considers global and stack data. Steinke *et al.* (2002b) consider global data along with instructions in functions or basic blocks. Both techniques are designed to reduce the overall energy consumption.

Static management techniques have limited capabilities to exploit the locality of large programs. Dynamic techniques, on the other hand, can utilize the locality in different parts of a program, updating the SPM contents during runtime. Kandemir and Choudhary (2002) propose a dynamic data management scheme based on loop transformation and data placement to maximize the data reuse. Many techniques have focused on reducing energy consumption by dynamically copying instructions and/or global variables into the SPM (Steinke *et al.*, 2002a; Verma *et al.*, 2004).

Our approaches are also a type of dynamic management techniques that focus on program code. Several dynamic code management techniques have been proposed (Egger

[9] For caches, we use the original binaries without inserting management code. In 'matmult', 1KB is not larger than the total code size after inserting management code, but is larger than the original binary. Similarly, in 'adpcm', 8KB is larger than the total code size only for caches.

et al., 2006; Janapsatya *et al.*, 2006), all of which aim to reduce the average-case execution time (ACET) or energy consumption. Unlike these techniques, our objective is to reduce the worst-case execution time (WCET) which is a more important metric in hard real-time systems.

Focusing on the time-predictable characteristics of SPMs, researchers proposed various management techniques that aim to reduce the WCET of a given program. Several techniques statically select variables (Suhendra *et al.*, 2005) or instructions (Falk and Kleinsorge, 2009; Prakash and Patel, 2012). There are also dynamic techniques that select variables (Deverge and Puaut, 2007; Wan *et al.*, 2012) or basic blocks (Puaut and Pais, 2007; Wu *et al.*, 2010) to be loaded into the SPMs during runtime. As a dynamic code management technique, our work is more closely related to the latter than the former. The main difference is at the granularity of the management; those techniques work in basic-block-level, whereas ours work in function-level.

Function-level dynamic code management techniques (Baker *et al.*, 2010; Pabalkar *et al.*, 2008; Jung *et al.*, 2010; Bai *et al.*, 2013) were originally proposed for software-managed memory architecture such as Cell processor (Kahle *et al.*, 2005). Here, cores can only access their local SPMs, so every executed instruction must be copied from the main memory to the SPM. Basic-block-level approaches are not applicable in this architecture; they load only selected basic blocks to the SPM and leave the rest of the basic blocks in the main memory. Function-level approaches let every instruction fetched from the SPM by loading a function before it is executed. This larger granularity can benefit from the characteristics of the burst mode DMA operations, as each DMA operation has a setup overhead. On the other hand, the function-level management can have drawbacks such as worse memory utilization due to fragmentation or fetching unnecessary code altogether, compared to the basic-block-level management. We leave the detailed performance comparison between

different granularities as future work and focus on developing WCET-aware function-level code management techniques.

Compared to the previous function-level code management techniques, our approaches have mainly two differences. First, all previous approaches aim to reduce the ACET, not WCET. They calculate the overall overhead of mapping multiple functions into one region considering function sizes and calling patterns, so mappings that incur high overhead scenarios such as reloading functions in a loop are avoided. They, however, fail to consider the worst-case execution scenario of each function nor the control flow within a function that does not have any function call. Our techniques find optimal mappings for reducing the WCET using inlined CFGs which comprehensively contain all information necessary to calculate the WCET. Second, all previous approaches use function-to-region mappings (Pabalkar *et al.*, 2008) in problem formulation, so the solution quality is limited by the abstraction of SPM addresses with regions. We present a technique to map functions directly to addresses, which can further reduce the WCET as we discuss in Section 2.3.2.

In function-level code management, the largest function must fit in the SPM, which can limit its applicability. Kim *et al.* (2016b) present a function-splitting technique to overcome this limitation. Splitting a function can not only enable using smaller SPM sizes but also improve performance by reducing memory footprints of functions.

Recently proposed time-predictable computer architectures such as PRET (Liu *et al.*, 2012), FlexPRET (Zimmer *et al.*, 2014) or MERASA (Ungerer *et al.*, 2010) uses SPM-based memory hierarchies, for which our approach can be used to develop compilers.

Schoeberl *et al.* proposes time-predictable Java optimized processor (JOP) with a method cache (Pitter and Schoeberl, 2010). Method cache is a software-controlled instruction cache in which the entire function (method in Java) is loaded and evicted. This is very similar to our dynamic code management using function-to-region mappings, but their

recent work (Whitham and Schoeberl, 2014) shows that SPMs outperform method caches in terms of the tightness of WCET bounds by static timing analysis.

Gracioli *et al.* (2015) present an extensive survey of worst-case related cache optimization techniques and cache analysis techniques. Cache locking (Plazar *et al.*, 2012; Ding *et al.*, 2014) and partitioning (Liu *et al.*, 2010; Suhendra and Mitra, 2008) can be used to lower WCETs by reducing conflict misses, but the granularity of these techniques is limited by blocks, lines or ways, which may cause a waste of cache space (Whitham and Audsley, 2009). Just like our code mapping techniques, code positioning techniques can be used to avoid conflict misses among functions to reduce WCETs (Falk and Kotthaus, 2011; Um and Kim, 2003; Li *et al.*, 2015), but the amount of reduction is not significant because these techniques are again, limited by the degree of control provided by caches; for example, line sizes and associativity cannot be changed.

Our interference analysis works similarly to the traditional may analysis based on abstract interpretation (Ferdinand and Wilhelm, 1999) with the use of union operation in join function. The semantics of the results are, however, the same as the must analysis in the sense that the interference sets are used to conservatively determine whether a function is guaranteed to be loaded (always-hit) when the interferences on all paths are considered. Although we can find first-misses using initial loading points (see Table 2.2), this is more pessimistic than the persistence analysis (Cullmann, 2013) in terms of identifying first-misses. For example, consider this code: $main()\{f_1();\ f_2();\ \texttt{while}(...)\ f_1();\}$. In this example, *main* calls f_1 and f_2 in a row and then calls f_1 in a while loop. Assume that f_1 and f_2 do not call any other function. The call to f_1 in the while loop is not an initial loading point, but it can still be a first-miss when f_1 does not share SPM space with neither *main* or f_2. Persistence analysis, on the other hand, can categorize the call to f_1 in the loop as first-miss. Developing a more advanced analysis for tighter WCET bounds is part of our future work.

Lastly, dynamic management techniques rely on bounded latencies of DMA operations. Analysis techniques (Kim *et al.*, 2016a) or predictable DRAM controllers (Reineke *et al.*, 2011; Paolieri *et al.*, 2013; Kim *et al.*, 2015) can help with bounding DMA latencies. These are orthogonal to our work and can be used to make the DMA timing more predictable.

2.8 Summary

SPM is a promising on-chip memory choice for real-time systems but needs explicit management. In this chapter, we present three code management techniques that allocate SPM space for functions, with a goal of minimizing the WCET by avoiding DMA operations overhead on the worst-case execution path. Two techniques are based on traditional function-to-region mappings, and the third techniques maps functions directly to SPM addresses. Two limitations with our approaches are not being able to handle recursive function calls, and requiring the largest function to fit in the given SPM. Experimental results with several benchmarks including automotive control applications from industry show that our techniques are highly effective in reducing the WCET. The heuristic algorithm can find a mapping solution within a second for all benchmarks without increasing the WCET more than 6.5% compared to the solution found by the ILP. Results show that region-free mapping can further reduce the WCETs than the optimal function-to-region mappings, but the room for optimization is limited in many cases. Overall, the reduction in the WCET estimates ranges from 0% to 97% compared to previous approaches. Compared to static analysis of caches, the reduction ranges from 0% to 87% when 4-way set associative caches of the same size are used and no link-time optimization for reducing the WCET (Falk and Kotthaus, 2011; Um and Kim, 2003; Li *et al.*, 2015) was applied.

Chapter 3

A COMPARISON OF DYNAMIC CODE MANAGEMENT TECHNIQUES WITH DIFFERENT MANAGEMENT GRANULARITIES

The previous chapter focused on function-level code management, but in the literature, there are other code management techniques work at different granularities, such as basic blocks or regions [1]. These techniques have several fundamental differences regarding WCET-reducing capability or timing predictability. In this chapter, we compare different management techniques with thorough evaluations and discuss their limitations and differences in detail.

3.1 Introduction

Although SPMs have time-predictable characteristics, the predictability and performance of an SPM-based system solely depend on the management technique used in the system. We discuss such impact of different management techniques in this chapter.

There are two different kinds of SPM management techniques: static management and dynamic management. In static management, code/data blocks are allocated in the SPM at loading time before execution, and the allocation does not change throughout the execution. While the allocated code or data are accessed from the SPM, the rest has to be accessed from the slower main memory, which can be a handicap for applications that are larger than the SPM. In dynamic management, on the other hand, such allocation changes during execution to cater for large applications and benefit from the faster on-chip memory.

Traditionally, static management techniques have been widely used for small applications running on simple micro controllers with fast on-chip main memory. Their efficiency,

[1] A region or trace is a straight line of code or a set of basic blocks in a continuous address region (Verma *et al.*, 2004; Whitham and Audsley, 2012)

however, quickly decreases with large applications running on processors with large-but-slow off-chip main memory. As the sizes of real-time applications are rapidly increasing with the integration of features and various regulations regarding safety, security or environmental impact, the demand for efficient dynamic management techniques is on the rise. Increasing on-chip memory sizes is practically limited by die area and size/latency trade off—SRAM access latency typically increases with the increase of the size (Amrutur and Horowitz, 2000).

There have been several proposals on dynamic instruction SPM management techniques that aim to reduce the WCET of a given task (Puaut and Pais, 2007; Wu *et al.*, 2010; Whitham and Audsley, 2012; Kim *et al.*, 2014). These techniques can be categorized according to their allocation granularities as shown in Table 3.1. In basic-block-level techniques (Puaut and Pais, 2007; Wu *et al.*, 2010), code blocks are allocated and loaded at the granularity of basic blocks. These techniques select a set of reloading points and then groups of basic blocks to be loaded into the SPM at each of the reloading points. The rest of the basic blocks are left in the main memory. On the other hand, our own work in function-level technique (Kim *et al.*, 2014) (described in Chapter 2) or Whitham and Audsley's region-level technique Whitham and Audsley (2012) load a whole function or a code region at once before its execution, completely avoiding instruction fetches from the slow main memory. A larger granularity can reduce the overall loading time with faster

Table 3.1: Categorization of code management techniques based on management granularity

Granularity	Techniques
Basic block	Puaut and Pais (2007); Wu *et al.* (2010)
Function	Kim *et al.* (2014) (from Chapter 2)
Region	Whitham and Audsley (2012)

burst mode transfers, compared to smaller granularities, but may degrade memory space utilization with fragmentation. In addition, our function-level technique has a cache-like sophistication that loads a function only when the function is *not* in the SPM. This can improve performance, but requires a separate analysis to obtain tight bounds on load operation timings in the worst-case, similarly to cache analyses (Ferdinand and Wilhelm, 1999; Cullmann, 2013). It also increases the overhead by having more instructions to execute for management, which can be more noticeable on slow embedded processors.

We compare these techniques with thorough evaluations and discuss their limitations and differences in detail. As the differences lead to various crucial aspects such as WCET-reducing capabilities, our detailed comparison can guide future researches on more advanced dynamic management techniques. All previous similar studies only considered static management techniques (Wehmeyer and Marwedel, 2005; Metzlaff and Ungerer, 2012, 2014) or special types of SPMs that are assisted with hardware-logic to reduce the complexity in management code (Metzlaff and Ungerer, 2012, 2014; Whitham and Schoeberl, 2014).

Here we take a basic-block-level technique by Puaut and Pais (Puaut and Pais, 2007) and our function-level technique (Kim *et al.*, 2014) for comparison. To have another allocation granularity between basic blocks and functions, we also present a technique to split functions into smaller partitions in Section 3.3.3. Using various benchmarks from Mälardalen suite (Gustafsson *et al.*, 2010a), MiBench suite (Guthaus *et al.*, 2001), and proprietary automotive control applications from industry, we compare the WCET estimates for different management techniques. Multiple different architectural configurations regarding main memory memory latencies and cache sizes are used for fair comparison.

3.2 Related Work

Traditional management techniques: Most early approaches for managing SPMs focus on reducing the overall energy consumption or average-case execution time either through static management (Avissar *et al.*, 2002; Steinke *et al.*, 2002b) or through dynamic management (Kandemir and Choudhary, 2002; Verma *et al.*, 2004; Egger *et al.*, 2006).

WCET-centric management techniques: Focusing on the time-predictable characteristics of SPMs, researchers proposed various management techniques that aim to reduce the WCET of a given program. Several techniques statically load variables (Suhendra *et al.*, 2005) or instructions (Falk and Kleinsorge, 2009; Prakash and Patel, 2012) to SPMs. Several dynamic techniques have been proposed for data (Deverge and Puaut, 2007; Wan *et al.*, 2012) and code at the granularity of basic blocks (Puaut and Pais, 2007; Wu *et al.*, 2010), code regions (Whitham and Audsley, 2012), or functions (Kim *et al.*, 2014).

WCET-centric dynamic management techniques for instruction SPMs: We focus on dynamic management of code on SPM, in this paper. We compare the differences and evaluate their performance of reducing the WCET of a given program with experiments. In basic-block-level techniques, basic blocks in loops are selected to be allocated to SPM at pre-selected loop pre-headers. An optimal allocation scheme for non-nested loops is presented in Wu *et al.* (2010), but the difference between this approach and the previous heuristic (Puaut and Pais, 2007) is not significant in terms of the resulting WCET bounds. The optimality holds only when every instruction takes one cycle to execute and there is no nested loop. In Whitham and Audsley (2012), a task is partitioned into disjoint regions, each of which is a set of vertices. Each region is smaller than the SPM size and loaded as a whole before execution. They find an optimal partitioning scheme for a given worst-case execution path (WCEP), considering the execution frequency of each entry edge to a region and the lump sum of the sizes of basic blocks in the region. This, however, does

not always lead to an optimal WCET; the WCEP can change after a partitioning decision and non-consecutive basic blocks need separate DMA operations. In our approach from Chapter 2 (Kim *et al.*, 2014), an optimal allocation of functions to SPM addresses is found.This approach can find an optimal allocation that actually minimizes the WCET of a given task, but the evaluation is only done against previous function-level approaches (Pabalkar *et al.*, 2008; Jung *et al.*, 2010) which makes it hard to gauge its effectiveness in line with other WCET-centric management techniques or caches. In this chapter, we compare a basic-block-level approach (Puaut and Pais, 2007) and our function-level approach through extensive evaluations.

Function-splitting techniques: To have a region-level approach, we propose a technique to split functions into smaller partitions, which can be used with our function-level approach Kim *et al.* (2014). Function-splitting can help further reduce the WCET with smaller footprints of functions and better memory space utilization than the function-level technique. Each partition is a single-entry single-exit (SESE) code block (Johnson *et al.*, 1994), such as a loop body or a group of basic blocks. This makes us to have another allocation granularity, between basic blocks and functions. This approach is similar to the techniques used in traditional compiler optimizations called partial inlining or function outlining (Zhao and Amaral, 2005) where a large function is divided into multiple code blocks and then rarely-executed code blocks are outlined as new functions in order to reduce the overhead in function inlining. Compared to call tree partitioning approaches like Whitham and Audsley (2012) that seeks optimality, our approach is an intuitive heuristic in that it splits a function at a fixed set of program points that are likely to be beneficial to bring down the WCET. After splitting a function, it gets a feedback information from WCET analysis and rolls back the splitting decision if the WCET is increased.

Kim *et al.* (2016b) recently presented a function-splitting technique for function-level code management, which is implemented in LLVM compiler framework as a transforma-

tion pass. This heuristic works with two policies that aim to either improve average-case performance or reduce the partition sizes. Compared to this, our function-splitting technique aims to reduce the WCET.

Comparison studies: There are several studies on comparing worst-case performance of on-chip memories, which are *the closest related work to this paper*. Wehmeyer and Marwedel (2005) present WCET-based comparison results of static SPMs and direct-mapped unified caches. They use an algorithm that statically load functions and global variables (Steinke *et al.*, 2002b) for energy reduction and calculates the WCET estimates using aiT tool [2]. As an early work, their worst-case cache analysis is rather primitive; they used only must analysis, without may or persistence analysis (Ferdinand and Wilhelm, 1999). This means only the accesses that lead to always-hits (guaranteed cache hits) are identified, and every other access is assumed to be a miss. In their experiments with 3 benchmarks, the WCET estimates for SPMs are much less than those for caches. The differences between simulated results and the WCET estimates are constant for SPMs regardless of the SPM size whereas they grow large for caches as cache size increases.

Metzlaff and Ungerer (2012; 2014) compare instruction SPM, instruction cache, and a hardware-assisted dynamic instruction scratchpad (D-ISP). D-ISP includes a hardware logic for allocating and evicting functions in the SPM. D-ISP, therefore, does not need a management technique. They use a static management technique (Falk and Kleinsorge, 2009) for selecting basic blocks or functions to load into SPMs. Assuming fully-associated caches with LRU replacement policy, they use the worst-case cache analysis algorithm based on abstract representation (Ferdinand and Wilhelm, 1999). The worst-case analysis of D-ISP is based on data flow analysis using concrete representation that records all possible states of D-ISP. This approach leads to the most precise analysis results, but is not scalable for large applications due to state explosion. In their experiments with 5 bench-

[2] AbsInt GmbH, http://www.absint.com/ait/

marks, D-ISP shows overall the lowest WCET bounds, and caches performs the worst. In static management, allocating at the granularity of basic blocks is only slightly better than function-granularity allocation due to better space utilization. The main reason behind D-ISP's lower WCET bounds is that they assumed unified main memory that has both code and data, which introduces the interference, thus an extra delay, in main memory accesses. As a function-level approach, D-ISP loads all instructions in a function before its execution, and this eliminates the interference in main memory accesses.

Whitham and Schoeberl (2014) compare dynamic instruction SPM and Method cache (Pitter and Schoeberl, 2010), which is just a different name of D-ISP. For SPMs, a region-level dynamic management technique (Whitham and Audsley, 2012) is used. In experiments with synthetic programs, SPMs tend to have lower WCET bounds than Method caches due to the pessimism of worst-case analysis of FIFO replacement policy in Method caches.

Bounding DRAM access latencies: Dynamic management techniques rely on bounded latencies of DMA operations. Analysis techniques (Kim *et al.*, 2016a) or predictable DRAM controllers (Kim *et al.*, 2015) can help with bounding DMA latencies.

3.3 Dynamic Code Management Techniques

In this section, we briefly explain the code management techniques used in our evaluation, one in basic block granularity (Section 3.3.1), and the other in function granularity (Section 3.3.2). Then, we present our function-splitting technique in Section 3.3.3 which is combined with function-level technique to make a region-level technique.

3.3.1 BL: Basic-block-level Approach

We use a technique presented by Puaut and Pais (2007), which is arguably the most extensively studied basic-block-level dynamic code management technique in the literature.

The algorithm works in two-steps, similarly to all other basic-block-level or region-level techniques Whitham and Audsley (2012); Wu *et al.* (2010). In the first step, the algorithm selects a set of basic blocks as reloading points where direct memory access (DMA) instructions to load code blocks *can* be inserted. For each loop, it estimates the reduction in the WCET after allocating the most frequently accessed basic blocks along the worst-case execution path (WCEP) in the particular loop to the SPM. The number of the allocated basic blocks are assumed to be as many as the SPM size can allow. The rest of the basic blocks are accessed directly from the main memory. The DMA instructions are assumed to be inserted at every loop pre-header of the loop, as it is guaranteed to be executed before all basic blocks in the loop. Based on the cost of DMA operation to load the allocated basic blocks, it may not be profittable to use the SPM for some loops, meaning that the WCET would rather increase because of the loading overhead. Only pre-headers for profittable loops are selected to be reloading points.

Whether or not a loading operation actually takes place at a particular reloading point is determined in the next step where the algorithm decides which blocks to load at each reloading point. Algorithm 6 shows the procedure, which we borrowed from Puaut and Pais (2007) with minor updates for readability. The algorithm first finds the WCET and WCEP (line 2) and finds the N-most frequently executed basic blocks on the WCEP (line 3). The found basic blocks are then removed from the list of candidate basic blocks, $ToBePlaced$. For each of the N basic blocks, the algorithm finds corresponding reloading points that dominate it (line 4-5) and inserts loading instructions for the basic block into each of the reloading points, rp (line 6-7). If this makes the sum of the sizes of basic blocks being loaded at the reloading point greater than the SPM size, the reloading point (rp) is removed from $ReloadPoints$ and the basic block (bb) is removed from $ToBePlaced$. After processing N basic blocks, the WCET is reevaluated (line 8) and the above process repeats for

Algorithm 6: Algorithm for selecting SPM contents in the basic-block-level approach by Puaut and Pais (2007)[3]

Input: List of basic blocks ($ToBePlaced$), Reloading points selected in the first step ($ReloadPoints$), The number of basic blocks to select each iteration (N)

Output: Modified program with DMA instructions inserted

```
1  (WCET, WCEP) = evaluateWCET()
2  ListBB = SelectMostBeneficialBB(ToBePlaced, N)
3  while |ListBB| ≠ 0 do
4      for each bb in ListBB do
5          ListRP = getReloadPoints(bb, ReloadPoints)
6          for each rp in ListRP do
7              Insert DMA instructions for bb at rp
8      (WCET, WCEP) = evaluateWCET()
9      if WCET > WCET_previous_iteration then return
10     ListBB = SelectMostBeneficialBB(ToBePlaced, N)
```

the next N-most frequently executed basic blocks (line 10). The algorithm stops when the WCET is increased or all loop basic blocks are selected ($|ListBB| = 0$).

In our implementation and evaluation in the remaining sections, we make the following changes to further improve the performance and the accuracy of the technique.

First, for nested loops, we consider the pre-headers of the outermost loop as reloading points for all loops in the loop nest. This is because loading of basic blocks for an inner loop can corrupt the contents of the SPM for the outer loop, as shown in Figure 3.1. Unless DMA instructions are duplicated at all exit blocks of inner loops, the task cannot execute

[3] ©2007 IEEE. Isabelle Puaut and Christophe Pais, "Scratchpad Memories vs Locked Caches in Hard Real-time Systems: A Quantitative Comparison", In Proceedings of the Conference on Design, Automation and Test in Europe, Apr., 2007.

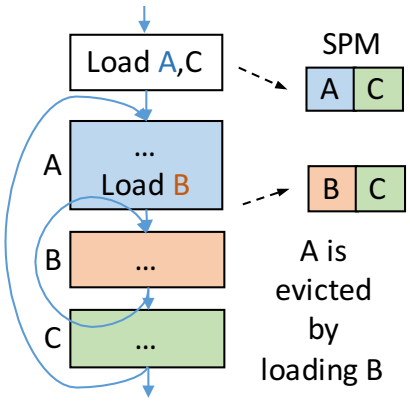
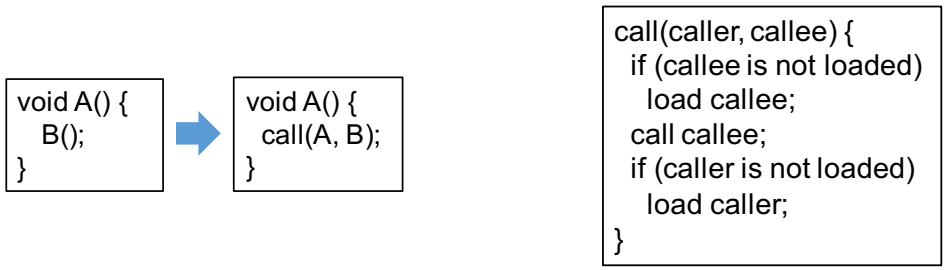


Figure 3.1: In a basic-block-level technique, reloading for an inner loop can evict the basic blocks loaded for an outer loop.

correctly due to the corrupted memory. Although the original paper does not mention how nested loops are handled, we consider this is the most reasonable solution. This is efficient for small loops that can fit in the SPM because all loadings take place only once before entering the loop nest.

Second, regarding the cost of reloading, we use a more realistic cost model. While the original technique only considers the transfer time, we consider the execution time of DMA instructions themselves, such as setting up arguments for source address, destination address, and transfer size, and also initiating a DMA operation. Also, we assume a separate DMA operation takes place for each of the consecutive address ranges to be loaded. For example, in Figure 3.1, `Load A,C` must be done in two separate DMA operations as basic block A and C lie in disjoint memory regions, whereas `Load A,B` can be performed by one DMA operation. All basic-block-level techniques including the work of Wu *et al.* (2010) only consider the transfer time for the lump sum of the basic block sizes.

Third, we consider the overhead of branches or fall-through paths between the SPM and the main memory. A single branch instruction can only have a limited displacement range, and the distance between an SPM address and a main memory address can be out of the range. We add the additional cost of a long jump with direct addressing between every



(a) Code transformation (b) Management function

Figure 3.2: An illustrative implementation of function-level code management

branch or fall-through path between SPM and main memory, as in the work of Falk and Kleinsorge (2009).

Fourth, the original algorithm finishes when the WCET increases as shown at line 9 in Algorithm 6. This leads to the final WCET greater than the WCET found in the previous iteration. To prevent this, we roll back the code modification done for the N basic blocks selected to be loaded in that iteration. Thus, we ensure that the final WCET is the best WCET found by the algorithm.

We refer to this modified approach as BL in this chapter.

3.3.2 FL: Function-level Approach

Function-level code management, which we used in Chapter 2, loads instructions at the granularity of functions around each call site, which enables fetching all instructions from the SPM. This method of code management originates from the code management techniques (Pabalkar *et al.*, 2008; Baker *et al.*, 2010; Jung *et al.*, 2010; Bai *et al.*, 2013) for IBM Cell BE procesor (Kahle *et al.*, 2005) where each of the accelerator cores can only fetch instructions from its local SPM. Here the question is not “what to load", but “where to load", i.e. the allocation of functions. In Chapter 2, we presented function allocation techniques to minimize the WCET of a given program.

Unlike basic-block-level approaches, this approach involves *conditional DMA operations*; a function is loaded by a DMA operation only when it is not loaded into the region. Figure 3.2(a) illustrates a possible software implementation of this using an overlay manager function, shown as `call`. It shows that a simple function call to B is transformed to a call to the management function `call` with the caller and callee function information. Function `call`, shown in Figure 3.2(b), checks the SPM state, loads the callee function if necessary, and then calls the callee. Loading a function changes the state of the SPM such that the loaded function is marked as loaded while all functions sharing the SPM space with the function is marked as invalidated. The management function also makes sure the caller function is loaded again before returning to the caller function. This function permanently resides in the SPM and ensures that the program executes correctly even if loading the callee overwrites the caller.

The basic-block-level approach (BL) from the previous section is a heuristic regarding the selection of basic blocks. To have a fair comparison in terms of WCET-reducing capability, we use the WMP heuristic from Chapter 2, instead of the ILP-based optimal allocation techniques, and refer to it as FL.

3.3.3 RL: Splitting Functions into Partitions

In function-granularity approaches, a whole function is loaded at once to a contiguous space in the SPM. This large granularity, compared to basic blocks or cache lines, is intended to keep the overhead of executing additional instructions low by loading as many instructions with locality as possible. This, however, can be an overhead since large functions may not be able to be loaded into a fragmented memory space. Moreover, this imposes a limitation that the largest function in a task must fit in the SPM.

In this section, we present a technique to split functions into partitions. Considering a partition as a function, we can use the aforementioned function-level approach to allocate

Algorithm 7: Function-splitting algorithm

Input: Inlined CFG (G), SPM size ($SPMSIZE$)

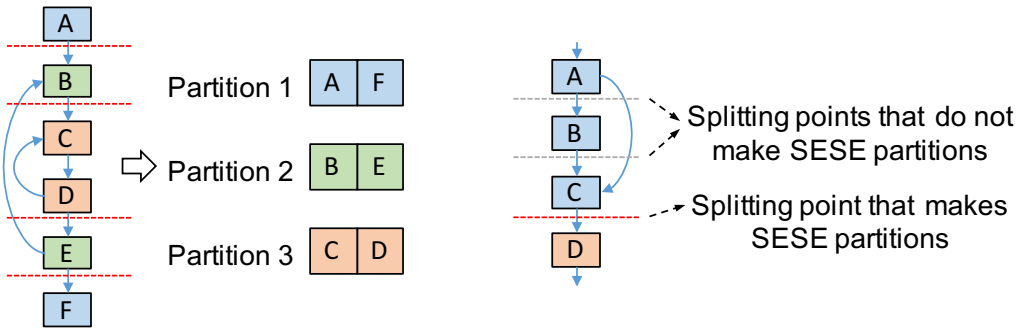
Output: A modified inlined CFG with functions split

```
1 for each function f that is larger than SPMSIZE do
2     G = split(G, f)
3 (WCET, Allocation) = CM(G, SPMSIZE)
4 for each function f in descending order of sizes do
5     G' = split(G, f)
6     (WCET', Allocation') = CM(G', SPMSIZE)
7     if WCET' < WCET then
8         G = G'
9         (WCET, Allocation) = (WCET', Allocation')
```

SPM space to partitions. This makes it a region-level management approach like the work of Whitham and Audsley (2012), and thus we refer to this as RL in this chapter.

Recently, Kim *et al.* (2016b) presented a function-splitting technique for function-level code management. Focused on the average-case performance, this technique cannot always reduce the WCETs. Another difference is that the technique by Kim *et al.* actually creates a function for each partition and inserts new function calls at branches between different partitions. Compared to this, our function-splitting technique only creates logical partitions, and there is no function call at branches between partitions. A partition is just a logical unit of loading that can be loaded separately from the parent function, and at the branches between two partitions, certain code modifications, described later in this section, need to be inserted.

Algorithm 7 shows the pseudocode of our function splitting algorithm. It takes as input, an inlined CFG of a task G and $SPMSIZE$. We first split all functions larger than the SPM



(a) Splitting a nested loop (b) Splitting a non-loop code

Figure 3.3: Examples of our function-splitting scheme. Each loop body forms a separate partition, and each partition forms an SESE region.

(line 1-2) using function *split* which modifies G. Then, we obtain an initial WCET using a function-level code management technique [4] from Chapter 2, represented as function *CM* (line 3). It outputs a tuple of the WCET bound $WCET$ and an allocation of functions to SPM space $Allocation$. We split each function (line 5) and keep the change in G only when it can reduce the WCET (line 6-9).

The procedure for splitting a function (noted as *split*) is sketched as follows. In an inlined CFG G (see Section 2.4.1), each basic block v is annotated with its function identifier, $fn(v)$. A partition is considered as a function in F, and each time a new partition is created, the set F is expanded with the new partition. Every basic block v in a partition p split from a function f are assigned the function identifier $fn(v)$ with the value p, instead of f.

A function is split at basic block boundaries. Each loop body form a separate partition, and for nested loops, the body of each loop in each level forms a partition. This prevents calling management functions in a loop repeatedly. The remaining part of a function after taking loops away forms a partition (e.g. {A,F} in Figure 3.3(a)), called non-loop partition. If a non-loop partition is larger than $x\%$ of the original size of the function, we try to split the partition into two such that both of the resulting partitions are single-entry single-exit

[4] As mentioned in the previous section, WMP heuristic is used in this work.

(SESE) regions (Johnson *et al.*, 1994), as shown in Figure 3.3(b). In our evaluation, we empirically set x as 75. We try every splitting point (a basic block boundary) that can form two SESE regions, and select the one that makes the sizes of resulting partitions as equal as possible. If any of the resulting partition is still larger than $x\%$ of the original size of the function, we roll back the splitting decision and keep the original non-loop partition as is. This is a heuristic way of keeping the overhead of function splitting low. The overhead comes from executing additional code for management at every branch that jumps into a different partition (e.g. function call). Forming SESE partitions keeps the overhead low by limiting the number of inter-partition branches.

Splitting a function involves the following code modifications, since partitions may no longer be in a contiguous memory address range after loaded to the SPM: 1) every fall-through control flow at the end of a partition is explicitly redirected to the next partition with a unconditional branch as shown in Figure 3.4; 2) every PC-relative branch instruction whose target address is outside the partition is modified to use direct addressing; 3) if any PC-relative load/store instruction accesses a constant in a literal pool outside the partition, the constant is duplicated at the end of the partition so that it becomes accessible using PC-relative addressing as shown in Figure 3.5. In an inlined CFG, these code modifications are seen as addition of new basic blocks and edges, or the increase in the size of corresponding basic blocks.

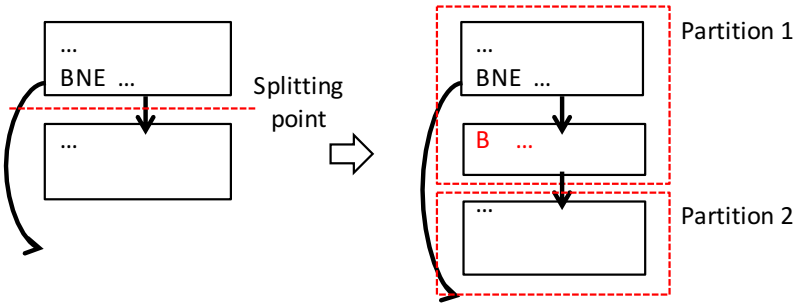


Figure 3.4: Fall-through paths across partition boundaries are redirected by explicit unconditional branches.

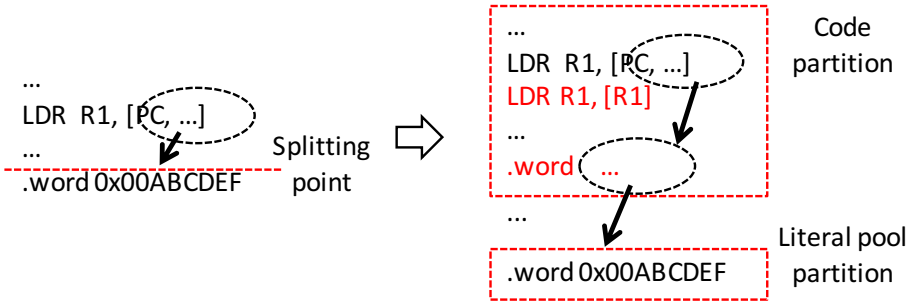


Figure 3.5: A literal accessed by a PC-relative load/store instruction in a partition split from a function, the literal is duplicated and moved to the partition so that it is always accessible from the instruction in the partition.

3.4 Qualitative Comparison

In this section, we discuss the differences of three management techniques from the previous section and their implications in several different aspects. The basic-block-level approach, function-level approach, and region-level approach are denoted as BL, FL, and RL, respectively.

3.4.1 WCET Analysis

A WCET analysis involves microarchitectural timing analysis, and here we focus on instruction memory access timings. Instruction memory accesses are present either in the form of instruction fetches or DMA load operations. It is trivial to analyze instruction fetch times because there is no variability in instruction fetch times for all techniques. In FL and RL, all instructions are fetched from the SPM whereas in BL, only selected instructions are in the SPM, but the selection is fixed at compile-time.

BL makes it simple to analyze the timing of DMA operations, too. In BL, DMA operations take place every time the control reaches a reloading point, without checking the state of the SPM. On the other hand, the cache-like management scheme in FL or RL performs load operations only when the code to execute is not in the SPM. Since the state of the SPM depends on execution history, it requires us to use a data flow analysis, as we described in

Section 2.4.2 and 2.4.3, similar to cache analysis Ferdinand and Wilhelm (1999), to predict if a DMA operation takes place or not at an entry to functions or partitions in the worst-case. Note that RL is not a representative in all region-level techniques. Another region-level technique Whitham and Audsley (2012) uses deterministic load operations.

The analysis for FL and RL can be still simpler than that for caches. Their replacement policy is analogous to direct-mapped caches, which is much simpler to analyze than set-associative caches, especially with non-LRU replacement policy (Guan *et al.*, 2012). All load operations are explicit in the source code, and the explicit management of SPMs makes it natural to partition the SPM and assign a completely private partition to each task. This eliminates cache-related preemption delays (CRPD) (Altmeyer *et al.*, 2011) regardless of what scheduling policy is used. A detailed comparison under multi-tasking environments is out of the scope of this work, and we leave it as future work.

3.4.2 SPM Size Limitations

BL does not have a limitation on the size of SPM for executing any task because it allows instructions to be fetched directly from the main memory. Strictly speaking, a basic block must fit in the SPM, but it is reasonable to assume that a basic block is smaller than typical SPM sizes. It is also trivial to split a basic block into smaller basic blocks by adding an explicit fall-through branch in between. It is obvious that with most basic blocks left in the main memory, the performance is limited by the overhead of accessing the slow main memory.

The size limitation in FL and RL is more strict, as they load a whole function or partition to the SPM. To execute a task on an SPM using FL, the SPM size must be at least as large as the largest function in the task. Function-splitting eases this limitation in RL, but the splitting technique is not capable of controlling the size of each partition. For example, if a task has a loop whose body is larger than the SPM size, we cannot execute the task on

the SPM using RL. It is possible to split the partitions further into smaller partitions, but it may cause performance degradation due to higher management cost from inter-partition branches as we discussed in Section 3.3.3.

This difference may not stand out in a single-tasking environment, but may become important in a multi-tasking environment where tasks can be allocated private partitions. While any inter-task partitioning scheme is possible for BL, FL or RL requires the partition for each task to be at least as large as the largest function/partition, which can limit the number of tasks executable on a core.

3.4.3 Management Efficiency

Different characteristics of the techniques can affect their WCET-reducing capabilities. We leave more detailed quantitative comparison to the next section.

As discussed in Section 3.3.1, BL has a limited ability to exploit locality of large nested loops and can leave a large part of a loop in the main memory. This is because only loop pre-headers are considered as reloading points for each loop, which are executed before the execution of the loop only once. As reloading for inner loops can corrupt the SPM contents for the outer loop, all loading needs to be performed once prior to executing the outermost loop. This simple management scheme is actually beneficial for small tasks where the entire body of most nested loops can fit in the SPM because of little management overhead; due to performing the load operation only once before entering the loop and not checking the SPM state. Compared to this, FL and RL take a more sophisticated cache-like approach that makes sure every code block is loaded after checking the SPM state. This can avoid unnecessary DMA operations but lead to a higher management overhead due to checking the SPM state. Here the focus of management techniques is to find a *good* mapping that avoids the interference among functions/partitions on the worst-case execution path, so the overhead of DMA operations is minimized.

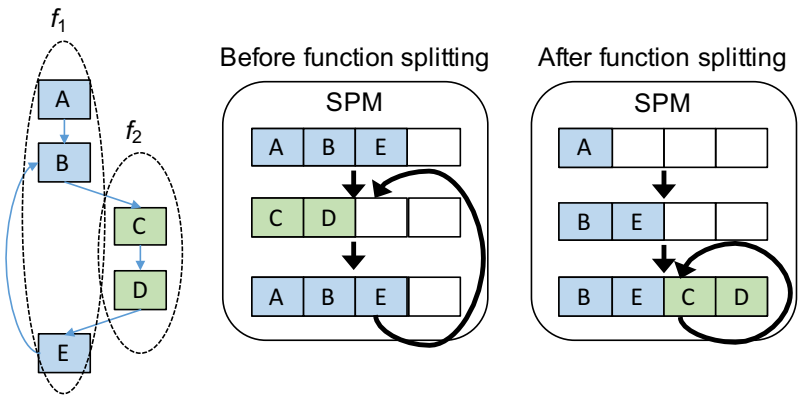


Figure 3.6: Function splitting can eliminate reloading operations in a loop by reducing the memory footprints of code during the execution of the loop.

Note that if a loop nest in a task is too large to fit in the SPM and BL has to leave much part of loop body in the main memory, then the task is not even executable with FL; the function that has the loop must be larger than the SPM. It can be possible, however, to use RL to execute the task since it makes the loop body of each loop as partition. Although it may lead to reloading the partitions for inner loops at every iteration of an outer loop, it can still be more efficient than fetching a lot of instructions from the main memory every iteration, thanks to burst mode DMA accesses.

One problem with FL and RL is the possibility of loading instructions that will not be executed. BL can also load basic blocks that will not be executed, but the problem is more noticeable in FL and RL for their large granularities of allocation. Infeasible path detection (Suhendra *et al.*, 2006) can help trimming down part of code that is unnecessary for loading.

Compared to FL, splitting functions in RL can reduce the WCET by reducing the memory footprints of functions. Consider an example illustrated in Figure 3.6. Function f_1 (composed of blue basic blocks) has a loop in which f_2 (composed of green basic blocks) is called. Assume that every basic block is of the same size, and the SPM size is of 4 blocks. In function-level, f_1 and f_2 needs to be reloaded at every iteration due to the size

Table 3.2: Benchmarks used in our evaluation

	Original code size	# of BBs	# of functions	Source
fft1	3320B	376	6	WCET
lms	3584B	256	8	WCET
edn	4288B	56	9	WCET
adpcm	7364B	295	17	WCET
rijndael	9152B	240	7	MiBench
statemate	10280B	365	8	WCET
1REG	26572B	754	28	Proprietary
DAP1	35300B	1951	17	Proprietary
susan	49848B	962	19	MiBench
DAP3	54852B	2728	28	Proprietary

limitation. If f_1 is split into {A} and {B,E}, all code that is executed in the loop can remain loaded in the SPM.

Note that the function-splitting scheme in RL is intended to demonstrate the impact of smaller allocation granularity and is not necessarily an optimal way of partitioning. Approaches such as (Whitham and Audsley, 2012) focus on finding an optimal partitioning scheme but use simple load operations as in BL.

3.5 WCET-Based Quantitative Comparison

In this section, we quantitatively compare the WCET-reducing capabilities of management techniques by calculating WCET bounds of various benchmarks under several different architectural configurations.

3.5.1 Experimental Setup

We use various benchmarks from the Mälardalen WCET suite (Gustafsson *et al.*, 2010a) and MiBench suite (Guthaus *et al.*, 2001), and three real-world proprietary automotive powertrain control applications from industry, which are the same benchmarks used in Chapter 2. Table 3.2 show the list of benchmarks and their relevant information. From the benchmarks we used in Chapter 2 (shown in Table 2.3), we use the ones whose code size is at least 3 kB. This makes us exclude 5 smallest benchmarks, which are too small to be used in evaluation of dynamic code management techniques or to test the effect of function-splitting. We consider only user code; a library call is assumed to take the same cycles as a single arithmetic instruction without dependencies.

To calculate the WCET bounds for BL, we use implicit path-based enumeration technique (IPET) (Li and Malik, 1995) with loop bounds found with profiling. For FL and RL, we use our ILP-based WCET analysis from Section 2.4 to calculate the WCET estimates for the code mapping solutions (functions/partitions to SPM regions) found by WMP heuristic from Section 2.5.2. We assume that every instruction, except DMA instructions, takes 1 cycle to execute and all data are accessed from a separate data SPM, without any contention with the accesses to the main memory or the instruction SPM. These assumptions are only for simplifying the evaluation and intended to focus on the capabilities of code management techniques. Our implementation of BL and RL (along with FL) is uploaded the source code in a public domain [5].

All SPM accesses take 1 cycle. The latency of accessing x bytes from the main memory ($AT(x)$), either by DMA operations or instruction fetches, is modeled as follows.

$$AT(x) = S + \lceil x/B \rceil \tag{3.1}$$

[5] `https://github.com/yooseongkim/SPMCodeManagement`

where S denotes setup time that is constant regardless of the access size, and B denotes bandwidth. We set S as 20 cycles and B as 4 (32 bits/cycles) in our baseline experiments. Since we are using ARMv4 ISA, every instruction is 4-byte wide, thus fetching an instruction from main memory takes 21 cycles. We change these parameters later in this section to model different execution environments.

To model the overhead of DMA operations, we consider a realistic scenario in an ARM core with an SPM or a tightly coupled memory (TCM). DMA operations in ARM are controlled by updating/reading register c11 of coprocessor CP15, which can be performed by MCR/MRC instructions [6]. We omit a detailed description of how to perform DMA instructions for brevity and give a rough estimation of the number of instructions needed for a DMA operation. A DMA operation requires at least 5 steps: setting up source address (SA), destination address (DA), transfer size (TS), initiation (IN), and polling to detect the completion (CD). It requires 2 instructions for each of SA, DA, and TS, one for loading a 32-bit word and then one MCR to update the register. IN can be done by 1 MCR instruction, and CD can be performed by a loop composed of 3 instructions, 1 MRC for reading channel status, 1 compare, and then 1 branch to repeat. Since each instruction takes 1 cycle, the loop bound of the polling loop is $(\lceil AT(x)/T(CD) \rceil + 1)$ where $T(CD)$ is the duration to finish operation CD.

We add the above DMA overhead for each set of consecutive basic blocks to be loaded at every reload point in BL. We implemented the management code for FL/RL as illustrated in Section 3.3.2 in ARM assembly, and calculated the number of additional instructions that are execute for management. We take into account this overhead in the WCET analysis and subtract the SPM size accordingly to keep the `call` function.

For BL, the algorithm takes N as an input parameter, which denotes the number of most-frequently executed basic blocks on the WCEP to select per iteration. This parameter

[6] ARM Technical Reference Manual, http://infocenter.arm.com

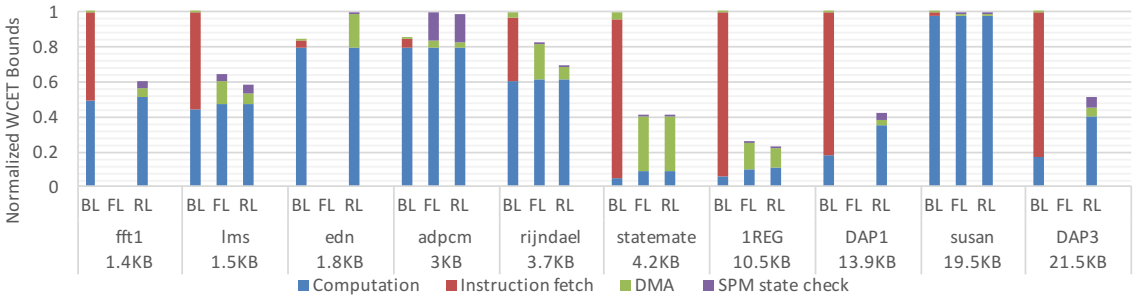


Figure 3.7: Baseline results when $AT(x) = 20 + \lceil x/4 \rceil$ and the SPM size for each benchmark is 40% of code size. Overall, RL outperforms other techniques with a better utilization of SPM space. BL performs poorly when it cannot allocate the whole loop body to the limited-size SPM.

determines the frequency of WCET analysis as the algorithm does not evaluate the WCET while it is allocating N basic blocks in the inner loop. We observed that using a too small value for N often causes premature termination of the algorithm and results in very high WCETs. Using too large values also leads to very high WCETs because it makes the algorithm almost blindly make decisions for many number of basic blocks at once, removing the WCET-awareness from the algorithm. In experiments with various values for N parameter, we observed overall the lowest WCET bounds when N was set 10% of the total number of basic blocks in a given task. Puaut and Pais (2007) also used the same in their original publication.

For each benchmark, the size of SPM is set to 40% of the original code size of the benchmark. All memory sizes are a multiple of 128. For comparison with FL, the SPM size must be at least as large as the size of the largest function in a benchmark. Given this constraint and a limited set of benchmarks, memory sizes are chosen to stress-test all techniques for effective evaluation. Using a fixed SPM size of, say 2KB, is not useful for evaluation purposes, since all techniques will perform perfectly for small benchmarks whose code size is smaller than 2KB and very poorly for large benchmarks whose code size is much larger than that.

3.5.2 Baseline Results

Figure 3.7 compares the WCET bounds for BL, FL, and RL with the default memory access time model in Equation 3.1. WCET bounds are divided by the maximum value among three WCET bounds of each benchmark. A WCET bound for BL is composed of computation time (execution time excluding memory access times and management code execution time) and instruction fetch time from the main memory, and DMA operation time to load instructions from the main memory to the SPM. Similarly for FL and RL, a WCET bound is composed of computation time, DMA operation time, and SPM state checking time. All instruction fetches are from SPM and do not incur any wait cycle in FL and RL. SPM state checking time is the total overhead that is in addition to simple DMA operations, which includes not only checking the region state but also maintaining region state and passing caller information, etc.. The SPM size for each benchmark is below its name.

Overall, RL shows the lowest WCET bounds except for a few of cases where BL outperforms the other two. FL never outperforms RL, and in most cases the RL and FL show comparable performance. We could not use FL for `fft1`, `edn`, `DAP1`, and `DAP3` due to its size limitation; the largest function does not fit in the SPM in these benchmarks. Function-splitting in RL enables the execution of these benchmarks.

BL's poor performance in the results mainly comes from its limitations in selecting reload points. BL loads basic blocks in loops before executing the loops at their pre-headers. When the SPM size is not large enough to hold the entire body of a loop, the basic blocks left in the main memory can cause significant delays (represented as large red bars) with instruction fetches from the main memory. In `fft1`, `lms`, `statemate`, `1REG`, `DAP1`, and `DAP3`, there are large loops on WCEP. BL allocated as many basic blocks to the SPM as possible in all cases, but large portions of loops have to be left in the main mem-

ory. Compared to this, FL benefits from burst transfers; even when entire functions have to be reloaded repeatedly in a loop, it can be still cheaper than executing many instructions directly from the main memory. This case is shown in `statemate` where FL has a large green bar but still its WCET bound is much less than that of BL [7]. Also, function-splitting in RL greatly helps in case of nested loops, which makes the body of each loop loaded separately. This reduces the memory footprints of functions with loops and can further reduce the WCET, which is demonstrated in all other benchmarks where RL outperforms BL.

The characteristics of `edn` and `adpcm` lead to the worst-case scenarios for both FL and RL, whereas they are the best-case scenario for BL. In `edn`, all functions/partitions are mapped to the same region because of the limited SPM size. Due to the direct-mapped cache-like management scheme, entire functions/partitions have to be reloaded every time. On the other hand, it has loops that are small enough to fit entirely in the SPM and have very high loop bounds. Even being executed from the SPM, the loops take the most of the execution time, and the main memory accesses for non-loop code are insignificant. In `adpcm`, FL and BL manage to find a mapping that can avoid the most of reloadings in loops, but SPM state checking code has to be repeatedly executed in a loop with a very high iteration count. While BL can allocate most loop basic blocks to the SPM, the overhead of SPM state checking dominates the WCET in FL and BL.

`susan` has the characteristics of an ideal scenario for all techniques. Only small part of code is executed with nested loops with high iteration counts, so the given SPM size is large enough to hold all of the code executed on the WCEP. Also, even though BL is a greedy heuristic that stops allocating basic blocks whenever the WCET increases, it was able to allocate as many loop basic blocks as SPM size permits in all benchmarks.

[7] In `statemate`, the greedy heuristic in BL prematurely stops allocating basic blocks after the WCET increases due to the overhead from long jumps. FL and RL largely outperformed BL even when we force-allocated as many basic blocks as possible to the SPM.

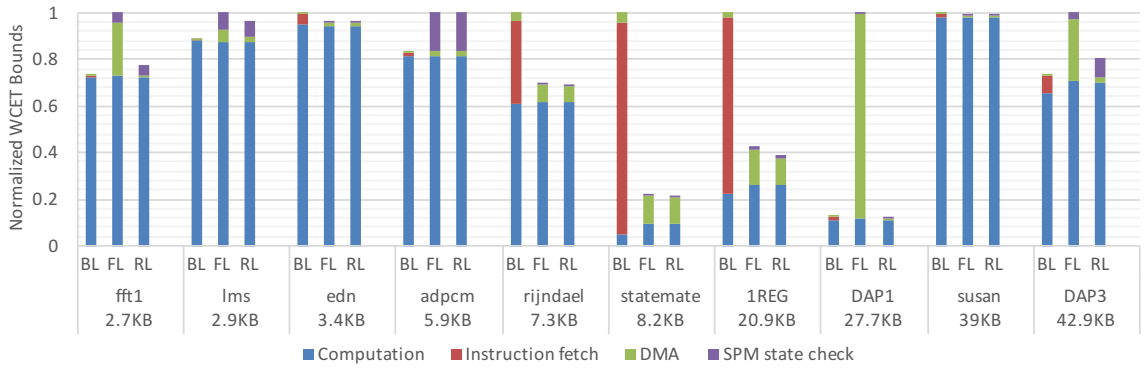


Figure 3.8: The SPM size for each benchmark is increased to the 80% of code size. While BL outperforms FL and RL in several benchmarks, RL is still the best performing technique overall, as it does not show disastrous results like BL in `rijndael`, `statemate`, and `1REG`.

Note that in several benchmarks such as `statemate` and `DAP3`, the WCEPs are different for different management techniques, so the computation components in WCET bounds take different proportions.

3.5.3 Changing Memory Sizes

Figure 3.8 shows the results when we change SPM sizes to be 80% of the original code size of the benchmark. With these larger SPM sizes, the largest functions of all benchmark can fit in the SPM, and the figure shows the results for all three techniques for all benchmarks.

Since more number of instructions can fit in the larger SPMs, the overheads of instruction fetch (red bars) for BL have been eliminated almost completely for most benchmarks. In `statemate`, the greedy algorithm prematurely stops and causes such a high WCET as same as in the baseline results (footnote 7). Even with a large SPM size, `rijndael` and `1REG` have large loops whose sizes are larger than the SPM, and some basic blocks in the loops had to be left in the main memory.

In `fft1`, `lms`, `DAP1`, and `DAP3`, function-splitting can effectively help reduce the WCET for FL, and there are significant differences in the WCETs for FL and RL.

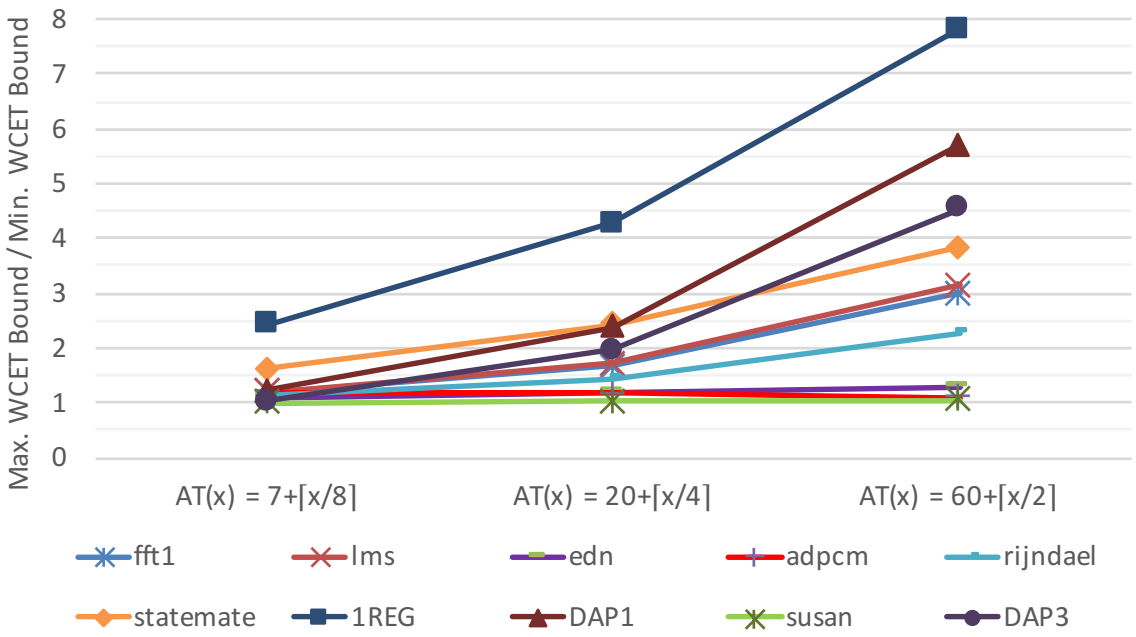


Figure 3.9: The differences in WCET bounds tend to increase as the main memory access latencies increase.

3.5.4 Changing Memory Access Times

Here we change the parameters in Equation (3.1) to model different execution environments in terms of memory access times. We use two additional sets of values for S and B. The first case favors directly accessing main memory with little setup time ($S = 7$) and a large bandwidth ($B = 8$, 64 bits/cycle). This setup models simple micro-controllers with slow core clock speeds and on-chip main memory interconnected through an internal bus interface. The second setup has a larger setup time ($S = 60$) and a narrower bandwidth ($B = 2$, 16 bits/cycle), modeling micro-processors with faster clock speeds and external off-chip main memory. Considering typical desktop environments where a main memory access typically takes hundreds of cycles, the accesses can become even more expensive in future systems. The baseline ($S = 20, B = 4$) lies in the middle.

For all benchmarks, we observed that the overall trend regarding which technique performs better than which technique did not change even with different memory access

time parameters. For example, let us consider DAP1 in the baseline results shown in Figure 3.7. The WCET bound for BL is greater than the one for RL. Such trends do not change with different parameters for all benchmarks, but the differences between WCET estimates change. We show the changes of the maximum differences of WCET estimates in Figure 3.9. The values are the maximum WCET bound divided by the minimum WCET bound for each benchmark. For example, the top right blue square means that with parameters ($S = 60, B = 2$), the WCET bound of DAP1 for BL is 8 times greater than that for RL.

The differences in the WCET estimates increase as the main memory accesses become more expensive. The exception is with the benchmarks where both BL and RL have very little memory access overheads, such as adpcm and susan. In these benchmarks, SPM checking is the dominant source of overheads, which is independent of main memory access times. With longer main memory access times, the WCET estimates for BL increase more rapidly than they do for FL or RL.

3.5.5 Changing Memory Organizations with Caches

The results in Figure 3.7 show that instruction fetches from the main memory are a source of the major performance overhead in BL. This motivates us to use an instruction cache in addition to the SPM so that main memory accesses can be cached.

Since it is already shown that FL can never outperform RL in previous sections, we focus on comparison of BL with cache and RL. Here we model a configurable hybrid architecture in NVIDIA GPUs [8] where the ratio of SPM size and cache size can be configured to 3:1, 1:3, or 1:1, keeping the total size of on-chip SRAM the same. Note that if we consider the fact that SPMs typically occupy much smaller die area than caches of the same

[8] CUDA compute capabilities 3.x, http://docs.nvidia.com/cuda

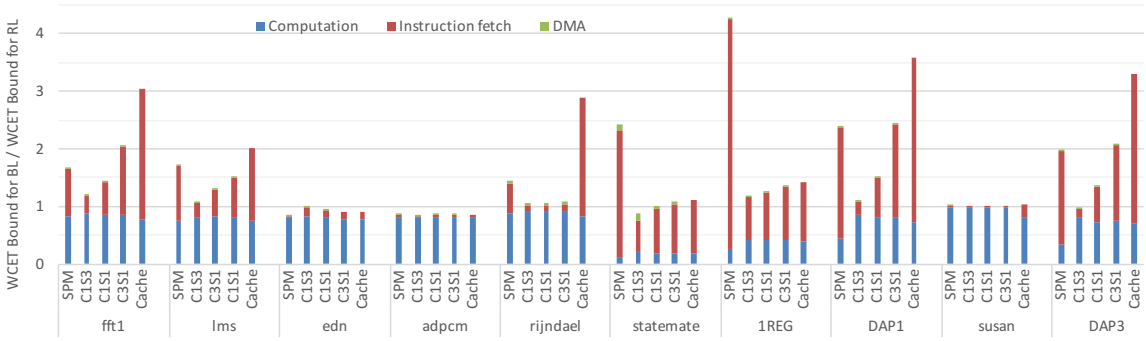


Figure 3.10: Converting part of the SPM to a cache can greatly reduce the WCET bounds, but cache-to-spm size ratios other than 1:3 are often not helpful.

size (Banakar *et al.*, 2002a; Redd *et al.*, 2014), this is not completely fair and has an effect of allowing larger on-chip SRAMs for BL.

We use 4-way set associative caches with cache line sizes of 8 words. We assume LRU replacement policy as it is in general the most predictable replacement policy (Guan *et al.*, 2012). To calculate the WCET bounds for caches, we implemented a state-of-the-art cache analysis algorithm by Cullmann (Cullmann, 2013), which uses must, may, and persistence analyses based on abstract interpretation. We use the baseline setup from Section 3.5.1 regarding the memory sizes and memory access times.

Figure 3.10 compares the WCET bounds for BL when there is only SPM or a mix of SPM and cache in different ratios, e.g. C1S3 means cache-to-SPM size ratio is 1:3. We also include cache-only configuration, denoted as CACHE, where the WCETs are only for cache, not using SPM at all. The values are divided by the WCET bound for RL of each benchmark for normalization. For example, the bar for SPM in `1REG` is over 4, and it means that the WCET bound for BL is 4 times higher than that for RL. In `statemate` and `1REG`, the results show that adding caches greatly helps with the long-latency instruction fetches from the main memory. These benchmarks have particularly large loops, whose sizes are much larger than the SPM size, and BL leaves many basic blocks in the main memory. Overall, C1S3 is the best configuration for BL, in which the WCET estimates are

the lowest among three different setups. Those WCET estimates are, however, still greater than the ones for RL in most benchmarks. Cache sizes larger than 1/3 of the total on-chip memory size show significantly higher WCET estimates in many benchmarks due to the pessimism in cache analysis.

Note that loading basic blocks can introduce additional cache misses. The additional load instructions or long jump instructions increase the code size, and the basic blocks allocated to the SPM bypass the cache, not loading memory blocks for the next basic blocks to the cache.

When most of the instruction cache accesses in a basic block are hits, loading its predecessor basic block into the SPM can thus increase the WCET, due to the additional cache misses. This effect causes the greedy heuristic of BL to stop prematurely in `edn`, and the WCET bound for C1S3 is slightly greater than the one for SPM.

3.6 Summary and Conclusion

As real-time applications become increasingly larger and more complex, it becomes important to have a system not only the time-predictable but also of high computing power. Using scratchpad memories (SPMs) is a promising way of achieving such characteristics, but it requires an efficient dynamic management technique to make the performance and predictability scalable. In this chapter, we compared and characterized dynamic management techniques for instruction SPMs with different allocation granularity and different management schemes. We discussed the impact of these differences on various aspects in detail, both qualitatively and quantitatively with a thorough evaluation. Our conclusion is the following list of considerations for future directions on improving existing code management techniques.

Firstly, regions (groups of consecutive basic blocks) appear to be the best granularity of allocation. Due to burst mode accesses, loading a large number of consecutive instructions

at once is beneficial, often for even non-loop code. A partitioning scheme that can minimize the WCET by reducing both memory footprints and loading frequencies is much needed.

Secondly, simplicity is important in a management scheme, but it should not be too simple. The cache-like management in function-level techniques introduces variability in load operations and requires a separate analysis to estimate the WCET, as we described in Chapter 2. While this can also increase the management overhead, we observed in the results that SPM state checking is rarely a dominant source of overheads [9]. On top of this, we also observed that this cache-like management scheme combined with function-splitting (RL) greatly outperformed the deterministic load-only management scheme (BL) in many cases. While all other techniques mainly try to solve "what to load'" problem, the function-level management focuses on "where to load" problem by finding a function-to-region mapping or region-free mapping. The results show that the latter problem is also important and cannot be ignored.

Thirdly, it is important to have an accurate timing model of load operations and other main memory accesses. Many previous approaches take too simple approaches by considering only the transfer sizes and ignoring any overhead involved in load operations. This can exaggerate the performance of basic block level code management techniques. Accurate timing models cannot only improve the accuracy of the results, but can open up more opportunities of improvements.

Fourthly, it may not be beneficial to fetch all instructions from the SPM. For small pieces of non-loop code that lie in nonconsecutive address ranges, loading them can be more expensive than fetching them directly from the main memory, depending on the cost of load operations and the main memory access times. At the same time, when main mem-

[9] To reduce this overhead, Cai *et al.* (2017) presented a technique based on static analysis to identify the call sites where the outcome of SPM-state-checking codes can be determined at compile-time. In the call sites where the callee function is guaranteed to be already loaded or to stay loaded once it is loaded, the state checking code can be eliminated or hoisted out of a loop. We did not use this technique in this work.

ory accesses are cached, we should avoid unexpectedly increasing the WCET by causing additional cache misses.

Chapter 4

WCET-AWARE DYNAMIC STACK FRAME MANAGEMENT

When the call stack resides in a size-limited scratchpad memory, the stack frames must be evicted to and loaded back from the main memory to avoid stack overflows. In this chapter, we present a technique to find optimal program locations to perform stack management operations such that the WCET of a given program is minimized.

4.1 Introduction

The typical sizes of SPMs are not large—within a few megabytes in most processors (Wang *et al.*, 2016). Moreover, given the lack of virtual memory support of SPMs (due to the nature of explicit management), programs running on an SPM-based processor can easily have stack overflow errors, if the entire stack is kept in the SPM. Keeping the stack in the main memory prevents the problem but will significantly degrade performance. Most previous approaches solve this problem by allocating the SPM space for selected stack variables (that are critical for performance) while the rest of the stack data in the main memory (Avissar *et al.*, 2002; Kim, 2011; Nguyen *et al.*, 2009; Suhendra *et al.*, 2005; Udayakumaran *et al.*, 2006; Wan *et al.*, 2012). The key question in both types of approaches is *which stack variables should be kept in the fast SPM*. To answer this question, a compiler would need to obtain all possible target addresses of all memory references and evaluate how much each reference contributes to the WCET. This is very challenging and often incurs significant pessimism when there are input-dependent memory references or data-dependent control flows (Ramaprasad and Mueller, 2005; Staschulat and Ernst, 2006).

Lu *et al.* (2013), on the other hand, take a different approach. In this approach, the stack resides in the SPM, but *stack frames* are temporarily evicted to (and restored from) the main

memory at *selected* call sites, to accommodate future stack frames and thus prevent any stack overflow. All stack data accesses can benefit from the fast latency of the SPM, and the only long-latency main memory accesses are for stack frame management before and after a selected function call. Therefore, the key decision here is to pick such call sites to perform stack management. One definite benefit with this approach is that the compiler can be completely agnostic about the data access patterns when making the decision to select call sites to perform stack management. Another benefit is that this approach makes WCET analysis trivial at least with regard to stack data references since all stack data references will have constant costs.

In this chapter, we present not only a *WCET analysis* technique that finds the WCET of a program for a set of call sites to perform stack management, but also a technique to *optimally select the call sites such that the resulting WCET is minimized*. Our approach is based on integer linear programming (ILP) and takes the stack frame sizes of all functions and the maximum execution frequency of each basic block as input. We evaluate our approach using Mälardalen WCET benchmark suite (Gustafsson *et al.*, 2010b). Compared to a recently-proposed WCET-optimizing management technique (Liu and Zhang, 2015) that also works at the granularity of the whole stack frames, our approach can reduce the WCET up to 48%. Compared to the caches of the same size, our approach can achieve WCETs that are comparable to the WCETs obtained by the de-facto standard static cache analysis technique (Cullmann, 2013).

4.2 Related Work

SPM management in the context of reducing WCETs has been extensively studied in the literature. As hard real-time applications rarely use heap data, researchers have focused on how to use SPMs for program code (Falk and Kleinsorge, 2009; Kim *et al.*, 2014) or stack/global data (Avissar *et al.*, 2002; Kim, 2011; Liu and Zhang, 2015; Nguyen *et al.*,

2009; Suhendra *et al.*, 2005; Udayakumaran *et al.*, 2006; Wan *et al.*, 2012). Using these techniques, a compiler can allocate SPM space for selected code/data blocks at compile-time (static management) (Avissar *et al.*, 2002; Falk and Kleinsorge, 2009; Kim, 2011; Nguyen *et al.*, 2009; Suhendra *et al.*, 2005) or transform the code so that different code/-data blocks are loaded to and evicted from their allocated space at runtime (dynamic management) (Kim *et al.*, 2014; Liu and Zhang, 2015; Udayakumaran *et al.*, 2006; Wan *et al.*, 2012). WCET-aware C compiler framework (WCC) (Falk and Lokuciejewski, 2010) includes similar techniques to use the SPM space for code and global data.

Our closest related work is dynamic stack management by Lu *et al.* (2013). This work optimizes circular stack management (Shrivastava *et al.*, 2009) by eliminating the runtime stack manager that checks the available space in the SPM. It, however, considers only the overall DMA transfer overhead and cannot optimize the WCET of a given program. We find an optimal set of such function call sites in a program to perform data movement such that the WCET is minimized.

In terms of stack data management, the stack-frame-level management mechanism used in our approach has two unique characteristics. Firstly, it makes all local variable accesses to be SPM accesses, which simplifies static WCET analysis and accords closely with the premise of SPMs as a time-predictable alternative to caches. Most previous approaches either i) *divide the stack into two* (one in the main memory and the other in the SPM) using two stack pointers (Avissar *et al.*, 2002; Nguyen *et al.*, 2009) or ii) *allocate SPM space for only selected stack variables* Kim (2011); Suhendra *et al.* (2005); Udayakumaran *et al.* (2006); Wan *et al.* (2012). This allocation is typically done by creating a copy of a stack variable in the SPM as a global variable (Suhendra *et al.*, 2005; Udayakumaran *et al.*, 2006). These techniques require accurate value analysis to decide which variables to keep in the SPM, which is challenging in the presence of input-dependence or data-dependence (Ramaprasad and Mueller, 2005; Staschulat and Ernst, 2006).

The second uniqueness is the use of data movement between the SPM and the main memory. Stack data have a unique characteristics of transience since the stack frame of a function disappears as the function returns. Therefore, almost all previous approaches (Avissar *et al.*, 2002; Kim, 2011; Liu and Zhang, 2015; Nguyen *et al.*, 2009; Suhendra *et al.*, 2005; Udayakumaran *et al.*, 2006; Wan *et al.*, 2012) do not involve data transfer operations. In our approach, the whole stack is moved from the SPM to make space for incoming stack frames before a selected function call, and the stack is restored after the function returns. Since the size of the stack before each function call can be determined at compile-time once stack frame sizes are available, the costs of data movement operations are also deterministic. Thus, the use of data movement does not harm time predictability. It also helps with performance as data movement based on direct memory access (DMA) operations can benefit from burst mode accesses to transfer the contiguous memory ranges.

Recently, Liu and Zhang (2015) proposed a WCET-optimizing allocation technique for multi-level SPMs. Since they treat a stack frame as a single aggregated data object, this approach also works at the granularity of stack frames, not individual variables. They formulate an ILP to decide whether the stack frame of each function should be allocated in the SPM or in the main memory. The ILP objective function tries to maximize the profit of using the SPM, but it cannot directly consider the WCET impact nor the change of the worst-case execution path. Since the worst-case execution path can dynamically change according to SPM allocation decisions, this approach cannot find an optimal allocation scheme.

In the context of caches, many researchers have proposed techniques to lock selected instructions or data blocks into cache to optimize worst-case performance. Mittal (2016) recently presented an extensive survey of such techniques. Also, Schoeberl and Nielsen (2016) have designed a specialized cache for stack data, separating the stack data accesses from other data accesses.

```
// Management code before a function call
DMA store the stack to the main memory;
reset the stack pointer;
call a function;   // selected call site
// Management code after a function call
DMA load the stack from the main memory;
restore the stack pointer;
```

Figure 4.1: Code modification at a selected call site

4.3 Background: Dynamic Stack Frame Management

The dynamic stack frame management mechanism used in this work is first presented by Lu *et al.* (2013). It works in two steps. First, the set of call sites (program locations where a function call takes place) to perform stack management is selected. At each selected call site, the program code is transformed to transfer stack frames between the SPM and the main memory, as shown in Figure 4.1. All stack frames residing in the SPM at the moment are evicted before the call and then restored after the call.

This mechanism is an optimized version of the circular stack management (Bai *et al.*, 2011; Shrivastava *et al.*, 2009) with much less dynamic code overhead (Lu *et al.*, 2013). In the circular stack management, a pair of stack management function call (for evicting and restoring stack frames) is inserted at every call site, unlike our mechanism that inserts management code at *selected call sites*. The management function keeps track of the stack size and performs transfer operations if the remaining SPM size is not enough for the stack frame size of the callee function. While the whole stack residing in the SPM is evicted in our management scheme, circular stack management evicts only the latest stack frames (the closest from the top) that are just enough to free up the SPM space for the next stack frame. This causes much higher computation overhead at runtime than our management mechanism.

As Figure 4.2 illustrates, a stack management operation in this management mechanism requires the previous execution history. For example, stack frames may need to be evicted

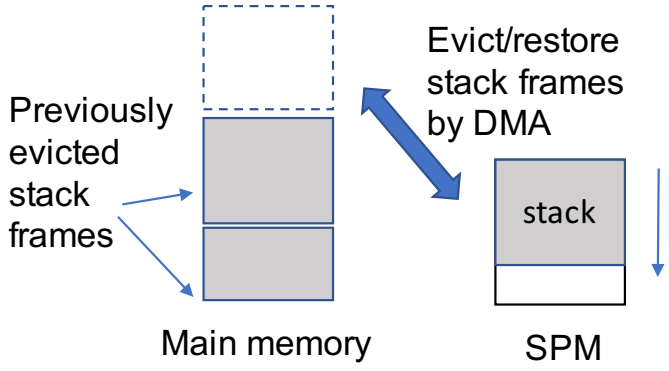


Figure 4.2: Moving stack frames needs the previous execution history to find source/destination addresses for DMA operations.

multiple times in a cascading manner, when the stack grows deep for a long call chain. In this case, the destination address for evicting stack frames depends on the previous eviction history. A naive implementation would use a data structure such as stack to record the sizes of evicted frame sizes and calculate the correct addresses for DMA operations at runtime.

While how to efficiently perform stack management is not our direct focus nor a contribution in this work, we would like to stress that all necessary information to perform management is in fact, available at compile-time. Using compile-time hard-coded information rather than runtime data structure can reduce the overhead of performing management operations. The stack depth at a given program location depends on its execution context (function call history), and an execution context can be represented as a unique path on a call graph. Using compile-time hard-coded information rather than runtime data structure can reduce the overhead of performing management operations.

For a function that can be called by multiple callers, software control flow checking method using signatures (Oh *et al.*, 2002) can be used. Using a signature variable, we can keep track of the call history and identify the execution context at runtime. The management code shown in Figure 4.1 can be inserted for each possible signature value.

4.3.1 Limitations

One limitation of the approach in this chapter is the lack of support for recursive functions calls. Since hard real-time applications hardly use recursion, this does not critically limit the application of our technique. The approach by Lu *et al.* (2013) supports direct recursions by simply evicting the stack frame of a recursive function every time it is called, but indirect recursion is not supported.

Another limitation is that our approach cannot guarantee a correct execution if a function accesses data in another function's stack frames through a pointer variable passed by a parameter. This is because the address value in a pointer variable is no longer valid after the stack frames are moved to the main memory. Cai and Shrivastava Cai and Shrivastava (2016) solve this problem by inserting additional code before pointer accesses to explicitly perform address translation at runtime.

4.4 WCET-Aware Dynamic Stack Frame Management

The focus of this work is to select the call sites to perform stack frame management operations. We present a technique based on integer linear programming (ILP) to find an optimal set of call sites to perform management operations such that the WCET of a given program is minimized. We use the inline CFG from Section 2.4.1 to represent an input program.

Note that unlike code management from Chapter 2, our stack frame management mechanism does not require any preliminary analysis. This is because all management operations are deterministic and have no runtime variability, whereas our code management techniques employ conditional DMA operations, like cache miss handling, and require static analyses to estimate the outcome of the conditional statements.

4.4.1 ILP Formulation

We take an inlined CFG G, the size of the SPM, stack frame sizes, and the loop bounds as input. Our formulation needs the graph G to be acyclic, so we remove all back edges first, assuming that G is reducible. Variables are written in capital letters, and constants are in small letters.

W_v is the WCET from basic block v to the end of the program. The objective is to minimize the WCET of the whole program.

$$minimize \quad W_{v_s} \tag{4.1}$$

Let n_v be the execution frequency of basic block v, and t_v be the time it takes to execute the instructions in v for once in the worst-case. Then v contributes to the WCET with the sum of its computation cost $n_v \cdot c_v$ and any management cost C_v (defined later). For W_v to be an upper bound of the WCET starting from v, W_v is greater than or equal to the sum of the contributions of v and its successor w as follows.

$$\begin{aligned} \forall (v,w) \in E, \quad & W_v \geq W_w + n_v \cdot (t_v + C_v) \\ & W_{v_t} = n_{v_t} \cdot (t_{v_t} + C_{v_t}) \end{aligned} \tag{4.2}$$

Let $C \subset V$ be the set of all basic blocks containing a function call. For each function call, there is the first basic block in the caller function after the callee function returns. Let $\mathcal{R} \subset V$ be the set of such functions. There is one-to-one correspondence between basic blocks in C and $\mathcal{R}$, so mapping $cl : R \rightarrow C$ states that for all $v \in \mathcal{R}$, $cl(v)$ is the basic block where the corresponding function call occurs. These are the candidates to place stack management operations.

All non-candidate basic blocks have zero management cost.

$$\forall v \in V \setminus (C \cup \mathcal{R}), \quad C_v = 0 \tag{4.3}$$

A basic block $v \in C$ has a management cost only if a management code block is inserted at v, which is denoted by a binary decision variable M_v (M_v is 1 if the management code is inserted before and after the function call at v, and 0 otherwise.). The management cost at $v \in R$ depends on the management operation before its corresponding function call ($M_{cl(v)}$) because any evicted stack frames must be restored when the execution of the caller function resumes.

$$\forall v \in C, \quad C_v = \begin{cases} mo_c(S_v) & \text{if } M_v = 1 \\ 0 & \text{if } M_v = 0 \end{cases} \tag{4.4}$$

$$\forall v \in \mathcal{R}, \quad C_v = \begin{cases} mo_r(S_v) & \text{if } M_{cl(v)} = 1 \\ 0 & \text{if } M_{cl(v)} = 0 \end{cases} \tag{4.5}$$

where S_v is the variable to calculate the stack size at v (defined later), and $mo_c(x)$ and $mo_r(x)$ are the management overhead to evict and restore the stack, respectively, when the size of the stack in the SPM is x bytes. These are not symbols but linear equations to calculate the management overhead on the target hardware platform, e.g., $mo_c(x)$ is the time to execute the additional code for management plus the DMA operation time for transferring x bytes, which again, is a constant setup time plus a transfer time proportionate to x.

The above constraints in Equation (4.4-4.5) need to evaluate `if-then-else` condition between variables, which can be linearized using the big M method Luenberger and Ye (2015). For example, the following is a linearized form of Equation (4.4).

$$\begin{aligned} C_v - mo_c(S_v) + \mathcal{M} \cdot (1 - M_v) &\geq 0 \\ C_v - mo_c(S_v) - \mathcal{M} \cdot (1 - M_v) &\leq 0 \\ C_v + \mathcal{M} \cdot M_v &\geq 0 \\ C_v - \mathcal{M} \cdot M_v &\leq 0 \end{aligned} \tag{4.6}$$

where $\mathcal{M}$ is a sufficiently large integer.

The stack size at basic block v depends on the the stack size at the parent function (the caller function) and whether the stack had been evicted before the current function was called. If the stack was evicted before the current function was called, the stack at v will only have the stack frame of the current function. Let $p(v)$ denote the basic block in which the current function is called. If v is in the starting function ($fn(v_s)$), e.g. `main`, there is no parent function. In the following, $V_{\texttt{main}}$ denotes the set of all basic blocks in the starting function.

$$\forall v \in V_{\texttt{main}},\ \ S_v = sz_{fn(v_s)} \tag{4.7}$$

$$\forall v \notin V_{\texttt{main}},\ \ S_v = \begin{cases} sz_{fn(v)} & \text{if } M_{p(v)} = 1 \\ S_{p(v)} + sz_{fn(v)} & \text{if } M_{p(v)} = 0 \end{cases} \tag{4.8}$$

$$\forall v \in V,\ \ S_v \leq SPMSIZE \tag{4.9}$$

where sz_f denote the stack frame size of function f and $SPMSIZE$ is the size of the SPM. The conditional expression in the Equation (4.8) needs to be linearized similarly as in Equation (4.6).

When a solver finds an optimal solution for the above ILP, we can find an optimal set of call sites from M_v variables. The final objective value is the WCET of the program. If the SPM size is smaller than the largest stack frame in the program, the solver would find it infeasible to solve the ILP.

Using the ILP for WCET analysis purpose only: When the set of locations to perform stack management operations is already given as input, this ILP can be used to calculate a safe upper bound of the WCET, after setting the values of the decision variables M_v's according to the input.

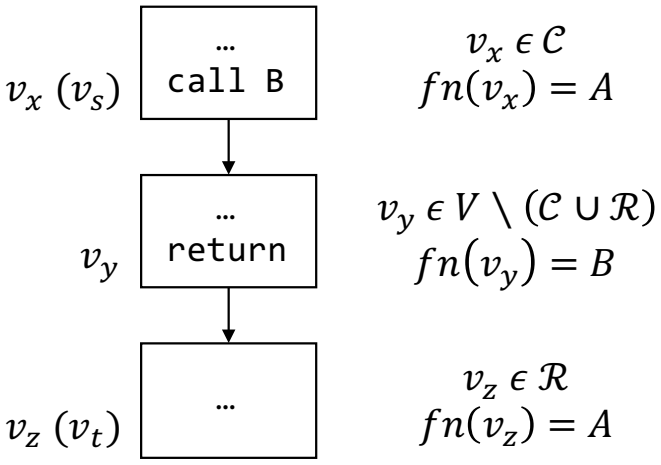


Figure 4.3: A simple inlined CFG used as an example

4.4.2 Example

We illustrate our ILP formulation using a very simple example shown in Figure 4.3. Function A consists of two basic blocks, v_x and v_z, and function B a single basic block, v_y. A calls B at v_x. For simplicity, we assume that each basic block is executed at most 10 times (n_v is 10 for all basic blocks.), and it takes at most 10 cycles to execute each basic block (t_v is 10 for all basic blocks.).

$$W_{v_x} = W_{v_y} + 10 \cdot (10 + C_{v_x})$$

$$W_{v_y} = W_{v_z} + 10 \cdot (10 + C_{v_x})$$

$$W_{v_z} = 10 \cdot (10 + C_{v_x})$$

Let us also assume that the management overhead when the stack size is x bytes is simply x cycles, regardless of whether it is evicting or restoring stack frames. In this case,

the management cost at C_{v_z} is the same as C_{v_x}.

$$C_{v_x} - S_{v_x} + \mathcal{M} \cdot (1 - M_{v_x}) \geq 0$$
$$C_{v_x} - S_{v_x} - \mathcal{M} \cdot (1 - M_{v_x}) \leq 0$$
$$C_{v_x} + \mathcal{M} \cdot M_{v_x} \geq 0$$
$$C_{v_x} - \mathcal{M} \cdot M_{v_x} \leq 0$$
$$C_{v_y} = 0, \quad C_{v_z} = C_{v_x}$$

Stack frame sizes of function A and B are 32 bytes and 64 bytes, respectively. The size of the SPM is 64 Bytes.

$$S_{v_x} = S_{v_z} = 32$$
$$S_{v_y} - 64 + \mathcal{M} \cdot (1 - M_{v_x}) \geq 0$$
$$S_{v_y} - 64 - \mathcal{M} \cdot (1 - M_{v_x}) \leq 0$$
$$S_{v_y} - (64 + S_{v_x}) + \mathcal{M} \cdot M_{v_x} \geq 0$$
$$S_{v_y} - (64 + S_{v_x}) - \mathcal{M} \cdot M_{v_x} \leq 0$$
$$S_{v_x} \leq 64, \;\; S_{v_y} \leq 64, \;\; S_{v_z} \leq 64$$

In this example, we can safely substitute all appearances of $\mathcal{M}$ for $SPMSIZE$ since it is the maximum possible value for both C_{v_x} and S_{v_y}.

The above formulation can find the WCET of the example, which is $((100 + 3200) + 100) + 100 + 3200 = 6700$, and the solution yields that $M_{v_x} = 1$ to insert the management code at v_x and v_z.

4.5 Evaluation

We evaluate our approach by comparing the WCET estimates with two previous approaches: SSDM (Smart Stack Data Management) heuristic by Lu *et al.* (2013) and a

WCET-optimizing allocation technique by Liu and Zhang (2015). We also compare against 2-way set associative caches using a static cache analysis technique (Cullmann, 2013).

We use benchmarks from Mälardalen WCET suite (Gustafsson *et al.*, 2010b). We modified our existing framework from Chapter 2 to generate inlined CFGs from binaries compiled for ARMv4 ISA, perform the cache analysis, and simulate the benchmarks. Due to the limitation of this implementation with instruction decoding, we were not able to generate CFGs for 2 benchmarks. After excluding 14 that have less than 3 functions, 2 with recursions, and 4 with pointer accesses to the local variables in another function, we use all remaining 13 benchmarks.

Here we consider only stack data accesses and ignore all instruction accesses or other global data accesses assuming that they are already loaded in another SPM. This is to simplify the analysis and to focus on the effects on stack data accesses.

Loop bounds are found by profiling, we use the Gurobi optimizer [1] to solve ILPs.

4.5.1 Comparison with Previous Techniques

We assume that transferring x bytes through a DMA operation takes $20 + \lceil x/4 \rceil$ cycles and a load/store access to the main memory takes 20 cycles. For each benchmark, we use two different memory sizes, $min + (max - min) * 0.5$ and $min + (max - min) * 0.7$, where min is the largest stack frame size and max is the maximum stack depth.

Figure 4.4 shows the WCET reduction results over the WCET optimizing technique from Liu and Zhang (2015). The memory access overhead for their technique comes from direct main memory accesses, which happens when the stack frame of a certain function needs to be left in the main memory. Based on ILP, their technique finds an optimal allocation to maximize the profit of using the SPM. In our approach, all SPM accesses have zero overhead (1 cycle), but the overhead comes from DMA operations and executing ad-

[1] Gurobi Optimization, Inc. `http://www.gurobi.com`

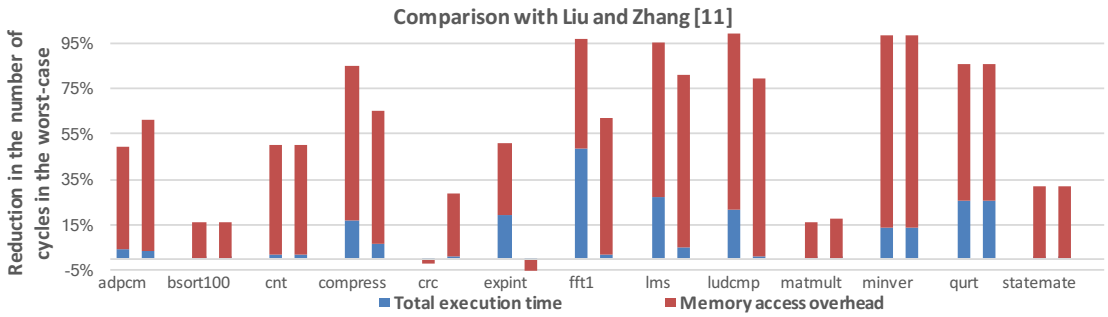


Figure 4.4: Compared to a technique by Liu and Zhang Liu and Zhang (2015), our approach can significantly reduce memory access overhead. The WCET reduction ranges from 0% to 48%.

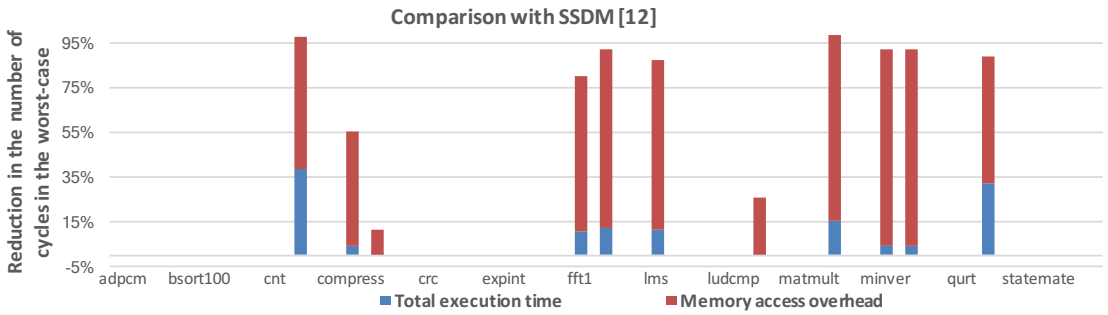


Figure 4.5: SSDM heuristic cannot always find a good solution.

ditional code for management. Their technique has very little overhead when most stack frames can be allocated in the SPM (e.g. `bsort100`, `crc`, `expint`, and `matmult`) but suffers greatly from the long latency of main memory accesses, in most cases. Unfortunately, many benchmarks do not extensively access stack data. Although memory access overhead was greatly optimized in most benchmarks, the WCET reduction ranges from 0% to 48%.

Figure 4.5 shows the WCET reduction results over SSDM (Lu *et al.*, 2013), a greedy heuristic optimized for average-case performance. We calculated the WCET by feeding the solution obtained by SSDM as input to the ILP. As the differences only come from the locations for stack management operations, there was no meaningful reduction in WCET when SSDM could find exactly or almost the same solution that the ILP did. This happens easily for benchmarks with very simple call patterns such as a called function never calling another function, e.g. `bsort100`, `expint`, and `statemate`. Often times, however, SSDM

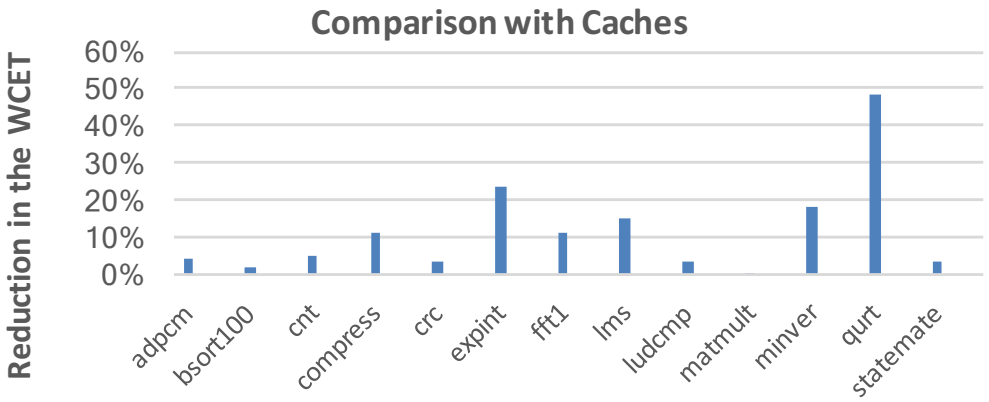


Figure 4.6: The reduction in WCET, compared to 2-way set associative cache, ranges from 0% to 49%.

suffers from its greedy characteristics and gets stuck in local optima. As the figure shows, in `cnt`, `matmult`, and `qurt`, SSDM leads to significantly longer WCETs for larger SPM sizes, quite counter-intuitively, because the larger memory made the heuristic to make bad initial choices.

4.5.2 Comparison with Caches

We estimate the WCETs for 2-way set associative caches with LRU replacement policy using the de-facto standard cache analysis technique (Cullmann, 2013) and implicit path enumeration technique (IPET) Li and Malik (1995). We assume that each line is 16 bytes and cache miss penalty is 20 cycles. Since a possible cache size for a 2-way set associative cache with 16-byte line size is a multiple of 32, the size of cache/SPM for each benchmark is set as the smallest multiple of 32, greater than $min + (max - min) * 0.5$ where min and max are the largest stack frame size and the maximum stack depth as used in the previous subsection.

Figure 4.6 shows the reduction in WCETs over the caches. Our approach outperforms the caches in all benchmarks, but the difference is overall, not significant, except a few benchmarks. It is, however, worth noting that using our approach can reduce almost 50%

of the WCET compared to caches for `qurt` and over 10% for several benchmarks. The main reason behind this is the cache conflict misses in nested loops on the worst-case execution path. While caches are not optimized for the worst-case, our approach always finds an optimal solution for the WCET.

4.6 Summary

When the program call stack is kept in a software-managed scratchpad memory, the stack frames must be explicitly managed between the SPM and the main memory at run-time, to avoid stack overflows. In this chapter, we described a technique to find optimal locations to perform stack management operations in order to minimize the WCET of a given program. Compared to most previous approaches that require selection of local variables to be used in SPM, this approach simplifies the WCET analysis. Evaluation results show that our approach can effectively schedule management operations on the worst-case execution path and reduce the WCET.

Chapter 5

CONCLUSION AND FUTURE WORK

Hard real-time systems must undergo a thorough timing verification to ensure the correct temporal behavior. Timing verification based on non-exhaustive measurement or testing, although widely used in practice, cannot be considered safe due to the lack of guarantee of coverage. While timing verification based on static analysis guarantees the safety of the approach, the conventional practice in processor microarchitecture design can complicate static analysis and make the results pessimistic. In efforts to find timing-predictable alternatives to the traditional microarchitectures, scratchpad memories (SPMs) have gained researchers' attention. SPMs facilitate static analysis with the software-controlled nature but require a change in the processor tool chain to insert explicit data movement instructions in the code. SPM management techniques used in a compiler decide how to use the SPM space by data allocation and access scheduling. As such decisions have a significant impact on performance and predictability, management techniques used for hard real-time systems must be intelligent and able to optimize the worst-case execution times (WCETs) of tasks. In this dissertation, we present several management techniques for program code and stack data. The proposed management techniques, while facilitating static analysis, can greatly improve the WCETs of many benchmarks in comparison with other management techniques or hardware caching. The comparison study with other management techniques offers several insights on future research directions.

The main focus of this thesis is to develop SPM management techniques for single-task execution environments. Although the results of this work demonstrates the effectiveness and the promise of using SPM-based processors in hard real-time systems, there is more work to complete this work. The following is the list of future work.

Global data management: All types of data need to be managed, not just code and stack. As hard real-time applications rarely use heap data, we need a WCET-aware management technique for global data. One can also consider converting local variables to global variables, or vice versa, as it can affect the overall WCET.

Intra-task SPM partitioning: After developing management techniques for all types of data, SPM space needs to be partitioned for different types of data. Each type of data will be allocated a separate SPM space, not corrupting the memory space for other types of data. This partitioning must be done in a way to optimize the WCET. For example, it could be beneficial for a task to have more SPM space for code and very little space for stack.

Single-core multi-tasking or **Inter-task SPM partitioning**: When multiple tasks are scheduled preemptively on one core sharing its local SPMs, we can divide the SPM into multiple partitions that will be assigned to tasks as private SPM spaces. As SPMs inherently provide access privatization, any preemption delays, caused by the access interference among tasks, can be automatically eliminated. This partitioning scheme will need to carefully consider the trade-off in sizing partitions, as the WCETs of the tasks will be affected by using only a fraction of a given SPM. One can also consider designating an SPM space for shared data or library code for the benefit of all tasks.

Multi-core multi-tasking or **Task-mapping**: When there are multiple cores, we need to decide which tasks to share a core. This will become more challenging and interesting in heterogeneous multi-cores where cores can have different characteristics. When the target architecture has a shared SPM, shared by different cores, it can be used similarly to the shared partition in SPM partitions. All these decisions can be either made at compile-time with a static scheduling policy or at run-time by the operating system with a dynamic scheduling policy.

REFERENCES

Altmeyer, S. and C. M. Burguière, "Cache-related preemption delay via useful cache blocks: Survey and redefinition", Journal of Systems Architecture **57**, 7, 707 – 719, URL http://www.sciencedirect.com/science/article/pii/S1383762110001062, special Issue on Worst-Case Execution-Time Analysis (2011).

Altmeyer, S., R. I. Davis and C. Maiza, "Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems", in "Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium", RTSS '11, pp. 261–271 (IEEE Computer Society, Washington, DC, USA, 2011), URL http://dx.doi.org/10.1109/RTSS.2011.31.

Amrutur, B. S. and M. A. Horowitz, "Speed and Power Scaling of SRAMâĂŹs", IEEE Journal of Solid-State Circuits **35**, 2, 175–185 (2000).

Avissar, O., R. Barua and D. Stewart, "An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems", ACM Trans. Embed. Comput. Syst. **1**, 1, 6–26, URL http://doi.acm.org/10.1145/581888.581891 (2002).

Axer, P., R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm and W. Yi, "Building Timing Predictable Embedded Systems", ACM Trans. Embed. Comput. Syst. **13**, 4, 82:1–82:37, URL http://doi.acm.org/10.1145/2560033 (2014).

Bai, K., J. Lu, A. Shrivastava and B. Holton, "CMSM: An Efficient and Effective Code Management for Software Managed Multicores", in "Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis", CODES+ISSS '13, pp. 11:1–11:9 (IEEE Press, Piscataway, NJ, USA, 2013), URL http://dl.acm.org/citation.cfm?id=2555692.2555703.

Bai, K., A. Shrivastava and S. Kudchadker, "Stack Data Management for Limited Local Memory (LLM) Multi-core Processors", in "ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors", pp. 231–234 (2011).

Baker, M. A., A. Panda, N. Ghadge, A. Kadne and K. S. Chatha, "A Performance Model and Code Overlay Generator for Scratchpad Enhanced Embedded Processors", in "Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis", CODES/ISSS '10, pp. 287–296 (ACM, New York, NY, USA, 2010), URL http://doi.acm.org/10.1145/1878961.1879011.

Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel, "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems", in Banakar *et al.* (2002b), pp. 73–78, URL http://doi.acm.org/10.1145/774789.774805.

Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel, "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems", in "Proceedings of the Tenth International Symposium on Hardware/Software Codesign",

CODES '02, pp. 73–78 (ACM, New York, NY, USA, 2002b), URL http://doi.acm.org/10.1145/774789.774805.

Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, "The Gem5 Simulator", SIGARCH Comput. Archit. News **39**, 2, 1–7, URL http://doi.acm.org/10.1145/2024716.2024718 (2011).

Bradley, S. P., A. C. Hax and T. L. Magnanti, *Applied Mathematical Programming* (Addison-Wesley Publishing Company, 1977).

Buttazzo, G. C., *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* (Springer Publishing Company, 2011), 3rd edn.

Cai, J., Y. Kim, Y. Kim, A. Shrivastava and K. Lee, "Reducing Code Management Overhead in Software-Managed Multicores", in "Proceedings of the Conference on Design, Automation and Test in Europe (To Appear)", DATE âĂŹ17 (2017).

Cai, J. and A. Shrivastava, "Efficient Pointer Management of Stack Data for Software Managed Multicores", in "Proceedings of the International Conference on Application Specific Systems, Architectures and Processors (ASAP)", (2016).

Cazorla, F. J., E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo and D. Maxim, "PROARTIS: Probabilistically Analyzable Real-Time Systems", ACM Trans. Embed. Comput. Syst. **12**, 2s, 94:1–94:26, URL http://doi.acm.org/10.1145/2465787.2465796 (2013).

Chattopadhyay, S. and A. Roychoudhury, "Cache-Related Preemption Delay Analysis for Multilevel Noninclusive Caches", ACM Trans. Embed. Comput. Syst. **13**, 5s, 147:1–147:29, URL http://doi.acm.org/10.1145/2632156 (2014).

Cousot, P. and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", in "Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages", POPL '77, pp. 238–252 (ACM, New York, NY, USA, 1977), URL http://doi.acm.org/10.1145/512950.512973.

Cullmann, C., "Cache Persistence Analysis: Theory and Practice", ACM Trans. Embed. Comput. Syst. **12**, 1s, 40:1–40:25, URL http://doi.acm.org/10.1145/2435227.2435236 (2013).

Deverge, J.-F. and I. Puaut, "WCET-Directed Dynamic Scratchpad Memory Allocation of Data", in "19th Euromicro Conference on Real-Time Systems (ECRTS'07)", pp. 179–190 (2007).

Ding, H., Y. Liang and T. Mitra, "WCET-Centric Dynamic Instruction Cache Locking", in "2014 Design, Automation Test in Europe Conference Exhibition (DATE)", pp. 1–6 (2014).

Egger, B., C. Kim, C. Jang, Y. Nam, J. Lee and S. L. Min, "A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization", in "Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems", CASES '06, pp. 223–233 (ACM, New York, NY, USA, 2006), URL http://doi.acm.org/10.1145/1176760.1176788.

Falk, H. and J. C. Kleinsorge, "Optimal Static WCET-aware Scratchpad Allocation of Program Code", in "Proceedings of the 46th Annual Design Automation Conference", DAC '09, pp. 732–737 (ACM, New York, NY, USA, 2009), URL http://doi.acm.org/10.1145/1629911.1630101.

Falk, H. and H. Kotthaus, "WCET-driven Cache-aware Code Positioning", in "Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems", CASES '11, pp. 145–154 (ACM, New York, NY, USA, 2011), URL http://doi.acm.org/10.1145/2038698.2038722.

Falk, H. and P. Lokuciejewski, "A Compiler Framework for the Reduction of Worst-case Execution Times", Real-Time Syst. **46**, 2, 251–300, URL http://dx.doi.org/10.1007/s11241-010-9101-x (2010).

Ferdinand, C., "Worst-case execution time prediction by static program analysis", in "18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.", p. 125âĂŞ127 (2004).

Ferdinand, C. and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems", Real-Time Systems **17**, 2, 131–181 (1999).

Gracioli, G., A. Alhammad, R. Mancuso, A. A. Fröhlich and R. Pellizzoni, "A Survey on Cache Management Mechanisms for Real-Time Embedded Systems", ACM Comput. Surv. **48**, 2, 32:1–32:36, URL http://doi.acm.org/10.1145/2830555 (2015).

Guan, N., M. Lv, W. Yi and G. Yu, "WCET Analysis with MRU Caches: Challenging LRU for Predictability", in "2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium", pp. 55–64 (2012).

Gustafsson, J., A. Betts, A. Ermedahl and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future", in Gustafsson *et al.* (2010b), pp. 137–147.

Gustafsson, J., A. Betts, A. Ermedahl and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future", in "Proceedings of International Workshop on Worst-Case Execution Time Analysis", edited by B. Lisper, pp. 137–147 (OCG, Brussels, Belgium, 2010b).

Guthaus, M. R., J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", in "Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop", WWC '01, pp. 3–14 (IEEE Computer Society, Washington, DC, USA, 2001), URL http://dx.doi.org/10.1109/WWC.2001.15.

Huber, B., S. Hepp and M. Schoeberl, "Scope-Based Method Cache Analysis", in "Proceedings of International Workshop on Worst-Case Execution Time Analysis", vol. 39, pp. 73–82 (2014).

Janapsatya, A., A. Ignjatović and S. Parameswaran, "A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric", in "Proceedings of the 2006 Asia and South Pacific Design Automation Conference", ASP-DAC '06, pp. 612–617 (IEEE Press, Piscataway, NJ, USA, 2006), URL http://dx.doi.org/10.1145/1118299.1118443.

Johnson, R., D. Pearson and K. Pingali, "The Program Structure Tree: Computing Control Regions in Linear Time", in "Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation", PLDI '94, pp. 171–185 (ACM, New York, NY, USA, 1994), URL http://doi.acm.org/10.1145/178243.178258.

Jung, S. C., A. Shrivastava and K. Bai, "Dynamic Code Mapping for Limited Local Memory Systems", in "ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors", pp. 13–20 (2010).

Kahle, J. A., M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. Shippy, "Introduction to the Cell Multiprocessor", IBM J. Res. Dev. **49**, 4/5, 589–604, URL http://dl.acm.org/citation.cfm?id=1148882.1148891 (2005).

Kandemir, M. and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management", in "Proceedings of the 39th Annual Design Automation Conference", DAC '02, pp. 628–633 (ACM, New York, NY, USA, 2002), URL http://doi.acm.org/10.1145/513918.514077.

Khedker, U., A. Sanyal and B. Karkare, *Data Flow Analysis: Theory and Practice* (CRC Press, Inc., Boca Raton, FL, USA, 2009), 1st edn.

Kim, H., D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava and J. Oh, "A Predictable and Command-level Priority-based DRAM Controller for Mixed-Criticality Systems", in "21st IEEE Real-Time and Embedded Technology and Applications Symposium", pp. 317–326 (2015).

Kim, H., D. De Niz, B. Andersson, M. Klein, O. Mutlu and R. Rajkumar, "Bounding and Reducing Memory Interference in COTS-based Multi-core Systems", Real-Time Systems **52**, 3, 356–395, URL http://dx.doi.org/10.1007/s11241-016-9248-1 (2016a).

Kim, S., "Using Scratchpad Memory for Stack Data in Hard Real-Time Embedded Systems", in "Memory Architecture and Organization Workshop, co-located with ESWEEK", (2011).

Kim, Y., D. Broman, J. Cai and A. Shrivastava, "WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores", in "2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)", pp. 179–188 (2014).

Kim, Y., J. Cai, Y. Kim, K. Lee and A. Shrivastava, "Splitting Functions in Code Management on Scratchpad Memories", in "Proceedings of the 35th International Conference on Computer-Aided Design", ICCAD '16, pp. 60:1–60:8 (ACM, New York, NY, USA, 2016b), URL http://doi.acm.org/10.1145/2966986.2967075.

Lee, E. A., "Cyber Physical Systems: Design Challenges", in "Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing", ISORC '08, pp. 363–369 (IEEE Computer Society, Washington, DC, USA, 2008), URL http://dx.doi.org/10.1109/ISORC.2008.25.

Li, F., M. Zhao and C. Xue, "C3: Cooperative Code Positioning and Cache Locking for WCET Minimization", in "2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications", pp. 51–59 (2015).

Li, Y.-T. S. and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", in "Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference", DAC '95, pp. 456–461 (ACM, New York, NY, USA, 1995), URL http://doi.acm.org/10.1145/217474.217570.

Liu, I., J. Reineke, D. Broman, M. Zimmer and E. A. Lee, "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance", in "2012 IEEE 30th International Conference on Computer Design (ICCD)", pp. 87–93 (2012).

Liu, T., Y. Zhao, M. Li and C. J. Xue, "Task Assignment with Cache Partitioning and Locking for WCET Minimization on MPSoC", in "2010 39th International Conference on Parallel Processing", pp. 573–582 (2010).

Liu, Y. and W. Zhang, "Scratchpad Memory Architectures and Allocation Algorithms for Hard Real-Time Multicore Processors", Journal of Computing Science and Engineering pp. 51–72, URL http://dx.doi.org/10.5626/JCSE.2015.9.2.51 (2015).

Lu, J., K. Bai and A. Shrivastava, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)", in "Proceedings of the 50th Annual Design Automation Conference", DAC '13, pp. 149:1–149:8 (ACM, New York, NY, USA, 2013), URL http://doi.acm.org/10.1145/2463209.2488918.

Luenberger, D. G. and Y. Ye, *Linear and Nonlinear Programming* (Springer Publishing Company, Incorporated, 2015).

Lundqvist, T. and P. Stenström, "Timing Anomalies in Dynamically Scheduled Microprocessors", in "Proceedings of the 20th IEEE Real-Time Systems Symposium", RTSS '99, pp. 12– (IEEE Computer Society, Washington, DC, USA, 1999), URL http://dl.acm.org/citation.cfm?id=827271.829103.

Metzlaff, S. and T. Ungerer, "Impact of Instruction Cache and Different Instruction Scratchpads on the WCET Estimate", in "2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems", pp. 1442–1449 (2012).

Metzlaff, S. and T. Ungerer, "A comparison of instruction memories from the WCET perspective", Journal of Systems Architecture **60**, 5, 452–466 (2014).

Mittal, S., "A Survey of Techniques for Cache Locking", ACM Trans. Des. Autom. Electron. Syst. **21**, 3, 49:1–49:24, URL http://doi.acm.org/10.1145/2858792 (2016).

Montgomery, S. L., *Guidelines for the Use of the C Language in Critical Systems* (Motor Industry Research Association, 2013).

Nguyen, N., A. Dominguez and R. Barua, "Memory Allocation for Embedded Systems with a Compile-time-unknown Scratch-pad Size", ACM Trans. Embed. Comput. Syst. **8**, 3, 21:1–21:32, URL http://doi.acm.org/10.1145/1509288.1509293 (2009).

Oh, N., P. P. Shirvani and E. J. McCluskey, "Control-Flow Checking by Software Signatures", IEEE Transactions on Reliability **51**, 1, 111–122 (2002).

Pabalkar, A., A. Shrivastava, A. Kannan and J. Lee, "SDRM: Simultaneous Determination of Regions and Function-to-region Mapping for Scratchpad Memories", in "Proceedings of the 15th International Conference on High Performance Computing", HiPC'08, pp. 569–582 (Springer-Verlag, Berlin, Heidelberg, 2008), URL http://dl.acm.org/citation.cfm?id=1791889.1791947.

Paolieri, M., E. Quiñones and F. J. Cazorla, "Timing Effects of DDR Memory Systems in Hard Real-time Multicore Architectures: Issues and Solutions", ACM Trans. Embed. Comput. Syst. **12**, 1s, 64:1–64:26, URL http://doi.acm.org/10.1145/2435227.2435260 (2013).

Pitter, C. and M. Schoeberl, "A Real-time Java Chip-multiprocessor", ACM Trans. Embed. Comput. Syst. **10**, 1, 9:1–9:34, URL http://doi.acm.org/10.1145/1814539.1814548 (2010).

Plazar, S., J. C. Kleinsorge, P. Marwedel and H. Falk, "WCET-aware Static Locking of Instruction Caches", in "Proceedings of the Tenth International Symposium on Code Generation and Optimization", CGO '12, pp. 44–52 (ACM, New York, NY, USA, 2012), URL http://doi.acm.org/10.1145/2259016.2259023.

Prakash, A. and H. D. Patel, "An Instruction Scratchpad Memory Allocation for the Precision Timed Architecture", in "Proceedings of the Conference on Design, Automation and Test in Europe", DATE '12, pp. 659–664 (EDA Consortium, San Jose, CA, USA, 2012), URL http://dl.acm.org/citation.cfm?id=2492708.2492874.

Puaut, I. and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison", in "2007 Design, Automation Test in Europe Conference Exhibition", pp. 1–6 (2007).

Radio Technical Commission for Aeronautics Special Committee (152), *Software Considerations in Airborne Systems and Equipment Certification: B* (RTCA, 1992).

Ramaprasad, H. and F. Mueller, "Bounding worst-case data cache behavior by analytically deriving cache reference patterns", in "11th IEEE Real Time and Embedded Technology and Applications Symposium", pp. 148–157 (2005).

Redd, B., S. Kellis, N. Gaskin and R. Brown, "The Impact of Process Scaling on Scratchpad Memory Energy Savings", Journal of Low Power Electronics and Applications **4**, 3, 231, URL http://www.mdpi.com/2079-9268/4/3/231 (2014).

Reineke, J., *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity.*, Ph.D. thesis, Saarland University (2009).

Reineke, J., I. Liu, H. D. Patel, S. Kim and E. A. Lee, "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation", in "Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis", CODES+ISSS '11, pp. 99–108 (ACM, New York, NY, USA, 2011), URL http://doi.acm.org/10.1145/2039370.2039388.

Reineke, J., B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger and B. Becker, "A Definition and Classification of Timing Anomalies", in "6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)", edited by F. Mueller, vol. 4 of *OpenAccess Series in Informatics (OASIcs)* (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2006), URL http://drops.dagstuhl.de/opus/volltexte/2006/671.

Schoeberl, M., "Is Time Predictability Quantifiable?", in "2012 International Conference on Embedded Computer Systems (SAMOS)", pp. 333–338 (2012).

Schoeberl, M. and C. Nielsen, "A Stack Cache for Real-Time Systems", in "2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)", pp. 150–157 (2016).

Shrivastava, A., A. Kannan and J. Lee, "A Software-only Solution to Use Scratch Pads for Stack Data", Trans. Comp.-Aided Des. Integ. Cir. Sys. **28**, 11, 1719–1727, URL http://dx.doi.org/10.1109/TCAD.2009.2030592 (2009).

Staschulat, J. and R. Ernst, "Worst case timing analysis of input dependent data cache behavior", in "18th Euromicro Conference on Real-Time Systems (ECRTS'06)", pp. 10 pp.–236 (2006).

Steinke, S., N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan and P. Marwedel, "Reducing Energy Consumption by Dynamic Copying of Instructions Onto Onchip Memory", in "Proceedings of the 15th International Symposium on System Synthesis", ISSS '02, pp. 213–218 (ACM, New York, NY, USA, 2002a), URL http://doi.acm.org/10.1145/581199.581247.

Steinke, S., L. Wehmeyer, B.-S. Lee and P. Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction", in "Proceedings of the Conference on Design, Automation and Test in Europe", DATE '02, pp. 409– (IEEE Computer Society, Washington, DC, USA, 2002b), URL http://dl.acm.org/citation.cfm?id=882452.874376.

Suhendra, V. and T. Mitra, "Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores", in "Proceedings of the 45th Annual Design Automation Conference", DAC '08, pp. 300–303 (ACM, New York, NY, USA, 2008), URL http://doi.acm.org/10.1145/1391469.1391545.

Suhendra, V., T. Mitra, A. Roychoudhury and T. Chen, "WCET centric data allocation to scratchpad memory", in "26th IEEE International Real-Time Systems Symposium (RTSS'05)", pp. 10 pp.–232 (2005).

Suhendra, V., T. Mitra, A. Roychoudhury and T. Chen, "Efficient Detection and Exploitation of Infeasible Paths for Software Timing Analysis", in "Proceedings of the 43rd Annual Design Automation Conference", DAC '06, pp. 358–363 (ACM, New York, NY, USA, 2006), URL http://doi.acm.org/10.1145/1146909.1147002.

Udayakumaran, S., A. Dominguez and R. Barua, "Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions", ACM Trans. Embed. Comput. Syst. **5**, 2, 472–511, URL http://doi.acm.org/10.1145/1151074.1151085 (2006).

Um, J. and T. Kim, "Code Placement with Selective Cache Activity Minimization for Embedded Real-time Software Design", in "Proceedings of the 2003 IEEE/ACM International Conference on Computer-aided Design", ICCAD '03, pp. 197– (IEEE Computer Society, Washington, DC, USA, 2003), URL http://dx.doi.org/10.1109/ICCAD.2003.49.

Ungerer, T., F. J. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. CassÃĹ, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff and J. Mische, "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability", IEEE Micro **30**, 5, 66–75 (2010).

Verma, M., L. Wehmeyer and P. Marwedel, "Dynamic Overlay of Scratchpad Memory for Energy Minimization", in "Proceedings of the 2Nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis", CODES+ISSS '04, pp. 104–109 (ACM, New York, NY, USA, 2004), URL http://doi.acm.org/10.1145/1016720.1016748.

Wan, Q., H. Wu and J. Xue, "WCET-aware Data Selection and Allocation for Scratchpad Memory", in "Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems", LCTES '12, pp. 41–50 (ACM, New York, NY, USA, 2012), URL http://doi.acm.org/10.1145/2248418.2248425.

Wang, C., C. Dong, H. Zeng and Z. Gu, "Minimizing Stack Memory for Hard Real-Time Applications on Multicore Platforms with Partitioned Fixed-Priority or EDF Scheduling", ACM Trans. Des. Autom. Electron. Syst. **21**, 3, 46:1–46:25, URL http://doi.acm.org/10.1145/2846096 (2016).

Wehmeyer, L. and P. Marwedel, "Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software", in "Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1", DATE '05, pp. 600–605 (IEEE Computer Society, Washington, DC, USA, 2005), URL http://dx.doi.org/10.1109/DATE.2005.183.

Whitham, J. and N. Audsley, "Implementing Time-predictable Load and Store Operations", in "Proceedings of the Seventh ACM International Conference on Embedded Software",

EMSOFT '09, pp. 265–274 (ACM, New York, NY, USA, 2009), URL http://doi.acm.org/10.1145/1629335.1629371.

Whitham, J. and N. Audsley, "Optimal Program Partitioning for Predictable Performance", in "2012 24th Euromicro Conference on Real-Time Systems", pp. 122–131 (2012).

Whitham, J. and M. Schoeberl, "WCET-Based Comparison of an Instruction Scratchpad and a Method Cache", in "2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing", pp. 301–308 (2014).

Wilhelm, R., J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat and P. Stenström, "The Worst-case Execution-time Problem–Overview of Methods and Survey of Tools", ACM Trans. Embed. Comput. Syst. **7**, 3, 36:1–36:53, URL http://doi.acm.org/10.1145/1347375.1347389 (2008).

Wu, H., J. Xue and S. Parameswaran, "Optimal WCET-aware Code Selection for Scratchpad Memory", in "Proceedings of the Tenth ACM International Conference on Embedded Software", EMSOFT '10, pp. 59–68 (ACM, New York, NY, USA, 2010), URL http://doi.acm.org/10.1145/1879021.1879030.

Zhao, P. and J. N. Amaral, "Function Outlining and Partial Inlining", in "Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing", SBAC-PAD '05, pp. 101–108 (IEEE Computer Society, Washington, DC, USA, 2005), URL http://dx.doi.org/10.1109/CAHPC.2005.26.

Zimmer, M., D. Broman, C. Shaver and E. A. Lee, "FlexPRET: A processor platform for mixed-criticality systems", in "2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)", pp. 101–110 (2014).

APPENDIX A

CACHE ANALYSIS

In our evaluations throughout this dissertation, we use our own implementation of the de-facto standard static cache analysis algorithm for set associative caches with the least recently used (LRU) replacement policy, first proposed by Ferdinand and Wilhelm (1999). The original algorithm, however, has a bug that can lead to unsafe and underestimation of the WCETs, which is caused by an overestimation of the capacity of set associative caches (Cullmann, 2013). The bug is fixed by Cullmann (2013) recently, which is used in a widely-used WCET analysis tool, aiT WCET Analyzer [1] and also in our implementation. We give a brief introduction of the algorithm in this section.

Based on abstract interpretation (Cousot and Cousot, 1977), the algorithm constructs an abstract representation of cache states before and after executing each instruction. It is a fixed point algorithm such that the abstract cache states are calculated for every instruction by visiting every basic block until the the abstract cache states do not change any more.

Algorithm 8 shows the overall procedure. For a basic block v, let $I(v)$ denote the set of instructions in v. The abstract cache states $\texttt{acs.in}(v,i)$ and $\texttt{acs.out}(v,i)$ represents the cache state before and after executing instruction i in v, respectively. For each instruction $i \in I(v)$, we use $addr(i)$ to denote the set of memory addresses accessed in i.

The algorithm consists of three analyses, called *Must*, *May*, and *Persistence* analysis, and calls `AnalyzeBB` (shown in Algorithm 9) to perform these analyses at each basic block. After performing an analysis, it categorizes each memory access as *Always-Hits*, Always-Misses, or *First-Misses*. As the names suggests, a memory access is categorized as Always-Hit if the access is guaranteed to never cause a cache miss. If the access occurs in a loop and can be guaranteed to be a cache hit in every iteration after only one miss, it is First-Miss. Otherwise, the access is categorized as Always-Miss.

[1] AbsInt Angewandte Informatik GmbH, `https://www.absint.com/ait`

[2] see Cullmann (2013) for the details.

Algorithm 8: Cache analysis

Input: Inlined CFG (G)

Output: Access Categorization (`cat(`*v, i, a*`)` for all $v \in V$, $i \in I(v)$, and $a \in addr(i)$)

```
1  Let acs be an empty cache state
2  repeat foreach v ∈ V do acs = AnalyzeBB(G, v, Must, acs)
   until acs does not change any more
3  foreach v ∈ V do
4      foreach instruction i ∈ I(v) do
5          foreach address a ∈ addr(i) do
6              if acs.in(v,i) contains a then cat(v,i,a) = Always-Hit
               else cat(v,i,a) = Not-Classified
7  repeat foreach v ∈ V do acs = AnalyzeBB(G, v, May, acs)
   until acs does not change any more
8  foreach v ∈ V do
9      foreach instruction i ∈ I(v) do
10         foreach address a ∈ addr(i) that is cat(v,i,a) = Not-Classified do
11             if acs.in(v,i) does not contain a then cat(v,i,a) = Always-Miss
12 repeat foreach v ∈ V do acs = AnalyzeBB(G, v, Persistence, acs)
   until acs does not change any more
13 foreach v ∈ V do
14     foreach instruction i ∈ I(v) do
15         foreach address a ∈ addr(i) that is cat(v,i,a) = Not-Classified do
16             if acs.in(v,i) contains a in a non-virtual line² then cat(v,i,a) = First-Miss
```

Each analysis defines two operations: *update* and *join*. Update operation is executed for each instruction in a basic block. It describes how the abstract cache state that corresponds

Algorithm 9: AnalyzeBB: perform cache analysis for a given basic block

Input: Inlined CFG (G), Basic Block (v), Analysis Mode ($Mode$), Current Abstract Cache State (`acs`)

Output: Updated Abstract Cache State (`acs`)

function `AnalyzeBB`(G, v, $Mode$, `acs`)

1 $\texttt{acs.in}(v, i_1^v) = \texttt{acs.out}(p_1^v, i(p_1^v)_1)$

2 **for** $k = 2$ *to* $|pred(v)|$ **do**

3 $\quad \texttt{acs.in}(v, i_1^v) = \texttt{Join}(\texttt{acs.in}(v, i_1^v), \texttt{acs.out}(p_k^v, i(p_k^v)_1), Mode)$

4 **for** $k = 1$ *to* $|I(v)|$ **do**

5 $\quad$ **foreach** *address* $a \in addr(i_k^v)$ **do**

6 $\quad\quad \texttt{acs.out}(v, i_k^v) = \texttt{Update}(\texttt{acs.in}(v, i_k^v), a, Mode)$

7 $\quad$ **if** $k \neq |I(v)|$ **then** $\texttt{acs.in}(v, i_{k+1}^v) = \texttt{acs.out}(v, i_k^v)$

8 **return** `acs`

to the state after executing an instruction is updated given the set of memory addresses that the instruction accesses. Join operation is, on the other hand, executed to calculate the starting cache state for a basic block that has two or more predecessors. It describes how to merge two incoming abstract cache states, each of which is the cache state after executing the last instruction in a predecessor basic block. Join operation is thus used to calculate the abstract cache state that corresponds to the state before executing the first instruction in a basic block with multiple predecessors. Algorithm 9 shows function `AnalyzeBB` where two operations are applied for a given basic block. For a basic v, $pred(v)$ denotes the set of

its predecessors. Each predecessor is uniquely identified as p_k^v where $1 \leq k \leq |pred(v)|$. The k-th instruction in a basic block v is represented as i_k^v where $1 \leq k \leq |I(v)|$.

In Cullmann (2013), the update and join operation for each analysis are precisely defined. Algorithm 10 and 11 show an implementation of update and join operation, and for brevity, we only show the implementation for Must analysis.

To update an abstract cache state `acs` with a new access to a in Must analysis, function `Update` first looks up a in `acs`. If it is present in the most recently used line (age 0), there is no change to be done for Must analysis. If it is present in the cache but not in the most recently used line, the line that a resides (`acs.line`(h)) is merged with the line younger by one (`acs.line`($h-1$)) and a is removed from the line. All lines younger than the line that a used to reside now age by one. If a is not present in `acs`, all cache lines are aged by one, which makes the least recently used line be flushed. The most recently used line becomes to contain only one item with address a.

The join operation for merging two abstract cache state `acsA` and `acsB` in Must analysis returns the abstract cache state that is a union of two. Thus, all items that are present either in `acsA` or `acsB` remain in the resulting abstract cache state. The age of each item is set by the maximum age between its age in `acsA` and `acsB`. The implementation for other analyses is straightforward and similar to the one for Must analysis.

Algorithm 10: Update operation

Input: Abstract Cache State (acs), Address (a), Analysis Mode ($Mode$)

Output: Updated Abstract Cache State after Accessing address a (acs)

function Update(acs, a, $Mode$)

```
1  h = lookup (acs, a)   // h is the age of a in its set, 0 being the most
   recently used.  If a is not present in acs, h = −1
2  switch Mode do
3      case Must
4          if h > 0 then      // Cache hit
5              acs.line(h) = acs.line(h − 1) ∪ acs.line(h)        // line h is
               merged with line h − 1
6              acs.line(h) = acs.line(h) \{a}          // remove a from line h
7              for l = h − 1 to 1 do
8                  acs.line(l) = acs.line(l − 1)          // line l ages by one
9              acs.line(0) = {a}                           // line 0 has only a
10         else      // Cache miss
11             for l = associativity − 1 to 1 do
12                 acs.line(l) = acs.line(l − 1)          // line l ages by one
13             acs.line(0) = {a}                           // line 0 has only a
14     case May
15         see Cullmann (2013)
16     case Persistence
17         see Cullmann (2013)
18 return acs
```

Algorithm 11: Join operation

Input: Two Abstract Cache States (`acsA` and `acsB`), Analysis Mode (*Mode*)

Output: Updated Abstract Cache State after Accessing address *a* (`acs`)

function `Join(acsA, acsB, Mode)`

```
1  Let acs be an empty cache state
2  switch Mode do
3      case Must     // Take a union with using the maximum age
4          for s = 0 to S − 1 do     // S is the number of sets in acs
5              for l = 0 to associativity − 1 do
6                  foreach address a in acsA.set(s).line(l) do
7                      h = lookup (acsB, a)
8                      if h > l then     // age in acsB is older
9                          add a into acs.set(s).line(h)
10                     else
11                         add a into acs.set(s).line(l)
12     case May
13         see Cullmann (2013)
14     case Persistence
15         see Cullmann (2013)
16 return acs
```