Scratchpad Management in Software Managed Manycore Architectures

by

Jian Cai

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

Approved October 2017 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Carole Wu
Fengbo Ren
Partha Dasgupta

ARIZONA STATE UNIVERSITY

December 2017

ABSTRACT

Caches have long been used to reduce memory access latency. However, the increased complexity of cache coherence can bring significant challenges in processor design as the number of cores increases. While making caches scalable is an important research problem, some researchers are also exploring the possibility of a more power-efficient SRAM called scratchpad memories or SPMs.

SPMs consume significantly less area, and are more energy-efficient per access than caches, and therefore makes the design of on-chip memories much simpler. Unlike caches, which fetch data from memories automatically and maintain coherence with other caches, an SPM requires explicit instructions to transfer code and data from and to main memory. SPM-only architectures are thus named as SMM architectures, since the data movements of such architectures rely on software. SMM processors have been widely used in different areas, such as embedded computing, network processing, or even high performance computing. SMM processors provide a low-power platform. However, the hardware alone does not guarantee the delivery of power efficiency, if applications on such processors suffer from low performance. Efficient software techniques are required. A big body of SPM management techniques for SMM architectures are compiler-directed, as inserting data movement operations by hand force programmers to trace flow of data, which can be error-prone and sometime very difficult. In this thesis, we develop compiler-directed techniques to efficiently manage different types of data of embedded applications between SPMs and main memory in SMM architectures, custom to unique characteristics of each type of data. Our approach takes as input a program, analyzes and finds out the proper program points, and inserts data movement instructions accordingly. Our approach manages code, stack and heap data, and reduce execution time by 14%, 52% and 80% respectively compared to their predecessors on typical embedded applications.

Other than managing local data, we also develop management techniques to manage shared data in SMM architectures. Experimental results show our approach achieve more than 2X speedup than the previous technique on average.

Acknowledgement

PhD life is the most challenging yet the most rewarding time of my life. It not only teaches me how to think critically, but also makes me realize the importance of communicating clearly with people, a skill that keeps helping me even after graduate school. It lets me learn patience and persistence. I can never finish my 7 years of study without them. Most importantly, it allows me to find the passion of my life and start a career I love.

I would never complete my PhD study without the help I received. My advisor, Professor Aviral Shrivastava, enlightened me with his insight, and never lost faith in me. My labmates, Di Lu, Bryce Holton, Yooseong Kim, and Moslem Didehban, always encouraged me, and brainstormed with me when I faced technical difficulties. Mahesh Balasubramanian, Mohammadreza Mehrabian, Shail Dave, Mohammad Khayatian, and the rest labmates all offered their help so I could focus on the defense towards the end of my PhD life. My committee members, Professor Fengbo Ren, Professor Partha Dasgupta, and Professor Carole-Jean Wu, offered their valuable opinions so that I could refine my work. Because of all these lovely people, my PhD life was so enjoyable and memorable.

Finally, I would like to thank my parents and my girlfriend Tianran. There are so many times I felt lost in life, it was always their unconditional support that helped me through.

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

Multi-/many-core processors have been widely used to improve processor performance, as gains from increasing operating frequency on uniprocessors gradually diminished. In a typical multi-/many-core processor, caches are used to store frequently accessed data to bridge the gap of processing speed and memory access latency. As a result, cache coherence is required to maintain coherence of shared data (which is cached in multiple cores), so that the semantics of applications running on the processors are not changed. Cache coherence largely improves programmability, as it hides the underlying details of the memory subsystem and creates an illusion of a single image of memory. However, it comes at a price—scaling cached-based architectures becomes increasingly difficult, as area and power overhead of implementing cache coherence increases rapidly as the number of cores grows Bournoutian and Orailoglu (2011); Choi *et al.* (2011); Garcia-Guirado *et al.* (2011); Xu *et al.* (2011). While designing scalable cache coherence mechanisms remains as an important research problem, many researchers pay attention to use more power-efficient fast memory instead of caches in processor designs. An noticeable example of such efforts is scratchpad memory (SPM).

1.1 Scratchpad Memory

An SPM is the local memory incorporated into a processor or System on Chip (SoC) architecture and controlled by software (the application itself, compiler, operating system, or a combination of them). SPMs can be found in many processors, such as in high performance computing Carter *et al.* (2013); REX Computing, Inc. (2014),

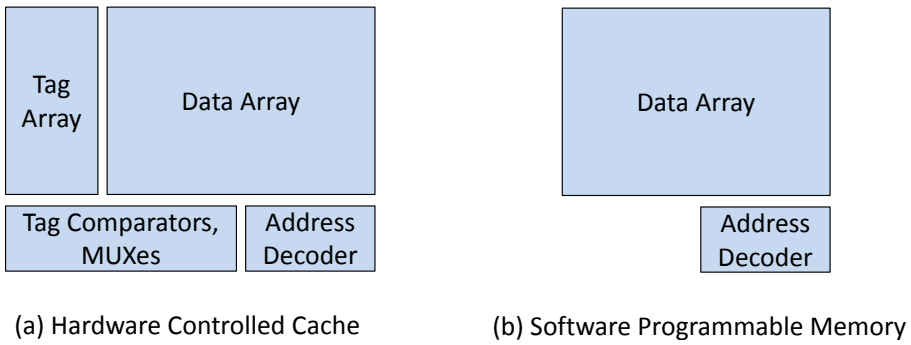


Figure 1.1: Difference between cache and SPM—the hardware view: SPM is raw memory without the hardware mechanism to manage it (as is present in caches).

gaming and multimedia processing Gschwind *et al.* (2006b), and networking Olofsson (2016). SPM is attached to the processor in much the same way as an L1 cache. However, an SPM is a piece of raw memory, in the sense that it only contains decoding and column access logic, without the complex circuitry required to achieve hardware control of replacement policies, and managing coherence (tag directory, tag look-up circuitry, etc.). As Figure 1.1 shows, while a cache stores both the data and its address, an SPM only stores data, avoiding the extra lookup circuitry. As a result, SPMs use less area yet consuming significantly less power than caches (for the same data capacity) Banakar *et al.* (2002); Redd *et al.* (2014).

Functionally SPMs are similar to caches, in that they allow for fast access to frequently used data, but with lower power and latency. However, replacing caches with SPMs comes with its own set of challenges, as in Figure 1.2. Using caches is automatic; if desired data is not present in the cache, hardware mechanisms are built to bring the requested data into the cache, potentially preventing the necessity of a repeated operation if the data is reused. However, SPM contains no such hardware mechanism to automatically bring the data that is requested to the SPM. It must be brought in explicitly through memory transfer instructions that trigger DMA trans-

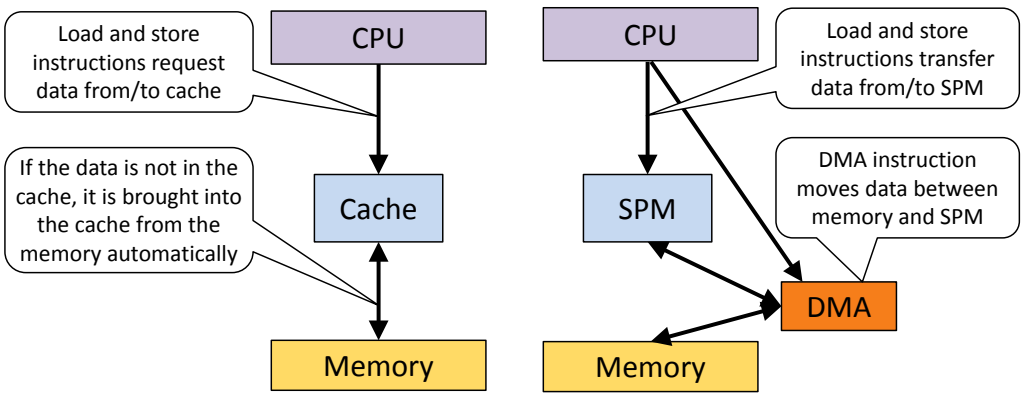


Figure 1.2: Difference between Cache and SPM—the software view: the data movement to and from the cache is performed automatically in hardware, in SPM-based systems, it must be present in the software in the form of data movement instructions.

fers. Furthermore, once data is brought in, it must be accessed using its new address in the SPM, and not the original address in the main memory.

While there are challenges in using SPMs instead of caches, the promise is the movement of application data (including code) on SPM is very flexible. Caches are a one-size-fits-all approach. They have one way of managing data, regardless of how the data is actually accessed. Whether some data is accessed randomly, or is accessed in a first-in-first-out manner, on a cache-based system, it will always be accessed in the manner implemented in hardware. On the other hand, users of SPMs can make use of application semantics and knowledge of data access patterns to create more efficient management of data, thereby enabling customization of data movement across the memory hierarchy. For example, stack data in SPMs can be managed on stack-frame level instead of cache blocks, as when a function call happens, it is likely that all the data within the stack frame will be needed during the execution of the function. By loading all the data in a stack frame at once, we can reduce the overhead for checking if the requested cache blocks during the execution of the function are already in the

SPM. In addition, if we know multiple stack frames of function calls along some path in the call graph can be held in the SPM at the same time, we can bring all these stack frames from the main memory into the SPM at once, instead of fetching each of them separately. By doing so, we can further reduce number of memory transfers, and eliminate status checking of stack frames between these calls.

1.2 Software Managed Manycore and Its Management

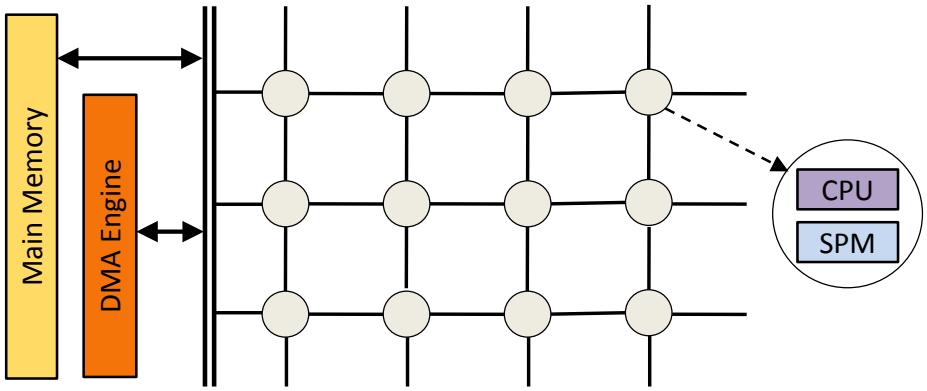


Figure 1.3: An example of SMM architectures.

Since code and data must be explicitly transferred by executing additional instructions to manage code and data between an SPM and main memory, SPM-based multi-/many-core architectures are called software managed manycore (SMM) architectures. Figure 1.3 shows an example of SMM architectures. In an SMM architecture, each core has a local SPM. A core can directly access the required code and data in slow main memory, or first load the code and data into its local SPM to improve performance. Code and data that are not immediately required can be evicted from SPM (to main memory).

To manage data on an SMM architectures, the space of an SPM is divided into different partitions, with one partition for code, stack, heap and global/static data

correspondingly (each core runs one task at a time). Different management techniques are developed to manage each type of application data. The differences of management techniques are required due to the distinct characteristics of different types of data. For example, the addresses of stack variables in a stack frame is all based on the value of stack pointer, therefore we can easily change addresses of stack frames by simply changing the value of stack pointer. On the other hand, it is more difficult to change instruction addresses at runtime as doing so would affect PC(Program Counter)-relative memory accesses. Therefore, stack management techniques on SMM architectures can move stack frames to any available SPM or memory or memory locations at runtime, while code management techniques on SMM architectures typically uses the same address for different functions, and rely on a technique called overlaying to load only the required functions (thus evicting the functions currently occupying the same memory addresses) at runtime. Similarly, we can not simply apply the same techniques of stack or code management to manage heap data, as heap objects are generally scattered in memory. We need to develop different techniques for heap management, such as software caching.

A big body of SPM management techniques for SMM architectures are compiler-directed, as inserting data movement operations by hand force programmers to trace flow of data, which can be error-prone. Early SPM management are mostly static approaches Panda *et al.* (1997); Sjödin and von Platen (2001); Avissar *et al.* (2001, 2002); Nguyen *et al.* (2005); Verma *et al.* (2003); Steinke *et al.* (2002b), which divides application data into SPM and memory. In other words, part of application data (typically most frequently accessed) goes to the fast SPM, while the rest is allocated and accessed in the slower main memory. Such techniques are limited as the size of application data grows, when most of the application data has to be directly accessed in main memory. Most recent works use dynamic techniques. Many dy-

namic techniques are driven by profiling Egger *et al.* (2008a, 2010); Ishitobi *et al.* (2010); Kandemir *et al.* (2001); Panda *et al.* (2000); Li *et al.* (2005a); Jia *et al.* (2015); Udayakumaran and Barua (2003); Dominguez *et al.* (2005); Udayakumaran and Barua (2006); Udayakumaran *et al.* (2006a); Dominguez *et al.* (2007); Verma and Marwedel (2006); Chakraborty and Panda (2012). Profilers are used to collect various information, such as loop bounds or outcomes of branches, provided representative program inputs are present. These information is then used to decide most frequently used data that can be stored in SPM, either statically or dynamically. Such approaches, while can be very efficient, cannot be applied when program behaviors change drastically when inputs vary, and are therefore lack of generality. To solve this problem, we present non-profiling-driven compiler-directed approaches to manage data movement between SPM and main memory, as in Figure 1.4. Such an approach takes as input a program, analyzes and inserts instructions for data transfers to the intermediate-representation (IR) of the program, and then generates the binary file for the SMM processors.

It is sometimes necessary to manage shared data in different cores, on top of the management of local data. This topic has been extensively studied in clusters. However, those approaches does not apply well to SMM architectures. In traditional clusters, inter-process communication is very expensive, while computation is relatively cheap thanks to powerful processors used in the clusters. Therefore, shared data management usually tries to minimize communication at the cost of introducing more computation. In SMM processors, however, inter-core communication becomes faster, while computation power of each core is weakened to save power. Instead, number of cores is increased to offset the lost of computation speed. Therefore, we developed a different approach that can significantly reduce computation overhead, even though it may increase data transfers among cores.

Figure 1.4: Our approach analyzes programs and inserts management instructions for efficient data movement between SPM and main memory.

1.3 Overview of This Thesis

This thesis develops efficient non-profiling-driven automatic compiler-directed data management techniques to manage different types of application data on SMM architectures. Chapter 2, 3, 4 and 5 explain the details and experimental results of the code, stack, heap and shared data management techniques developed in this thesis. Chapter **??** summarizes this thesis and introduces publications contribute to this thesis.

1.3.1 Efficient Code Management

One way to manage code on SMM architectures is to divide the space of each SPM into regions, and map the functions in a program into these regions. Each function is mapped to exact one region, and therefore multiple functions may be mapped to the same region, if there are more functions than regions, which is not uncommon as SPM space is typically limited, just like caches. Code management introduces overhead in the form of increased memory transfers and dynamic instructions. To reduce such overhead, we can either i) reduce memory transfers, or ii) reduce number of instructions executed. Existing code management techniques for SMM architectures focus on the first issue Lu *et al.* (2015a). In Chapter 2, we show the second issue is very important, introducing 22% extra instructions on typical embedded applications even when SPM size is larger than code size. Our technique reduces the instruction

overhead by 16% and execution time by 14% respectively, compared to managing code without solving the second issue. Our approach reduces execution time by 9% on average compared to caches, when only code management is considered, i.e. assume the accesses to the rest of application data always hits.

1.3.2 Efficient Stack Management

Stack management is also very important, as stack accesses account for around 64% of overall memory accesses Kannan *et al.* (2009) in Mibench Guthaus *et al.* (2001a), which consists of typical embedded applications. To ensure the functionality of applications, stack management techniques on SMM architectures have to i) copy stack frames between SPMs and main memories at right timing, and ii) solve the pointer corruption issue caused by changed addresses of stack variables. An existing work Lu *et al.* (2013) has solved the first problem efficiently. The solution to solve the second problem is however still relatively preliminary, focusing on correctness but not efficiency. In Chapter 3, we show an approach of efficient manage stack pointers that reduces execution time by 52% compared to the basic pointer management approach. Additional experiments show our approach reduces execution time by 12% on average compared to caches, when only stack data is considered, i.e. assume the accesses to the rest of application data always hits.

1.3.3 Efficient Heap Management

Heap management on SMM architectures are mostly based on software caching, i.e. manage data movements between SPM and data in a way emulating how a cache works. Memory accesses are intercepted and replaced by management calls that transfers data between SPM and main memory when necessary. For example, one technique Bai and Shrivastava (2013) blindly intercepts all the memory accesses

at runtime, and then filters non-heap accesses at run time. The management function treat an SPM as a 4-way set-associative cache and manages data accordingly. In Chapter 4, we develop a heap management that identifies heap objects and their alias at compile time, and eliminate most of runtime checking. Next, we reduce the instruction overhead of manage calls by emulating a direct-mapped cache, which does not require the sequential search of four entries in each set (assuming parallel search is not supported) and complex cache replacement policy. Furthermore, we inline the management calls and remove redundant steps. Finally, we show an optional optimization that relies on profiling. The experimental results show it reduces execution time of benchmarks by 80%. With the optional profile information, the reduction can be increased to 83%.

1.3.4 Shared Data Management

Last but not least, we present a technique to manage shared data among different cores in SMM architectures. Traditional shared data management favors coarse-grain data movement, such as at page level, to reduce number of communication. Such coarse-grain data management typically ends up introducing more computation. For example, if multiple cores need to write to different locations of the same page, each core needs to create a local copy to work on, and finally compare it with the original page to apply the differences accordingly. This is necessary to avoid false sharing. However, the comparison of two pages is compute-intensive and is not efficient in SMM processors. We develop a technique that records the exact location and the size of each change, so that we can notify other cores about the exact changes to avoid transferring an entire page and comparing. While it may cause higher number of communication, the expensive page comparison is saved. The technique we developed achieved more than 2X speedup than the previous technique.

Chapter 2

CODE MANAGEMENT ON SPM

2.1 Introduction

The power and area overheads of cache coherence logic increase exponentially with the increasing number of cores, posing serious challenges in designing multicore architectures Bournoutian and Orailoglu (2011); Choi *et al.* (2011); Xu *et al.* (2011). Using scratchpad memories (SPMs), on the other hand, considerably simplifies the hardware by removing circuitry for tag comparison, replacement and coherence Banakar *et al.* (2002). The simplified hardware, however, shifts the work of memory management from hardware to software and requires executing additional management instructions in software. Multicore architectures based on SPMs are, therefore, called software-managed multicore (SMM) architectures. In an SMM architecture, each core has a local SPM. Instructions or data can be transferred into an SPM to reduce access latency, by direct memory access (DMA) operations.

One way to manage instructions on a local SPM is overlaying Levine (1999). Overlaying divides SPM space into different regions, with each function allocated to one of the region. Before every function call, additional code for calling the management function must be inserted. The management function checks the SPM state to see if the called function is loaded in the SPM and if not, performs a DMA operation. Similarly, the management function needs to be called again right before the called function returns back to the caller, because the caller function might have been evicted by the called function, in case they share the SPM space.

There are two sources of overhead in such code management. The long-latency

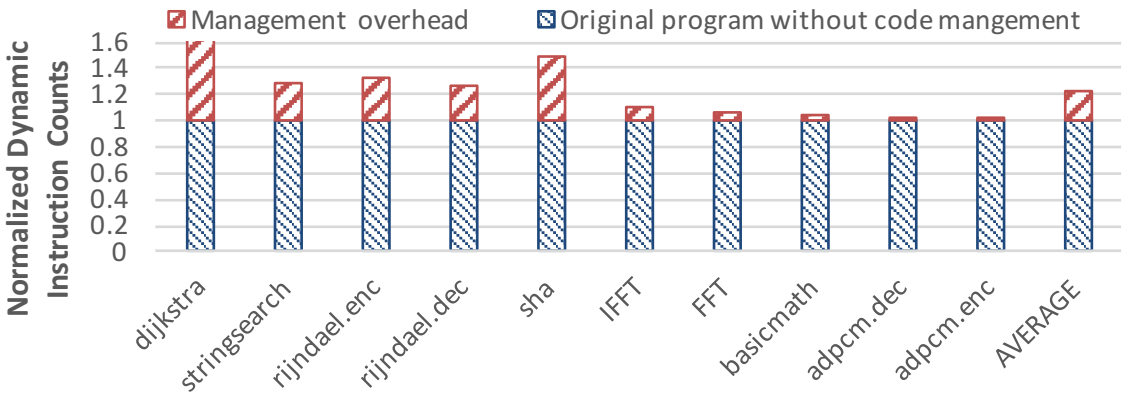


Figure 2.1: Code management increases instruction count by 22% on average even when the SPM is larger than the code size.

DMA operations are one source of overhead. The SPM allocation determines the sharing among functions, and a poor allocation scheme can increase the overhead of DMA operations by causing frequent reloading of functions. Another source of overhead comes with calling the management function at every call site. A management function call can cause a noticeable overhead even without any DMA operation, as it involves multiple table lookups for obtaining the information (allocated address, size, SPM state) of functions. If the required function is absent in SPM, the manage function will need to perform additional operations updating the SPM state after the loading operation.

Most previous code management efforts based on overlaying focus on the first overhead and try to find memory allocation schemes that avoid reloading functions frequently Pabalkar *et al.* (2008); c. Jung *et al.* (2010); Baker *et al.* (2010); Jang *et al.* (2012); Bai *et al.* (2013); Lu *et al.* (2015b). For example, allocating the caller and the called function in a loop to different memory addresses eliminates competition for SPM space between them, and thus is a desirable allocation scheme. These approaches, however, do not address the second overhead and blindly insert manage-

ment function calls to every call site even though some of them may not be necessary. For instance, even if a function is loaded into its private space in the SPM, a management function has to be called every time the function is called, only to find out that the function is already loaded in the SPM. Figure 2.1 shows such overhead of executing code management instructions *in vain*. It shows that over a set of typical embedded programs Guthaus *et al.* (2001b), 18% of executed instructions on average are from management functions. Note that this is the *lower bounds* of the overhead when the SPM size is large enough to assign a private region to every function. For smaller SPM sizes, the number of loading operations will increase as the conflicts between functions increases, thus the overhead of executing management instructions, e.g. looking up function sizes and addresses for DMA operations, and updating the SPM state after DMA.

In this chapter, we present a compiler-based approach to reduce the overhead of management function calls. Given the allocation of functions to the SPM, our analysis statically determines whether a function can be safely assumed to be loaded before each call site. If a function is guaranteed to be already loaded at a call site, we label the call site as *always-hit* and do not insert a management function call. Similarly, a call site in a loop is labeled as *first-miss*, if the called function cannot be guaranteed to be always loaded but once loaded, is never evicted until the end of the loop. In this case, we place the management function call in the loop preheader to call the management function only once before entering the loop.

The static instruction cache analysis based on abstract interpretation Ferdinand and Wilhelm (1999); Cullmann (2013) also tries to identify always-hit and first-miss cache lines. However, within nested loops, their first-miss analysis Cullmann (2013) is only able to identify cache lines that are first-miss at the outermost loop. On the other hand, our first-miss analysis can identify any first-miss call site including

those that are only first-miss within inner loops. In addition, our solution consists of two steps: i) static analysis for finding always-hits and first-misses; ii) inserting management function calls based on the analysis result. The cache analysis techniques only deal with the first part and cannot be directly used for reducing overhead of code management on SPM.

For evaluation, we use the state-of-the-art overlay-based function allocation technique Lu *et al.* (2015b), and various benchmarks from Mibench suite Guthaus *et al.* (2001b), with varying SPM sizes. The results show that our approach reduces the management overhead by 84%, and overall dynamic instruction counts by 16%. Consequently, execution time is reduced by 14% on average. In addition, our approach is able to reduce the execution time by 9% on average and up to 15% compared to hardware caching, even with conservative measurement.

2.2 Related Work

The early approaches on code management are based on static management Steinke *et al.* (2002b); Angiolini *et al.* (2004); Verma *et al.* (2004). In static management, a selected part of the code is loaded into the SPM only at loading time before the execution. These techniques can become inefficient for large programs when most of the code remains in the slow main memory.

Dynamic management techniques overcome this problem by loading instructions at run-time. A large body of these approaches Janapsatya *et al.* (2006); Verma and Marwedel (2006); Udayakumaran *et al.* (2006b); Steinke *et al.* (2002a); Egger *et al.* (2006, 2010), however, allocates only part of the code to the SPM, assuming that instructions can be directly fetched from the main memory as in ARM ARM (2004). These techniques first find a set of reloading points, and then determine the (frequently-used) code blocks to be loaded at each reloading point. Whenever the

control reaches a reloading point, the corresponding loading operation is performed without checking the SPM state. Any instruction left in the main memory has to be fetched from the main memory. All these approaches are profiling-driven, and are not applicable if representative inputs of programs are not present.

Our analysis techniques directly target non-profiling-driven code management techniques Pabalkar *et al.* (2008); c. Jung *et al.* (2010); Baker *et al.* (2010); Jang *et al.* (2012); Bai *et al.* (2013); Lu *et al.* (2015b); Kim *et al.* (2014). All these approaches work at the granularity of function and conceptually perform as a direct-mapped cache where the entire code of a function is loaded into a cache line at once. Here, each cache line is called *region*, and management techniques find a *mapping* of functions to the regions with the goal of reducing the conflicts between functions. In this chapter, we apply our approach to the latest among these techniques Lu *et al.* (2015b) and compare the performance before and after using our approach for evaluation.

It is worth noting that the key idea of our approach is not limited to function-level code management approaches. Our analysis technique can be extended for any management techniques where code blocks are conditionally loaded after checking the SPM state.

Some approaches Francesco *et al.* (2004); Egger *et al.* (2008b) require hardware support such as MMUs, which may not be available in some processors Gschwind *et al.* (2006b); Texas Instrument (2014) and therefore are not considered in this chapter.

Previous code management techniques for SMMs solve the problem of "where-to-load" each function but not "when-to-load"; management functions are inserted blindly at every call site. Even with an ideal allocation where each function is assigned to a private region, management functions are executed before and after every function call, just finding out loading is unnecessary. On the other hand, the technique presented in this chapter focuses on reducing management instructions.

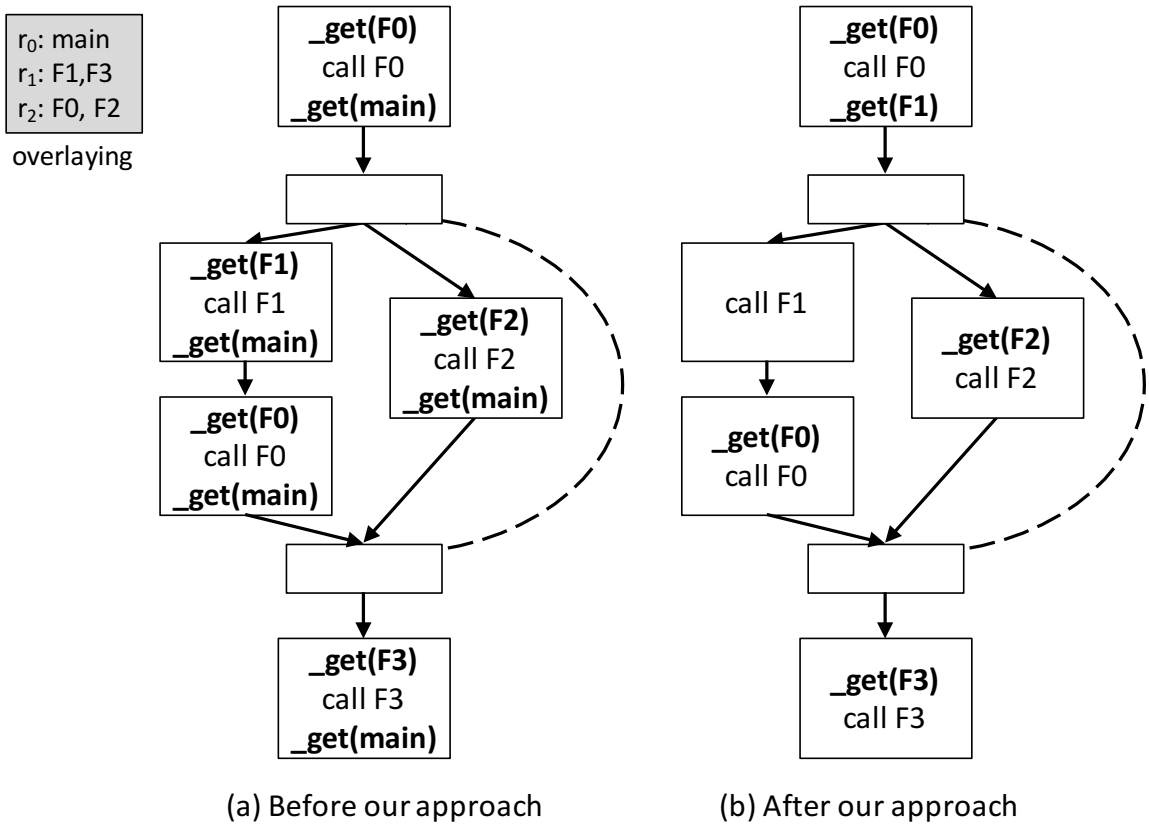


Figure 2.2: Our analysis avoids unnecessary management functions.

2.3 Motivating Example

Figure 2.2 shows that current code management techniques may insert unnecessary management function calls, and how we can avoid them. The code management function, referred as *_get* in the rest of our discussion, is inserted around each function call. The *_get* function checks if the required function is currently in the region it is allocated to. If not, it loads the function into the SPM. The SPM space is divided into three regions r_0, r_1 and r_2. Functions are allocated to the regions respectively as follow: $\{main\}$, $\{F1,F3\}$, $\{F0, F2\}$. Previous code management approaches insert code management functions around each function call as in Figure 2.2(a). However, with proper analysis, we can remove some of code management function calls as

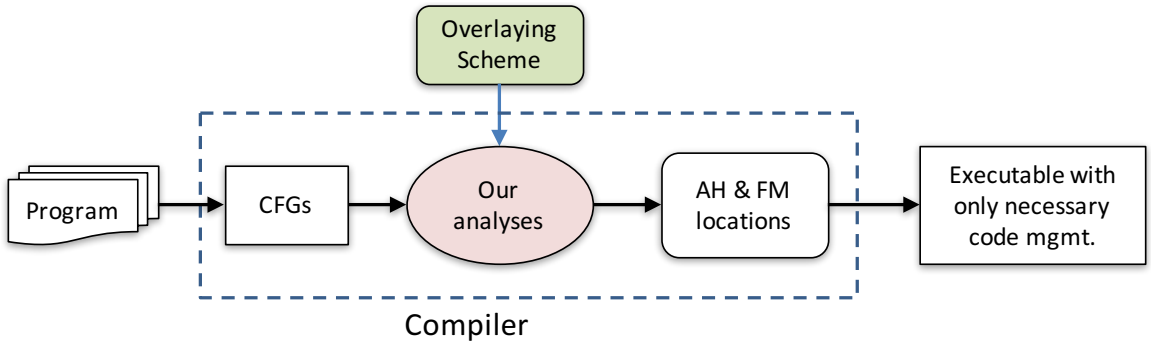


Figure 2.3: Overview of our approach.

Figure 2.2(b) shows. For example, throughout the execution, *main* will never be evicted since it is the only function mapped to r_0, so none of the calls of $\_get(main)$ after each function call is necessary. Also, since we know $F1$ is the only function called within the loop in region r_1, it will not be evicted after it is loaded into the region for the first time. On the other hand, the *_get* function called before each call to $F0$ and $F2$ in the loop are required, since the exact order of execution is not know at compile-time, so we have to conservatively assume either of them may be evicted in previous iterations from r_2.

2.4 Our Approach

Figure 2.3 shows the general flow of our approach. The compiler takes as input a program, and generates a control flow graph (CFG) for each function. All the CFGs, as well as the mapping between functions and regions, are then fed as input to our analyses. The output of our analyses are as follow: i) before each function call, whether the called function is always-hit/first-miss; ii) after the function call, whether the caller function is always-hit/first-miss. The result is then used to insert (necessary) code management functions accordingly.

2.4.1 Notation

For ease of discussion, we define the following symbols. Let the set of regions the SPM space is divided into be $R = \{r_1, ..., r_n\}$, and the set of functions called in a program (including the main function) be $F = \{f_1, ..., f_m\}$. We define a *SPM state* as the function $ss : R \to F$. Given a region id r_x, $ss(r_x)$ returns the current function that owns this region. Each SPM state describes the memory state of the SPM at a certain moment in time—it specifies the current owner function in each region. In our analyses, we maintain an *in* and an *out* SPM state for each function, basic block, and call instruction, which record the current functions in the SPM before and after the function, basic block and call instruction, respectively. In particular, each call instruction maintains an extra *int* SPM state. It is used to record the state of the SPM right before the called function returns. The difference between *int* and *out* is as follow. Let F_x be the caller function of a call instruction, and r_x be the region F_x is mapped to. If the caller function is evicted during the execution of the called function, then $int[r_x] \neq F_x$. Right before the called function returns, the caller function must be brought to the SPM in order for the execution to continue. Therefore, $out[r_x] \equiv F_x$, while $int[r_x]$ may not always be F_x. A helper function $map : F \to R$ tells which region a function is mapped to.

Two SPM states can be joined via the following function:

$$\bigcup^{ah}(ss_1, ss_2) = \begin{cases} r_i \leftarrow ss_1(r_i) & \text{if } ss_1(r_i) = ss_2(r_i) \\ r_i \leftarrow null & \text{otherwise,} \end{cases}$$

and

$$\bigcup^{fm}(ss_1, ss_2) = \begin{cases} r_i \leftarrow ss_1(r_i) & \text{if } ss_1(r_i) = ss_2(r_i) \\ r_i \leftarrow ss_1(r_i) & \text{if } ss_2(r_i) = NULL \\ r_i \leftarrow ss_2(r_i) & \text{if } ss_1(r_i) = NULL \\ r_i \leftarrow null & \text{otherwise} \end{cases}$$

where r_i denotes the i_{th} region, $1 \leq i \leq n$. The $\bigcup^{ah}$ operation keeps only a function when it appears in its mapped region on both SPM states. This is because for a function to be always-hit, it must has been loaded and not be evicted in all the possible paths leading to the program point. On the other hand, $\bigcup^{fm}$ keeps a function when it is the only possible function in its mapped region of the two SPM states, since for a function to be first-miss, all we need is to ensure that it is impossible for the function to be evicted by other functions mapped to its region once it is loaded in a loop.

2.4.2 Always-hit Analysis

Algorithm 1 shows the procedure of always-hit analysis. *simFunc* serves as the entry point of the analysis. It repeatedly calls *sim* function to statically simulates the execution of *main* function and other functions called either directly or transitively by *main* and record changes of SPM states, until the output (*out*) SPM state of each call instruction does not change any more. Initially all the SPM states are empty.

Figure 2.4 shows an example of applying the analysis from Algorithm 1 to find always-hit functions. Initially only the *main* function is in the SPM. Every time a function is called, it becomes the current function of the region it is mapped to. We assume none of $F0$, $F1$ and $F2$ calls any functions, therefore, after they return, they are still current functions in corresponding regions. When a basic block has multiple

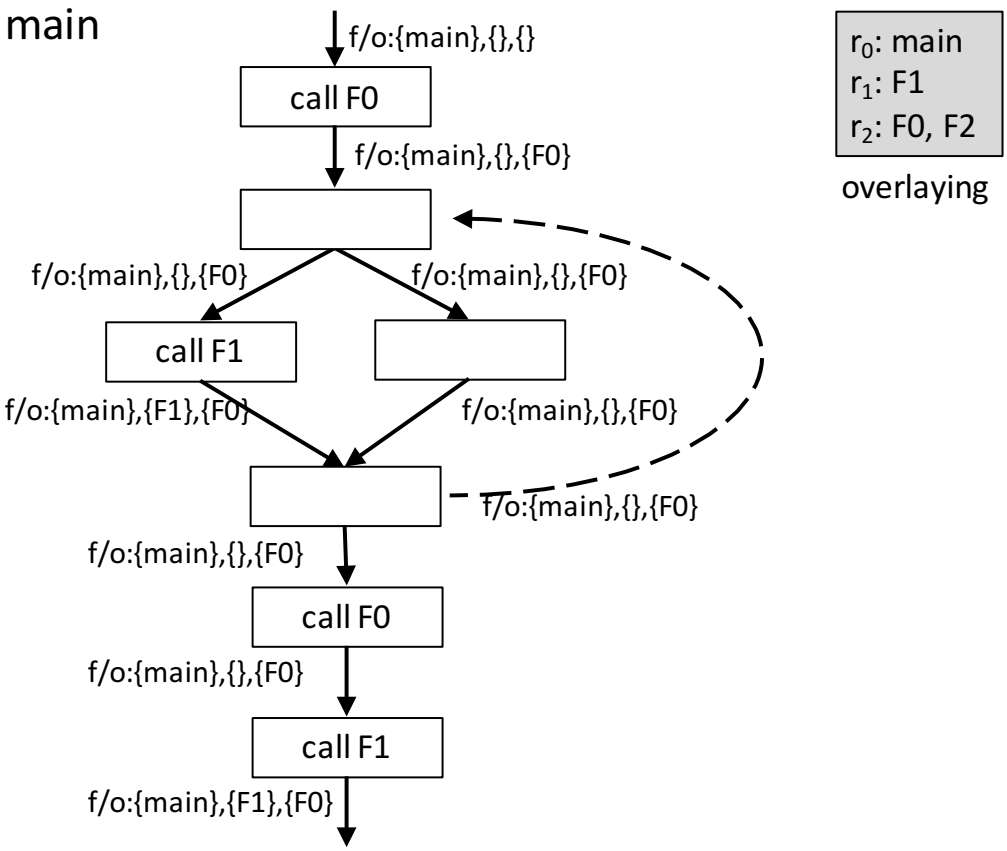


Figure 2.4: Always-hit program points in main function, given the overlaying (function-to-region mapping). The out SPM state of each basic block is shown at the edge leaving the basic block. f in the prefix denotes the output is for the first traversal, o denotes the output is for the other traversals, and f/o means the output are the same for both.

predecessors, all their output should be joined (line 10 to 14). Since Algorithm 1 ignores the output of the back edge of a loop (which are empty) when producing the input to the loop header during the first traversal ($skipBE$ is set to $true$ in line 2), the input to the loop header in the example is therefore the same as the output after the first call to $F0$. In the second traversal, the output of the back edge is used to join with the output of $F0$ before entering the loop ($skipBE$ is set to $false$ in line 4).

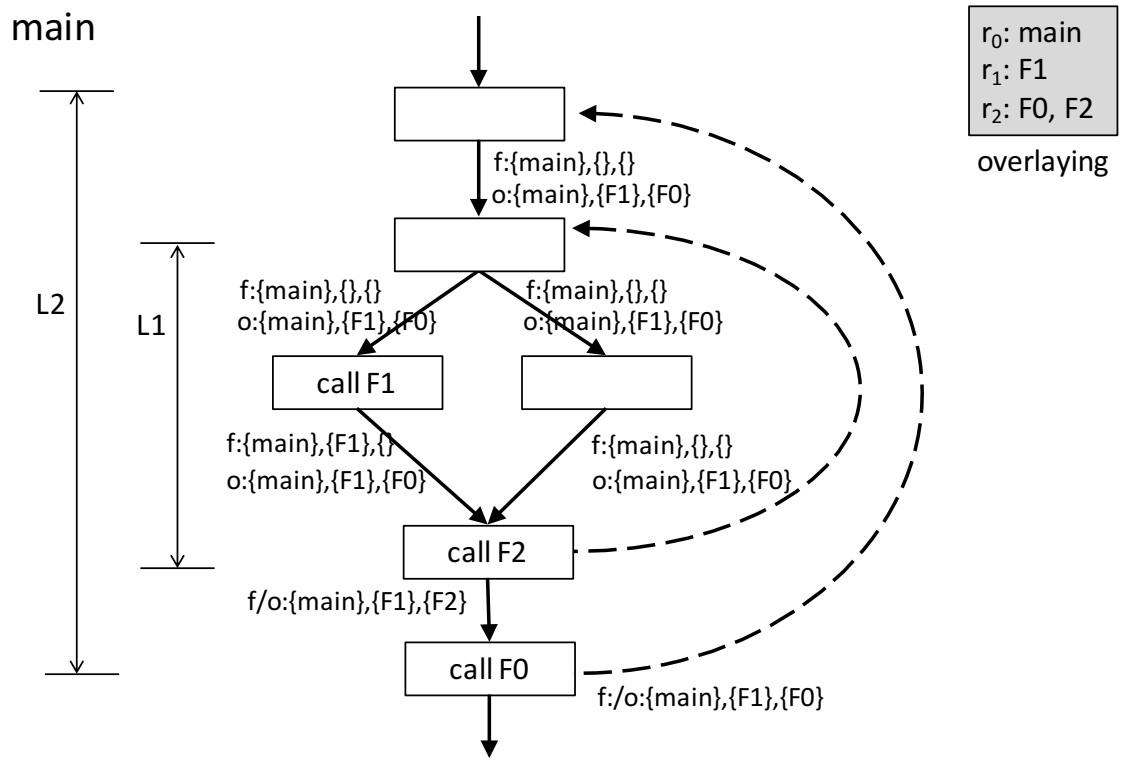


Figure 2.5: First-miss program points in loop L2 of main function.

The join operation however does not change the input, thus the other SPM states. Therefore, after the second traversal, the algorithm stops.

The identification of both always-hit and first-miss program points after the analysis stops is as follow: for each function call, if the called function is in the *in* state of a call instruction, then the program point before the function call is always-hit; if the caller function is in the *int* state, then the program point after the function call is always-hit. The program point before the call to $F0$ after the loop is categorized as always-hit, since $F0$ is in r_2 within the *in* SPM state of the function call. Therefore, we do not need to insert $\_get(F0)$ there. For the same reason, the management functions after every call that checks for *main* can be skipped.

2.4.3 First-miss Analysis

First-miss analysis is performed following always-hit analysis Algorithm 2 explains the analysis. The entry point is at *simLoop* function, where each loop L of a program is singled out and passed to $simL$ function, which statically simulates the execution of L and functions called transitively. The analysis is done on each loop individually and independent of other loops, so before the analysis for each loop starts, all the SPM states are reset as empty.

Figure 2.5 shows an example of applying Algorithm 2 to find first-miss program points in the outer loop $L2$. Again, we assume $F0$, $F1$ and $F2$ do not have any function calls. The output of the back edge of $L2$ (initially empty) is used as the input of its header, since we do not care about the SPM states before the loop (line 6). Since $L2$ must be executed within its parent function, *main* must have been brought into the SPM. Therefore, the *in* SPM state of the loop head becomes {*main*}, {}, {}. When entering loop $L1$, the output from its back edge is ignored when calculating the input to its loop header (line 10). This is because if any function is evicted in $L1$, it will not show up in *out* states of all the edges leaving $L1$. Disregarding the back edge therefore will not affect the correctness, since the eviction is already reflected in the forward edge(s). At the bottom of $L1$, SPM states of the two branches are joined by the $\bigcup^{fm}$ operation. The output of loop $L2$ after the first iteration then becomes the input of its loop header when the second traversal starts. The same process is repeated, until all the output SPM states do not change.

The program point before the only call to $F1$ are first-miss in $L2$, since $F1$ is in r_1 within the *in* SPM state of the call when the analysis for loop $L2$ stops. Notice that our first-miss analysis is done loop by loop, so while the program point before the call to $F2$ is not classified as first-miss in $L2$, it will be identified when the analysis

runs on $L1$.

The first-miss analysis based on abstract interpretation Cullmann (2013) is done for all the loops at the same time. As a result, it is not able to identify program points before $F2$ in the above example to be first-miss in the inner loop $L1$, since the analysis finds out that $F2$ will be evicted in the outer loop $L2$ by $F0$. On the other hand, since our first analysis is done loop by loop, $L2$ is not considered during the analysis of loop $L1$. Therefore, it is able to identify $F2$ to be first-miss in the inner loop $L1$.

2.5 Evaluation

2.5.1 Experimental setup

We run our always-hit and first-miss analysis on top of CMSM Lu *et al.* (2015b), the state-of-the-art function-level code management, and compare its performance with the original CMSM. We implement both approaches as transformation passes in LLVM compiler infrastructure Lattner and Adve (2004). We compile benchmarks from Mibench Guthaus *et al.* (2001b) with the passes enabled. Then, we run and collect performance statistics of generated binaries on gem5 CPU simulator Binkert *et al.* (2011b).

The CPU frequency is set to 3.2 GHz in gem5. We built an SPM along main memory, and implemented DMA operations. The cost of a DMA transfer consists of setup time and transfer time. The setup time is set to 91 nanoseconds (about 291 CPU cycles), and the data transfer is set to 0.075 nanoseconds per byte (0.24 CPU cycles) for each byte of data. These specs are borrowed to the well-known IBM Cell BE Processor Kistler *et al.* (2006).

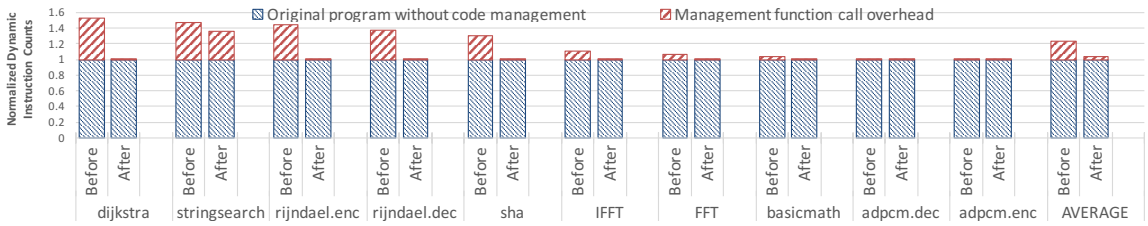


Figure 2.6: Our approach reduces the management overhead by 84% and the overall execution time by 15% on average.

2.5.2 Code Management Overhead Reduction

Figure 2.6 compares the dynamic instruction count before and after applying our techniques, both normalized to the dynamic instruction count of the original program without code management. The instruction counts for management functions calls are represented as red portions above 1, as overhead. The number of regions is set to the middle number between one and the number of functions of each benchmark. Such choice of the SPM size is for demonstration of the average-case performance between two extremes: i) the SPM space is so restrictive that all functions have to be mapped to one region, and ii) the SPM space is so large that each function can be placed in a separate region. Our approach reduces the number of instructions executed significantly,, with the management instructions in most benchmarks almost eliminated, e.g. *basicmath*, *FFT* or *rijndael.enc*. Overall, as the rightmost columns show, the normalized dynamic instruction counts are decreased by 16% on average as a result of reducing 84% of the management overhead on average.

Benchmarks that receive insignificant overall performance improvement, such as *adpcm.dec* and *adpcm.enc*, have only a few function calls so the overhead of code management was already negligible before our approach. However, the management overhead is reduced over 99% in these benchmarks. For *stringsearch*, while it has many function calls, it has only three functions, with the *main* function calling the

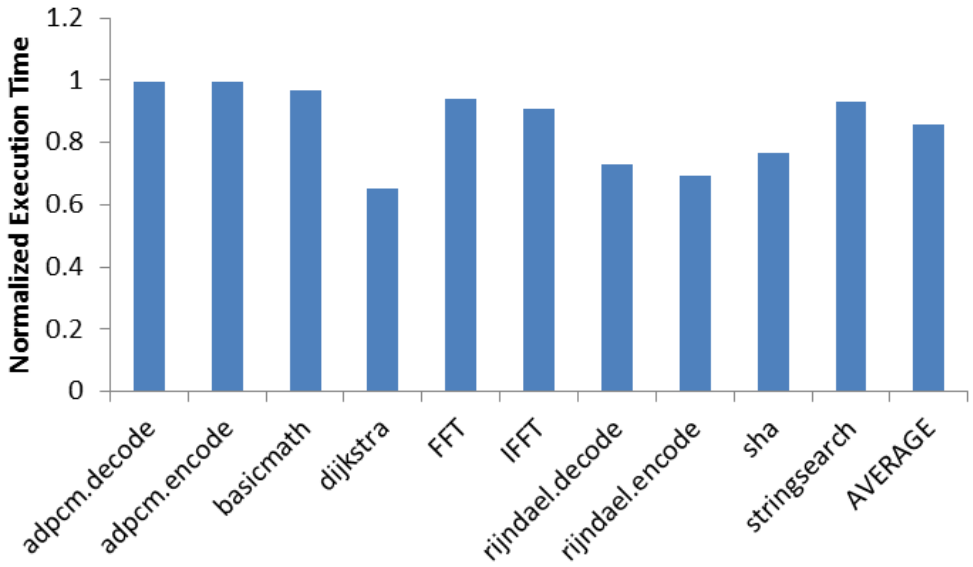


Figure 2.7: Execution time is reduced by over 14% on average.

other two in a loop. Since there are only two regions (average of one and three), two of the three functions have to be mapped to one region, causing them to evict each other at every iteration. The management function calls are necessary and cannot be reduced.

Figure 2.7 shows the reduction in execution time after using our approach. On average, the execution time is reduced by more than 14%. The reduction in execution time is less than the reduction of dynamic instruct count. This is because management overhead includes both instructions and DMAs, and our approach only reduces management instructions.

2.5.3 Comparison with Hardware Caching

We compare our approach with caching in a cache-based architecture. The cache-based system has a 2-way L1 instruction cache with 64-byte cache lines on gem5 simulator. The sizes of the SPM are the same as the experiments in section 2.5.2,

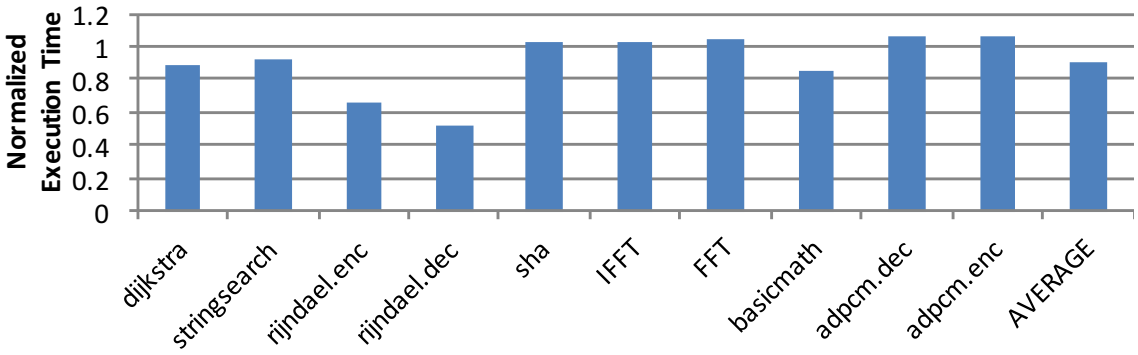


Figure 2.8: The execution times are normalized to those with hardware caching.

in which the number of regions is a half of the number of functions. Cache size for each benchmark is set to the smallest power of two that is no less than the SPM size. Cache miss latency is the same as the DMA setup time. This configuration is conservative since it leads to significantly larger cache sizes than SPM sizes in several benchmarks, *sha*, *IFFT*, *FFT*, *adpcm.dec* and *adpcm.enc*.

Figure 2.8 shows the normalized execution time of benchmarks with our approach compared to hardware caching. The overhead of code management in a cache-based architecture is measured as the number of cache misses times the cache miss penalty, while the overhead in the SMM architecture is measured as the sum of the time spent executing instructions of code management function calls and DMA cost.

In several benchmarks, using an SPM-based architecture with our approach can significantly reduce the execution time. However, caching is better in *adpcm.dec* and *adpcm.enc*, in which most of the execution time is spent on small loops that are small enough to fit in the instruction cache. However, even in these cases, the execution times are comparable, and the differences are not more than 6%.

2.6 Conclusion

In this chapter, we focus on management of code in SPM-based SMM architectures and present two analyses that find the locations where the outcomes of checking can be safely guaranteed and thus the management code can be removed or hoisted. With various benchmarks and various memory configurations, our experimental results show that our techniques can reduce the execution time by 14% on average. Using our approach on an SPM-based architecture, we observe that the execution times of benchmarks are significantly less or at least comparable to those on a cache-based architecture.

Algorithm 1 Always-hit analysis

1: **function** SIMFUNC
2: sim($main$,$true$)
3: **while** not converged **do**
4: sim($main$,$false$)
5: **function** SIM(F, $skipBE$)
6: $region = \text{map}(F)$
7: $F.in[region] = F$
8: $F.Entry.in = F.in$
9: **for** each basic block B in F **do**
10: **if** B is not $F.Entry$ **then**
11: **if** $skipBE$ **then**
12: $B.in = \bigcup^{ah}_{P\,a\,forward\,predecessor\,of\,B} P.out$
13: **else**
14: $B.in = \bigcup^{ah}_{P\,a\,predecessor\,of\,B} P.out$
15: **for** each call instruction I in B **do**
16: $I.in = B.in$ or out of the previous call
17: $I.CalledFunction.in = I.in$
18: sim($I.CalledFunction$, $skipBE$)
19: $I.int = I.CalledFunction.out$
20: $I.out = I.int[region] \leftarrow F$
21: $B.out = out$ of the last call
22: $F.out = \bigcup^{ah}_{R\,a\,return\,instruction} R.out$

Algorithm 2 First-miss analysis

1: **function** SIMLOOP
2: **for** each loop L in the program **do**
3: **while** not converged **do**
4: simL(L)
5: **function** SIML(L)
6: $L.Header.in = L.BackEdge.output$
7: $L.Header.in[region] = F, the\ parent\ function\ of\ L$
8: **for** each basic block $B \in L$ **do**
9: **if** B is not $L.Header$ **then**
10: $B.in = \bigcup^{fm}_{P\ a\ forward\ predecessor\ of\ B,\ P \in L} P.out$
11: **for** each call instruction I in B **do**
12: $I.in = B.in$ or *out* of the previous call
13: $I.CalledFunction.in = I.in$
14: sim($I.CalledFunction$, $true$)
15: $I.int = I.CalledFunction.out$
16: $I.out = I.int[region] \leftarrow F$
17: $B.out = out$ of the last call

Chapter 3

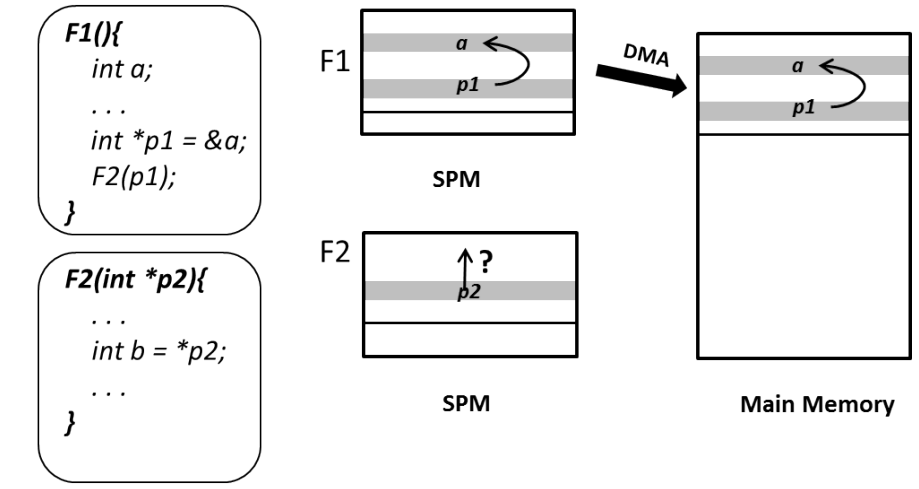
STACK MANAGEMENT ON SPM

3.1 Introduction

Low-power, yet high core-count embedded processors cannot afford the overhead of coherent caches Bournoutian and Orailoglu (2011); Choi *et al.* (2011); Garcia-Guirado *et al.* (2011); Xu *et al.* (2011). The scratchpad memory (SPM) based system is a promising alternative, as it provides a fast, low-power, and scalable memory hierarchy—the SPM has 34% less area and consumes 40% less power than a cache of the same capacity Banakar *et al.* (2002). Using SPMs instead of caches not only improves power, but also greatly simplifies the hardware design (and verification). SPMs shift the task of data management from hardware to the software, and therefore, multi-core architectures with SPM-based memory hierarchy are termed Software Managed Multicore (SMM) architectures.

In SMM architectures, a core has to fetch data it needs to its local SPM before accessing it. Therefore we need techniques to manage data transfers between the SPM and the main memory. Among all the different types of data (heap, stack or global) to manage, optimized data management for stack data is especially important for performance. Kannan *et al.* (2009) shows (via profiling) that stack accesses account for around 64% of overall data accesses in Mibench, a benchmark suite of typical embedded applications Guthaus *et al.* (2001c).

State-of-the-art techniques to manage stack data on SMM architectures move stack data between SPM and main memory at the function call level. Therefore, these techniques need to solve two inter-related problemsBai *et al.* (2011). **i) Stack**



(a) **F2** access the variable **a**, a stack variable defined in function **F1**, through pointer **p2**

(b) When **F2** is being executed, if the stack frame of **F1** has been evicted to the main memory, then pointer **p2** points to an invalid address.

Figure 3.1: Pointer management problem.

frame management: stack frame of the function that is going to be executed must be brought into the SPM before it executes, and the stack frames of the functions that are not immediately needed may be evicted to the main memory. **ii) Pointer management:** if a stack frame of a function was evicted to the main memory, and the currently executing function accesses a local variable of the evicted stack frame (typically through a pointer), then the access is invalid, as shown in figure 3.1. This is because the pointer still contains the address of the local variable in the SPM before it is evicted. It is therefore vital to correct the address of the pointer, as otherwise the result of the execution will be incorrect.

A previous work, Bai *et al.* (2011) solves the problem of pointer management, by instrumenting the code to translate the pointer address at each definition and use—A definition refers to the write of a new value to the pointer, while a use refers to the read of the value defined by the reaching definition or the last write. While this enables correct execution, it incurs high performance penalty. In this chapter, we

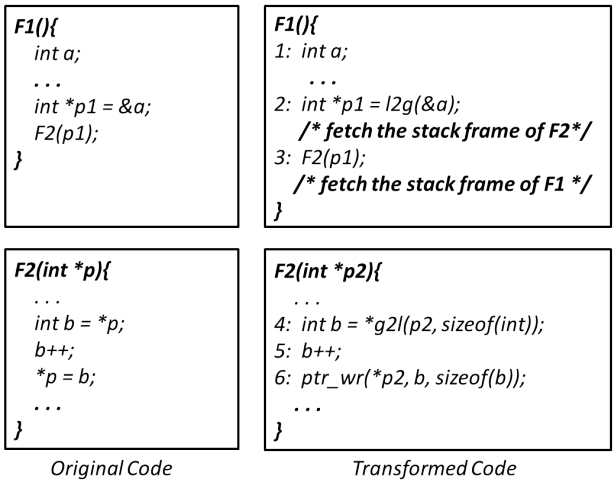
present an efficient compiler technique for managing pointers to stack data on SMM architectures. The two key ideas of our approach are: i) instead of translating the pointer address at each use of the pointer inside a function, we translate it only once when it is passed as the argument of the function. As a result, our technique is able to remove a significant portion of the overall translations. ii) if the stack frame of the function whose local variables are being accessed through pointers is guaranteed to be present in the SPM, then when any of the pointers is accessed, no translation is needed.

Experiments on benchmarks from the MiBench suite Guthaus *et al.* (2001c) show that our approach almost completely eliminates the pointer management overhead, and results in 52% reduction of the average execution time, as compared to the state-of-the-art pointer management technique Bai *et al.* (2011) on top of the state-of-the-art stack frame management technique Lu *et al.* (2013) on SMM architectures. We also compare the performance of our stack pointer management on SPM with that on a cache-based architecture. Even with conservative estimates, stack data management on SPM outperforms stack data management on a cache-based architecture by 12% on average.

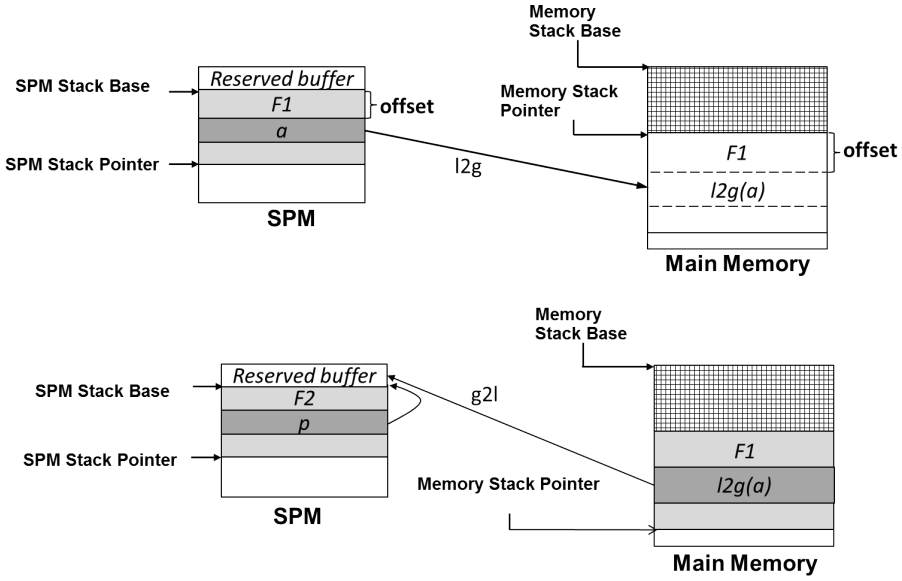
3.2 Related Work

Stack management techniques in general can be divided into static approaches and dynamic approaches. Static approaches Avissar *et al.* (2002); Verma *et al.* (2003); Nguyen *et al.* (2005) map the most frequently used data to SPM and keep the allocation fixed throughout execution, while dynamic approaches allow the changes to the locations of stack data at run-time. Static approaches do not perform well since they do not take dynamic program behaviors into consideration. As a result, most recent works focus on dynamic SPM management techniques.

Many dynamic techniques Mamidipaka and Dutt (2003); Poletti *et al.* (2004); Park *et al.* (2007); Cho *et al.* (2007); Dominguez *et al.* (2007); Kandemir *et al.* (2001); Li *et al.* (2005b); Udayakumaran *et al.* (2006c); Gauthier and Ishihara (2011); Kannan *et al.* (2009); Bai *et al.* (2011); Lu *et al.* (2013) have been developed to manage stack data on SPM. Mamidipaka and Dutt (2003); Poletti *et al.* (2004); Park *et al.* (2007); Cho *et al.* (2007) introduce new hardware functionality to manage the SPM, while the interest of this chapter lies in providing software solution to simplified hardware. Among software solutions, Kandemir *et al.* (2001) and Li *et al.* (2005b) target on arrays specifically, Dominguez *et al.* (2007) mainly focuses on managing stack data for recursive functions on SPM, while we manage all the data in stack. Udayakumaran *et al.* (2006c) and Gauthier and Ishihara (2011) both rely on profile information, therefore the input has to be representative for either of them to deliver high-quality output, which is generally difficult. In addition, both approaches have limited support for pointer management. Udayakumaran *et al.* (2006c) relies on pointer analysis to identify and translate the address for any pointer that refers to the variable that is moved from the main memory to the SPM. If the point analysis fails to identify any such accesses through pointers and thus the requested memory addresses are not translated, the execution may fail, since these accesses end up accessing incorrect locations. Gauthier and Ishihara (2011) simply does not support using stack pointers as call arguments. In this chapter we are interested in generic approaches of stack management that do not rely on profiling or pointer analysis, and are able to manage pointers correctly. To our best knowledge, only the Circular Stack Management (CSM) Kannan *et al.* (2009); Bai *et al.* (2011); Lu *et al.* (2013) approaches provide such solution. In this chapter we will compare our work with Lu *et al.* (2013), which is the latest CSM work with the best performance.



(a) Code transformation.



(b) Illustration of pointer management functions.

Figure 3.2: The way pointer management functions work.

3.3 Background

All the CSM techniques use the pointer management presented in Bai *et al.* (2011). The pointer management maintains two stack pointers: one in SPM and one in main memory (assume the stack grows from the higher address (stack base) to the lower address (stack top)). It uses three pointer management functions: *l2g*, *g2l*, and *ptr_wr*.

Figure 3.2 explains the functionality of these pointer management functions. Figure 3.2a shows the original code and the transformed code with pointer management. Figure 3.2b illustrates *l2g* and *g2l* calls at line 2 and line 4 in the transformed code respectively, assuming the SPM is not large enough to hold both the stack frames of function *F1* and *F2*. When *F1* calls *F2*, the stack frame of *F1* must be evicted from SPM to the main memory to make space. The SPM address of stack variable *a* defined in *F1* which is passed to *F2* will become an invalid reference by the time it is accessed, since the entire frame of *F1* (thus the the stack variable *a*) will have been moved to the main memory. In this case, *l2g* function should be called on *a* in *F1* to calculate the address of the actual location of *a* in the main memory (line 2 in the transformed code in Figure 3.2a). Notice when *l2g(a)* is called, the stack frame of *F1* has not been evicted to the main memory yet. Therefore, the value of *l2g(a)* indicates the memory location *a* will be moved to. At that time, *l2g(a)* is smaller than the value of the memory stack pointer, and their distance is the same as the difference between the value of the SPM stack base and the SPM address of *a*, as *offset* in Figure 3.2b indicates (the upper figure of Figure 3.2b). After *F1* is evicted to the main memory, the value of memory stack pointer is decreased by the size of the stack frame of *F1*. Consequently, *l2g(a)* refers to the actual location of *a*, which becomes larger than the value of memory stack pointer (the lower figure of Figure 3.2b).

The result of *l2g(a)* is passed as an argument to *F2* as its parameter *p*. *F2* then calls *g2l* function before dereferencing it (line 4 in the transformed code). Since cores cannot access main memory directly, *g2l(p)* allocates a local buffer in SPM, followed by a DMA instruction to read the value from the main memory location specified by *p*—or equally l2g(&a)—and then return the address of the local buffer. The correct value can then be read from the local buffer. Finally, *F2* modifies the value pointed by *p*, and calls *ptr_wr* to write back the modification from SPM to main memory (line 6 in the transformed code).

Pointer management functions will not alter program semantics. Consider the example in Figure 3.2 again, but this time we assume that the SPM is large enough to hold the stack frames of both *F1* and *F2*. Therefore, when *F2* is called, the stack frame of *F1* is still in the SPM, and we can safely remove the stack frame management around line 3 at the transformed code in Figure 3.2a. In such a case, the call to *g2l* in *F2* (line 4) will do nothing but reverting the address translation done by the *l2g* function in *F1* (line 2), and reading from the SPM address of *a*, which is the input to the *l2g* function. Similarly, *ptr_wr* will revert the translation and write to the SPM address of *a* directly. Whether a stack variable has been evicted from the SPM to the main memory can be told as below. If an address that is passed to *g2l* or *ptr_wr* is smaller than or equal to the value of memory stack pointer, then the stack variable the address refers (thus the enclosing stack frame) is still in the SPM. In this case, *g2l* or *ptr_wr* just need to revert the address translation done by *l2g* and read from or write to the SPM address. Otherwise, if the address is greater than the value of the memory stack pointer, then the stack variable has been evicted, and *g2l* and *ptr_wr* will go ahead and perform required DMA operations. Therefore, even though pointer management from Bai *et al.* (2011) unnecessarily inserts extra pointer management functions, it can still ensure the correctness of the execution of programs.

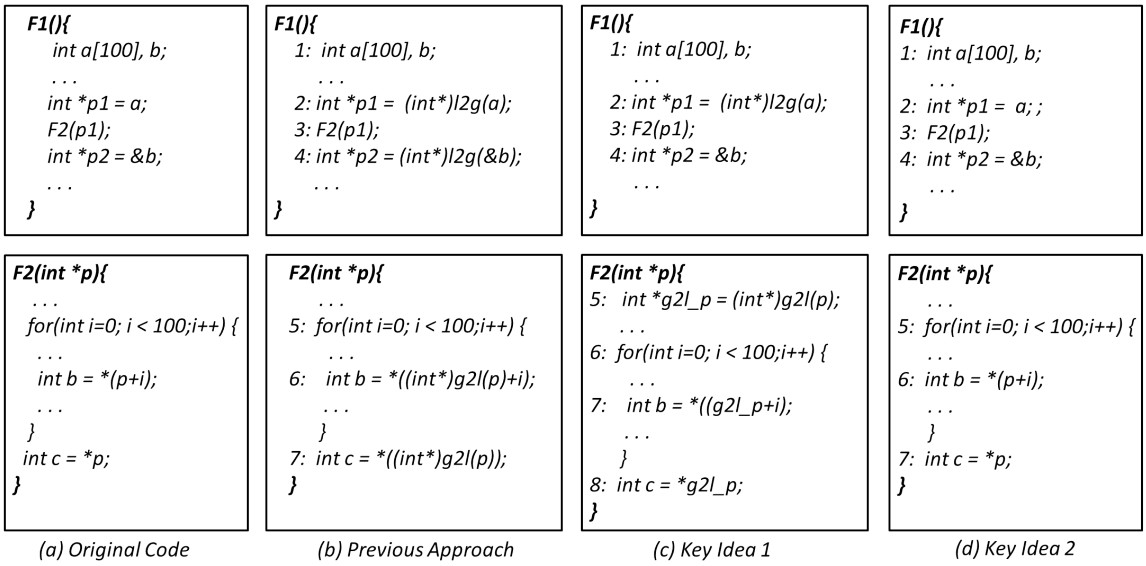


Figure 3.3: The key ideas of our approach.

Pointer management from Bai *et al.* (2011) solves the problem for correctness, but not for performance. On the other hand, our approach removes unnecessary calls to pointer management functions and improves performance of applications noticeably.

3.4 Key Ideas of Our Approach

While the state-of-the-art pointer management from Bai *et al.* (2011) solves the pointer corruption issue correctly, it calls the *l2g* function at every definition of stack data pointers, and the *g2l* function on every use of the pointers, and results in unnecessary calls to pointer management functions, which eventually slows down the execution of programs.

Figure 3.3a and 3.3b show the original code and the code with the pointer management from Bai *et al.* (2011). The calls to *l2g* in line 4 in Figure 3.3b is not necessary, since the pointer *p2* is only used in the same function the variable *b* is defined. Meanwhile, although the calls to *g2l* in line 6 is necessary, it can be promoted to be outside of the loop to avoid repeated computations. Notice the calls to *g2l* in

line 6 can not be eliminated or reduced by standard compiler optimizations such as common subexpression elimination or loop invariant code motion. This is because *g2l* function needs to access some global states (implemented as global/static variables), such as the current values of the stack pointers, which could be changed by function calls and stack frame management between any two consecutive *g2l* calls. These interactions with global states prevent standard compiler optimizations from removing or relocating *g2l* calls, since the compiler cannot guarantee the changes will not cause any unexpected side effect to the semantic of programs.

This work aims to reduce these overheads based on two key ideas: i) we only manage pointers when they are used as call arguments instead of each of the uses, so that we only need to translate once at the caller and the called function respectively. ii) if the stack frame of a function is definitely in the SPM, then any accesses via pointers to the local variables in the stack frame do not need management. Figure 3.3c demonstrates the first idea. Instead of calling *l2g* on every definition of pointers (line 2 and 4 in Figure 3.3b), we only call *l2g* once in *F1* when it calls *F2* and passes a pointer-type argument (line 2 in Figure 3.3c). Also, we only call *g2l* function once at the beginning of *F2* and reuse the result (line 5, 7 and 8 in Figure 3.3c), instead of calling it for every use (line 6 and 7 in Figure 3.3b). The *g2l* function is called outside the for loop to reduce overhead. Figure 3.3d shows the further optimized code with the second idea. If we know for sure the stack frames of *F1* is in the SPM when *F2* is called, then no stack frame management is needed, neither the pointer management—the references to the array a of *F1* through pointer p in *F2* will be guaranteed to access the correct locations, as the stack frame of *F1* is not moved. The code in Figure 3.3d becomes exactly the same as the original code. In other words, it eliminates the overhead of stack management—both stack frame management and pointer management.

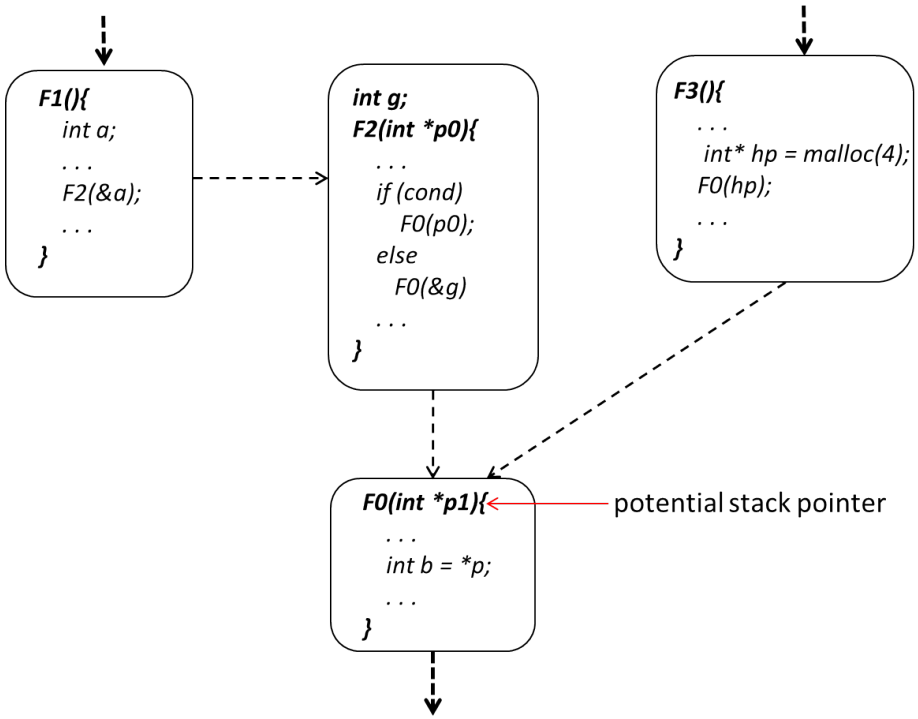


Figure 3.4: Identification of the potential pointer to stack.

3.5 Details of Our Approach

3.5.1 Steps of Our Approach

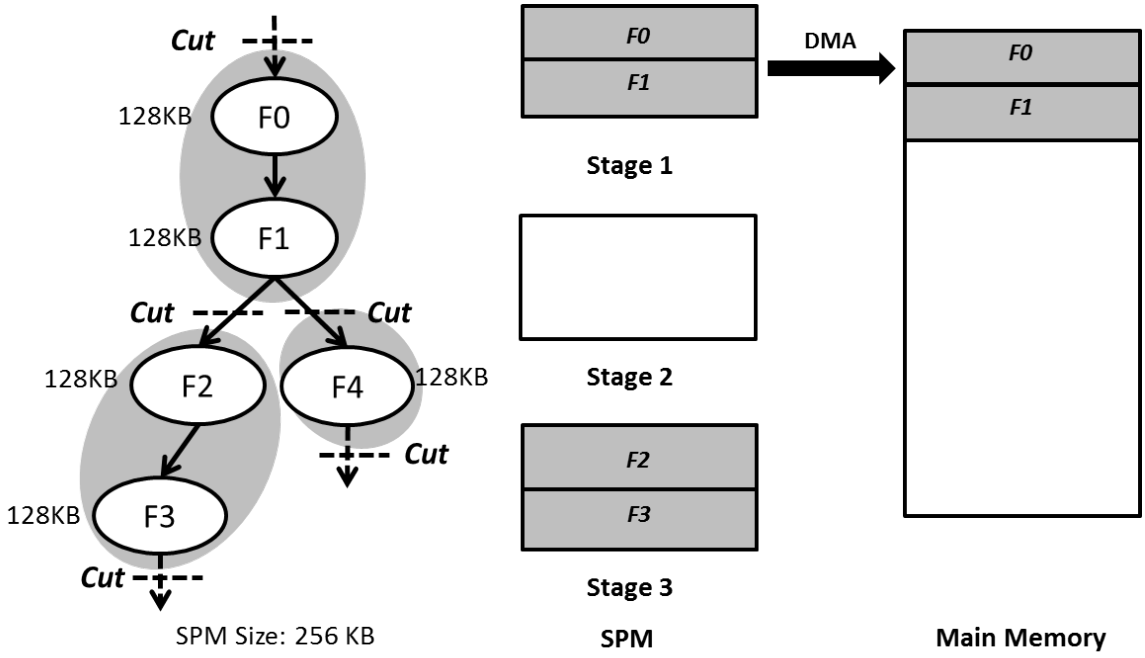
To achieve the efficient pointer management, our approach takes three steps. First, we need to decide if any pointer-type arguments of a function can be potential references to stack variables. Second, we will run the analysis to divide function calls in the call graph into groups, so that all the stack frames of each group can fit into the SPM at once. In the last step, we insert pointer management functions properly based on the previous analyses.

Identifying Stack Data Pointers at Function Calls

First of all, we perform an inter-procedural analysis to find out if any pointer-type arguments at function calls are potentially referring to stack variables. A function may be called at multiple locations during the execution of a program, therefore the same parameter may refer to multiple arguments that can be stack, heap or global variables, depending on the control flow at run-time. Consider the call graph in Figure 3.4. When *F0* is called, the type of the argument that is referred by the parameter *p1* may have three different types. If the control flow comes from *F2*, and *cond* is evaluated to be *true* in *F2*, then the argument passed to *F0* is a pointer to stack variable *a*, which is defined in *F1* and passed to *F2* when it is called; Otherwise, if the *cond* is evaluated to be *false*, then the argument is a pointer to the global variable *g*. If the control flow comes from *F3* to *F0*, then *p1* in *F0* refers to the heap data referred by *hp* in *F3*. Since the actual control flow is not known at compile-time, we have to conservatively assume *p1* refers to a stack variable.

To accomplish such analysis, we first go though all the functions and identify all the pointer-type formal parameters. Once we find such a parameter, we check all the call sites of the function, and find out all the possible arguments. Any of these arguments that refers to stack data needs to be managed.

Several cases pose challenges for this analysis. When a pointer to stack data is passed as an argument to a recursive function, then we need to call *l2g* on the pointer, and call *g2l* on the result of *l2g* function that is passed to the called function in the called function. This is because we do not know how many times the recursion will happen at compile time, so we need to conservatively assume the stack frame of the caller is evicted when the called function is executed. When the type of the variable a pointer refers to cannot be identified, we conservatively assume it is a stack pointer



(a) The function calls in the call graph are divided into three groups, with **F0**, **F1** as the first group, **F2**, **F3** as the second group, and **F4** as the last group.

(b) Before the execution of **F2**, the stack frames of the group **F0** and **F1** in the SPM is evicted to the main memory at once to make space.

Figure 3.5: The analysis to find out at which stack frames will exist in SPM at the same time.

and manage it.

Identifying Coexistent Stack Frames

This step decides which stack frames can exist in SPM at the same time. We use the same analysis from Smart Stack Data Management heuristic (SSDM)Lu *et al.* (2013). Here we just explain the high-level idea. Details of the algorithm can be found in Lu *et al.* (2013).

The general idea of SSDM is that instead of managing stack frames one at a time at every function call, we can manage multiple stack frames of consecutive function

calls along any path of a call graph at the same time. Instead of evicting the stack frame of the caller function from the SPM to main memory to make space for the called function whenever a function call happens, we can keep allocating SPM space for stack frames of function calls, and perform the stack frame management all at once only when there is not enough SPM space.

In the given call graph in Figure 3.5, the size of the stack frame of each function is 128 KB, and the size of the available SPM space is 256 KB. Therefore, the available SPM space can hold two stack frames at once. Assume the SPM is empty at first, then in the given call graph, we know the stack frame management are only necessary when *F1* calls *F2*, or when *F1* calls *F4*. Any other function calls do not need such management. For example, right before *F0* calls *F1*, the SPM only keeps the stack frame of *F0*, which takes up 128 KB space. The spare space in the SPM is large enough to hold the stack frame of *F1*. Therefore, no stack frame management is needed when this call happens. As a result, we can divide the call graph into three groups, {*F0,F1*}, {*F2,F3*}, and {*F4*}. The three groups are separated from each other by the dash lines in the figure, or what are termed *cuts*. Each cut between two adjacent groups indicates the need for inserting stack frame management functions. When any function call crosses a cut, the stack frames of the group the caller is in (currently in the SPM) are moved to the SPM, and the stack frames of the group the called function is in are brought from the main memory to the SPM. For instance, before *F1* calls *F2* in Figure 3.5, the SPM holds the stack frames of *F0* and *F1*. When *F1* calls *F2*, the stack frames of *F0* and *F1* are evicted to the main memory, and the stack frames of *F2* and *F3* are brought to the SPM.

Once we divide the call graph into different groups, we know the stack frame management is only needed for function calls that crosses any two different groups. This is true for pointer management as well, since pointer management is necessary

only if stack frames are moved due to stack frame management.

Our analysis initially places a *cut* on each edge of the call graph, which specifies the need of stack management (both of stack frame management and pointer management) for the call represented by the edge, and then greedily remove the cut which will result in the greatest reduction of stack management overhead—for instance, inserting management functions within a loop should be avoided as far as possible—while not violating the constraint that the sizes of stack frames between any two cuts should not be greater than the available SPM space, until it can not find any such cut.

Inserting Pointer Management Functions

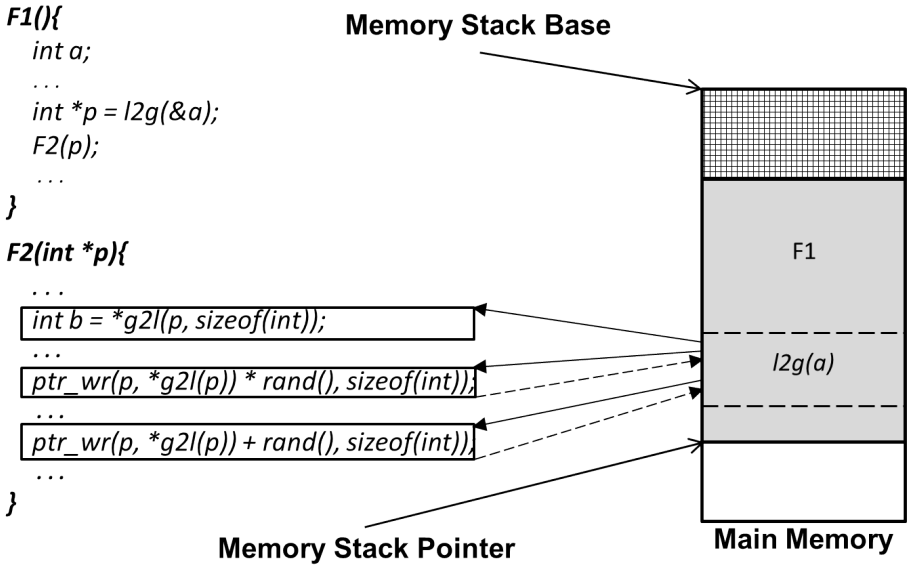
Once we have the necessary information ready, we can decide where to insert pointer management functions. We first go through the call graph and check (1) if any function call passes any pointers that may refer to stack variables (from the analysis done in 3.5.1), and (2) if the stack frames that enclose these stack variables are in the SPM when the pointers are accessed in the called function, or in other words, if the called function that accesses the pointers belongs to the same group of the function that defines the stack variables (from the analysis done in 3.5.1). Upon the confirmation of both conditions, we need to call *l2g* function on these pointers; otherwise, if any such pointer is not referring to a stack variable, or the stack variable the pointer refers to is in the SPM, then no pointer management is required for this pointer.

If we call *l2g* on a pointer-type argument on any call site of the called function, we need to call *g2l* and *ptr_wr* function in the called function for reads and writes to the pointer respectively, since we do not know which call site will the control flow comes at compile-time. While this may cause unnecessary calls to *g2l* and *ptr_wr* functions, we

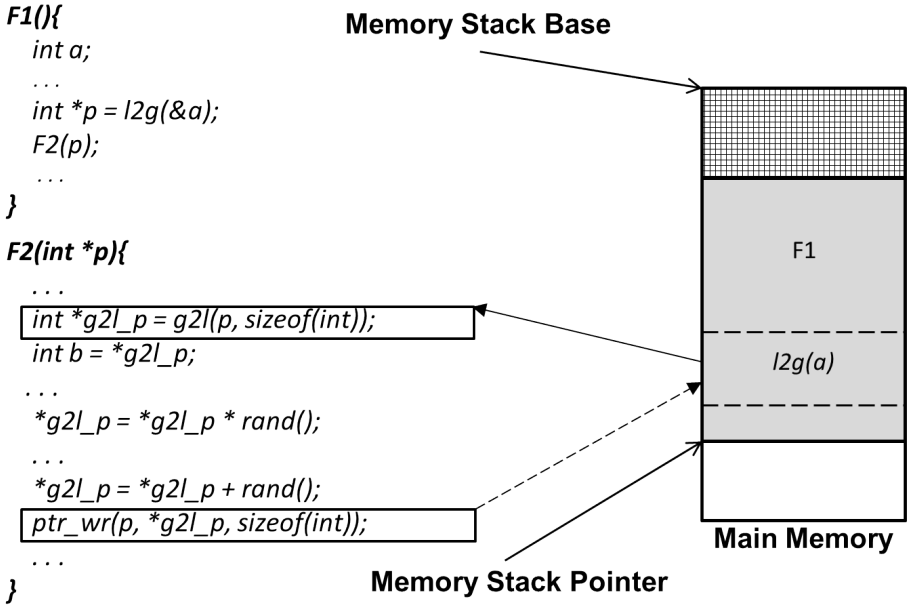
have explained that extra pointer management functions will not affect the correctness of programs. On the other hand, if we do not conservatively insert these pointer management functions in the called function, the correctness of execution will not be guaranteed, since there is a chance that the control flow may come from the caller function with *l2g* function calls at run-time.

As an optimization, we reuse the local buffer created by *g2l* function, in contrast to creating new buffer and destroying it every time by the previous approach. Figure 3.6 shows an example. The previous pointer management Bai *et al.* (2011) will call *g2l* and *ptr_wr* function on each read and write to stack data pointer *p* in *F2* respectively, even if these memory accesses are to the same memory location. On the other hand, our approach only inserts *g2l* before the first read and *ptr_wr* after the last write of *p*, and redirect the other memory request to the local buffer *g2l_p* created by the *g2l* function call. With such a policy, we can avoid redundant memory allocation and DMA requests.

When *g2l* or *ptr_wr* function is called, the compiler needs to pass the size of the stack variable in case of triggering DMA transfers. When there are multiple possible sizes, we need to use the maximal possibility. For example, a function may be called at two different cites with two array-type arguments of different sizes. In this case, since we do not know from which call site control will flow at run-time, we have to use the size of the larger array. While this approach may transfer more than necessary data if at run-time the control flow comes from the function with the smaller array, using the maximum size will not affect the correctness of the program being executed.



(a) Previous pointer management calls g2l and ptr_wr functions on every read and write to stack variables.



(b) Our pointer management calls the g2l function before the first read and ptr_wr after the last write, and reuse the local buffer for other accesses to the same memory location.

Figure 3.6: Compared to previous pointer management, our approach reuses the local buffer created by g2l function and saves management overhead.

3.6 Experiments

3.6.1 Improvement Over The State of The Art

We compare our pointer management with the state-of-the-art pointer management presented by Bai *et al.* (2011), on top of the latest stack frame management technique Smart Stack Data Management (SSDM) Lu *et al.* (2013). We implement the two approaches of stack management as passes in LLVM compiler infrastructure Lattner and Adve (2004). We compile benchmark applications from Mibench benchmark suite Guthaus *et al.* (2001c) with each of the two LLVM passes, then run and collect performance statistics of the execution of generated binaries in the Gem5 CPU simulator Binkert *et al.* (2011b).

The compilation process of benchmarks is shown in Figure 3.7. All the compilations in our experiments are done with O3 optimization on. The LLVM passes are implemented at the Intermediate Representation (IR), a transitional stage between

Table 3.1: Overhead of pointer management

	Previous Pointer Management			Our Pointer Management						Hardware Caching
benchmark	#l2g	#g2l	#ptr_wr	#l2g	#g2l	#ptr_wr	#DMA	Overall DMA Size	#Management Instructions	#L1D Misses
adpcm.decode	3428	2740	1370	0	0	0	0	0	0	30
adpcm.encode	3427	2740	1370	0	0	0	0	0	0	63
CRC	1368874	2737731	1368866	2	2	2	2	160	156	5361
dijkstra	90548	45309	44925	0	0	0	30758	1477664	784309	51964
patricia	104017	52763	3801	0	0	0	4902	436896	124551	274607
rijndael.decode	136447	1422796	11	1	1	0	2	2336	87	244983
rijndael.encode	155940	1442301	28	1	1	0	2	2336	87	244983
sha	5041	19827	12270	0	0	0	2	608	50	1578
stringsearch	606	798	57	0	0	0	0	0	0	756
susan.corners	50	242719	103	5	4	3	44	1703104	1331	36
susan.edges	50	550091	2716	5	4	3	44	1703104	1331	37
susan.smoothing	48	1630815	1535	7	6	4	46	2423392	1459	29

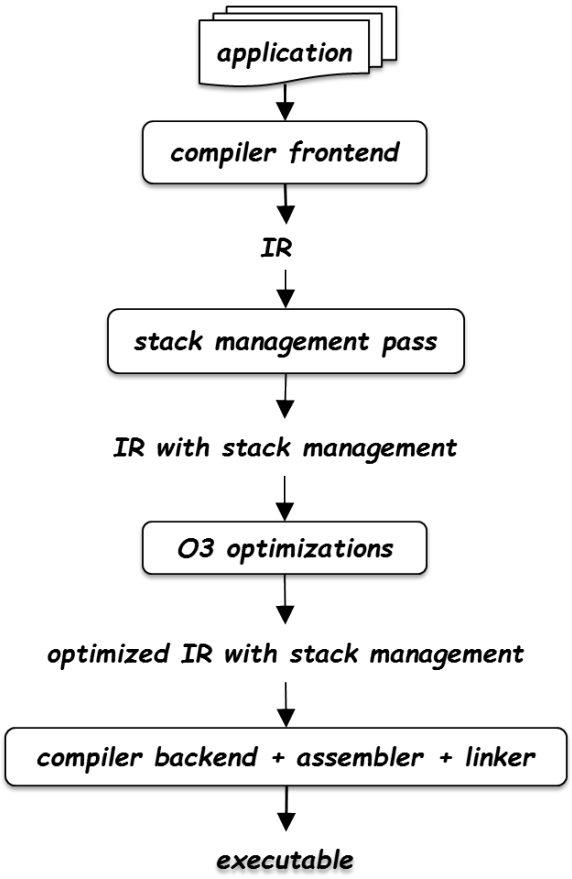


Figure 3.7: The compilation process of benchmarks used in experiments.

the translation from source code to machine language. In other words, our passes are independent of the Instruction Set Architecture (ISA) used, and should work with different compiler back ends for code generation.

We build the SPM aside the main memory, and implement a DMA instruction for data transfers between them in the Gem5 simulator. The DMA cost in our experiments consists of the start-up cost and the transfer time. The start up cost includes all the time spent setting up the DMA transfer, and the transfer time is the time spent on transferring the requested data, which can be calculated by dividing the size of the data by the bandwidth. The CPU frequency is set to 3.2 GHz in the

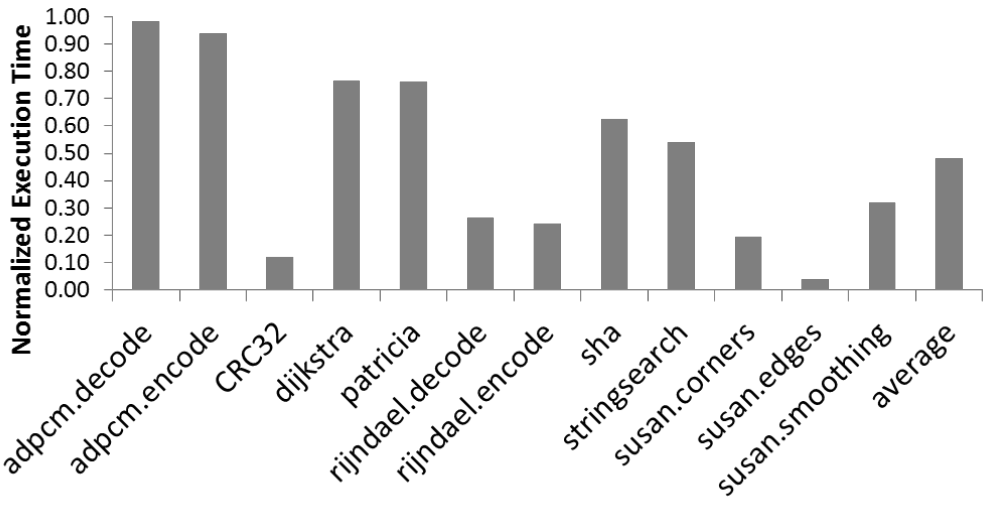


Figure 3.8: Execution time of our approach normalized to the previous pointer management. Our pointer management reduces the execution time by 52% on average.

Gem5 simulator. The start up cost is set to 91 nanoseconds (about 291 CPU cycles), and the data transfer rate is set to 0.075 nanoseconds (0.24 CPU cycles) per byte of data Kistler *et al.* (2006) (4 bytes/cycle). These numbers are consistent with the parameters used in Lu *et al.* (2013).

Table 3.1 shows the number of pointer management function calls introduced by the state-of-the-art pointer management and our approach (the first three columns under *Previous Pointer Management* and *Our Pointer Management* categories respectively). The numbers show that our approach almost completely eliminates calls to the pointer management functions, i.e. *l2g, g2l* and *ptr_wr*. For example, for *rijndael.encode*, the numbers of calls of *l2g*, *g2l*, and *ptr_wr* are reduced from 155940, 1442301, 28 to 1, 1, and 0 respectively. These results are for experiments on SMM architecture with the SPM size equal to the average of the minimum and maximum stack size for each application. We will explain our choice of the SPM size later.

The reduction of pointer management consequently reduces the execution time of

applications. Figure 3.8 shows the normalized execution time of benchmarks using our approach over the previous approach. Our approach achieves on average 52% reduction of execution time. Notice that even for the benchmarks in which we do not achieve significant performance improvement, for instance, *adpcm.decode*, and *adpcm.encode* our pointer management still reduces the pointer management overhead. We get less improvement because the time spent on pointer management is insignificant compared to the execution time in these applications.

3.6.2 Comparable Performance Compared to Caches

On top of the comparison with the state-of-the-art stack management techniques for SMM architectures, we also compare the performance of our technique with hardware caching. We perform a conservative comparison with cache-based architectures. The cache-based system is configured to have a 4-way L1 data cache which only caches the stack data. All the other memory accesses are considered as cache hits. The size of the cache is configured to be the smallest power of two greater than the SPM size. Also, we set the cache miss penalty to be the same as the DMA start-up cost. The overhead in a cache-based architecture is equal to the number of cache misses times the cache miss penalty. Meanwhile, the overhead of stack management in a SPM-based architecture includes both the time for executing the extra management instructions, and the time for DMA operations to move data. For each application, the SPM size is the average of the minimum and maximum stack size of the application (again the reason will be explained later).

Table 3.1 shows the stack management overhead caused by our approach on an SMM architecture (the fourth to sixth columns under *Our Pointer Management* category) versus that caused by hardware caching. The DMA transfers for benchmarks without pointer management are triggered by stack frame management. When the

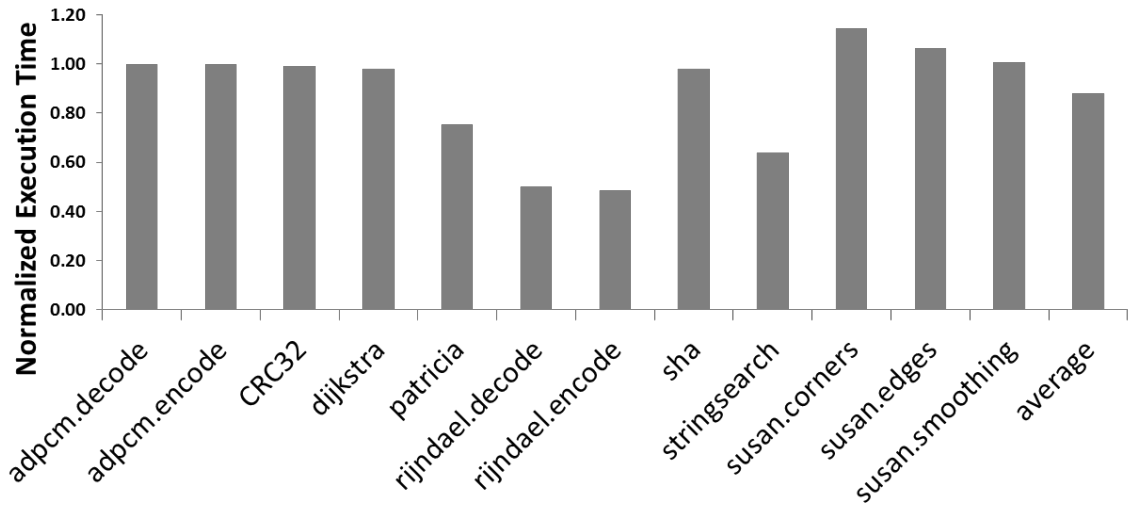


Figure 3.9: Normalized execution time of our approach on an SMM architecture normalized to caching.. Our approach achieves 12% reduction of execution time on average, even with conservative estimates.

benchmark *rijndael.encode* is executed on the SMM architecture, our approach requires 87 management instructions and 2 DMA calls which transfers 2336 bytes of data. When this benchmark is executed on a cache-based system with the stack data being managed on a cache slightly larger than the SPM, it incurs 244983 misses. Therefore, even with the extra instructions, SPM management is still more efficient. Figure 3.9 plots the execution time of a benchmarks on the SMM architecture, normalized to the execution time of the same application running on the cache-based system. The plot shows that our approach reduces the execution time by 12% on average.

3.6.3 Choice of SPM Stack Size

Figure 3.10 shows the execution time of the benchmarks with three SPM sizes: the minimum required stack size (size of the largest stack frame in the application),

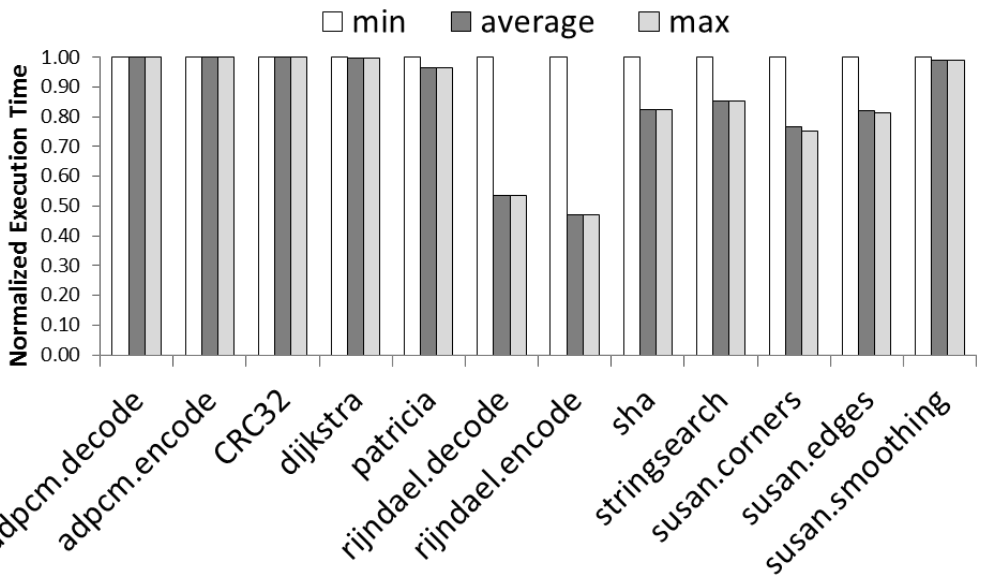


Figure 3.10: Execution time of our approach using three different SPM sizes, all normalized to the execution time with the minimum SPM size.

the maximum possible stack size (sum of the sizes of all the stack frames), and their average. All of them are normalized to the execution time when using the minimum size. As the figure shows, while using the minimum size may cause longer execution, using the average size or using the maximum size are not much different.

This is because in all the benchmarks we used, the size of the largest stack frame is much larger than the others, so even if we only use the average SPM size (greater than the size of the largest stack frame), it is large enough to hold multiple small stack frames, which is able to eliminate pointer management for calls between these functions with small stack frames. However, if we only allocate the minimum required size (exactly equal to the size of the largest sack frame), and the function with largest stack frame happens to be called in the loop, there will be no other choices but to evict stack frames within the loop every time the function is called, which causes much higher overhead, such as *rijndael.encode* and *rijndael.decode*. Therefore, the

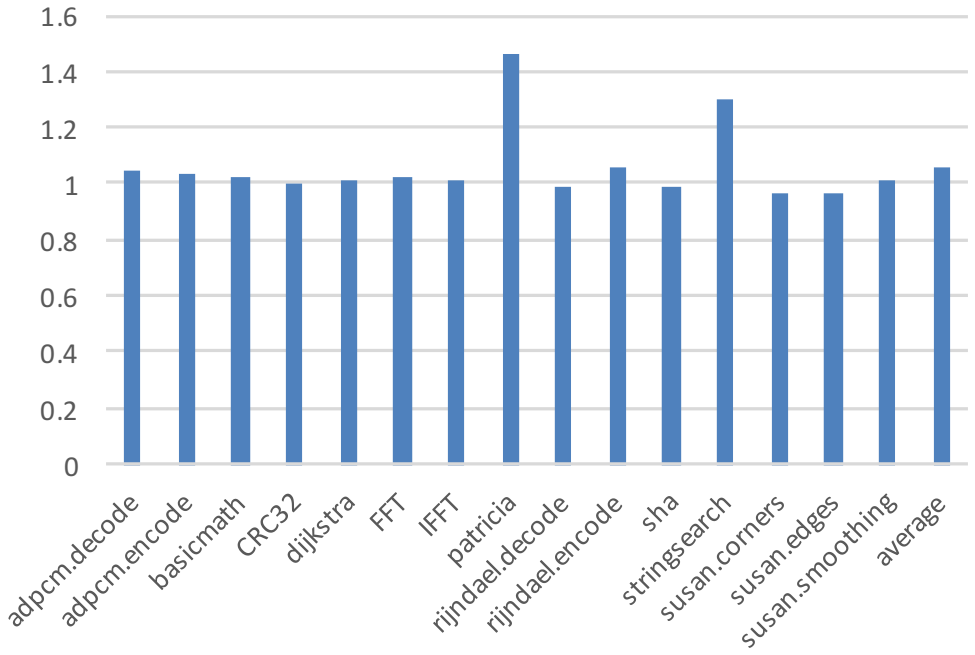


Figure 3.11: Execution time of integrated code and stack management, normalized to the execution time of caching.

average size is chosen to balance the execution time and SPM space used.

3.6.4 Integrated Management

We integrate the stack management technique with code management technique from Chapter 2, and compare the performance with caches. The caches-based architecture has an L1 instruction and L1 data cache. For each application, the SPM size is the sum of i) the average of the minimum code size and maximum code size, and ii) the average of minimum and maximum stack size of the application. The L1 instruction cache size is set to the smallest power of 2 equal to or greater than the average of i), and the L1 data cache size is set to the smallest power of 2 equal to or greater than the average of ii). Figure 3.11 shows the execution time of benchmarks on the SMM architecture, normalized to the execution time of the same application

running on the cache-based system. The combined execution time is6% slower on average. This is because while our code management outperforms cache, the stack management was worse than caches [1] , causing the integrated performance slightly worse than caches.

3.7 Conclusion

In this chapter we presented an approach of pointer management on stack data for Software Managed Multicore (SMM) architectures. Our approach divides function calls of a program into groups based on the call graph and inserts pointer management functions only if a pointer to stack data is defined and used in two different groups. The experimental results demonstrate that our approach not only significantly improves overall performance compared to the state-of-the-art pointer management in stack management, but also delivers comparable performance over using the cache for stack data management.

[1]While the experiments in Section 3.6.2 includes stack data of library calls, the integrated management technique only manages user-defined stack data in order to be consistent with the code management technique introduced in Chapter 2, which only manages user-defined code. Therefore, the results are different when compared to caches

Chapter 4

OPTIMIZING HEAP DATA MANAGEMENT ON SOFTWARE MANAGED MANYCORE ARCHITECTURES

4.1 Introduction

Caches significantly reduces memory access latency in today's processors. However, they consume significant amount of silicon area and energy Niar *et al.* (2004), and the cost of maintaining cache coherence increases rapidly with the number of cores Bournoutian and Orailoglu (2011); Choi *et al.* (2011); Garcia-Guirado *et al.* (2011); Xu *et al.* (2011). For these reasons, Processor vendors have opted to remove caches and use only Scratchpad Memories (SPMs) Gschwind *et al.* (2006b), or reconfigurable SRAMs that can be configured as SPMs Texas Instrument (2014). An SPM is raw memory that stores only data, without the complex circuitry in a cache to implement automatic movement of data between the lower-level and upper-level memories, replacement policies and coherence. As a result, SPMs consume about 40% less area and energy per access Banakar *et al.* (2002).

To use SPMs, however, data movements in and out of the SPM must be managed explicitly by software. For this reason, we refer to such an SPM-only manycore architecture as Software Managed Manycore (SMM) architecture. SMM processors have been used for high performance computing Carter *et al.* (2013); REX Computing, Inc. (2014), gaming and multimedia processing Gschwind *et al.* (2006b), digital signal processing Texas Instrument (2014), and networking Olofsson (2016).

Different management techniques for each type of data is required. This is because different types of data each has its own characteristics. For example, stack frames of

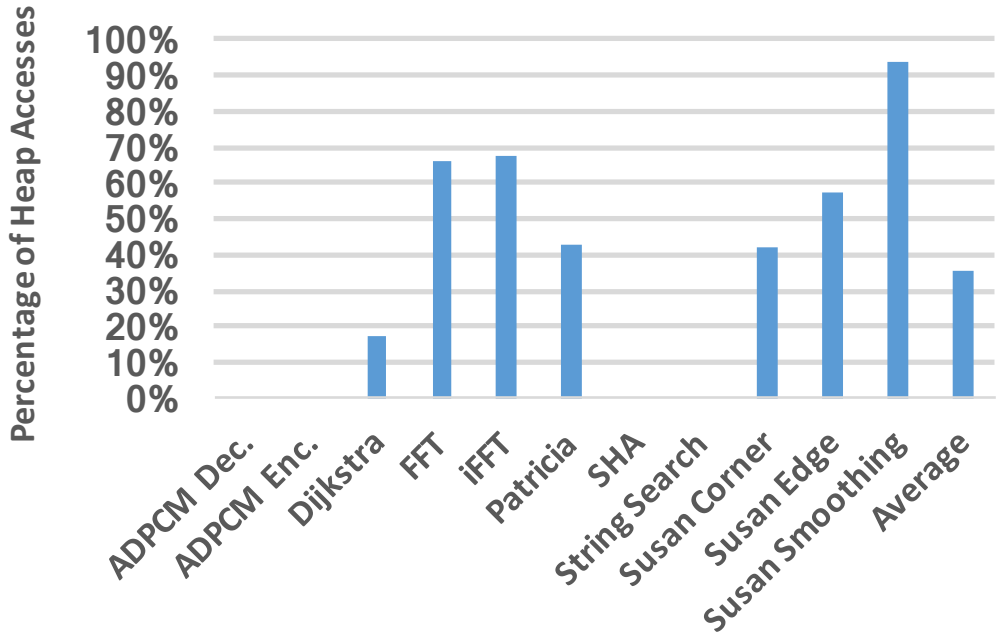


Figure 4.1: Percent of heap accesses among all the accesses (excluding code accesses).

adjacent function calls are stored contiguously in memory, therefore we can manage stack frames at groups. On the other hand, heap accesses are more unpredictable, as heap objects typically scattered in memory, even for those accessed within the same function. Therefore, typically the space of an SPM is separated into different division, with one division dedicated to the management of one type of data. In this paper, we focus only on heap data management, assuming code, global and stack data have been properly managed. Heap management is very important to application performance, since heap accesses may account for a significant portion of overall memory accesses. Figure 4.1 shows the percentage of heap accesses out of all the data accesses in MiBench benchmarks. While heap accesses may not be present in all the programs, it is dominant in some, with more than 90% in `Susan Smoothing`.

By default, all heap data are accessed via memory addresses (that run correctly on cache-based processors). However, when we bring the heap data to the SPM, it must

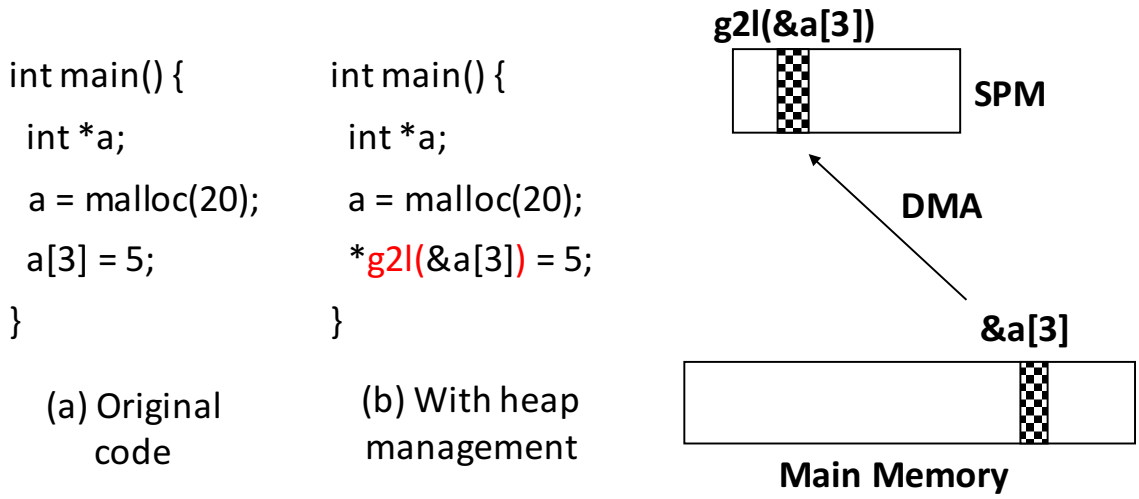


Figure 4.2: The heap management function g2l takes an memory address as input and maps it to a location in SPM. If the data in the memory location is not already present in the SPM location, g2l function will also issue a DMA to copy the data.

be accessed using the SPM address , which is different from the main memory address (unlike caches). Figure 4.2 (a) shows that in the original code, `malloc` function is called to allocate a heap object in the main memory. The accesses to the heap object are directly to the main memory. For an SPM-based processor, we first fetch the data into SPM before accesses. This is achieved by calling a heap management function, that we refer as *g2l* in the rest of the paper. In Figure 4.2 (b), the function *g2l* first brings the data into SPM if it is not present in the SPM yet.

The implementation of the *g2l* function in the state-of-the-art dynamic heap management techniques on SMM architectures Bai and Shrivastava (2013) enables the correct execution of programs with heap data on an SPM-based processor, but causing high overhead. It is due to i) many *g2l* calls are not necessary, and ii) the implementation of *g2l* introduces high instruction overhead.

To solve the problems, we propose optimization techniques i) reduce management

overhead by not calling $g2l$ when absolutely not needed, ii) simplify complexity and reduces the instruction overhead of $g2l$, iii) inline and remove redundant operations of $g2l$ calls. We implement the proposed techniques on LLVM Lattner and Adve (2004), and evaluate them on Gem5 CPU simulator Binkert *et al.* (2011a). The benchmarks used are from Mibench suite Guthaus *et al.* (2001a). Experimental results show that compared to the state of the art, our techniques our approaches reduce execution time by 80% on average.

4.2 Related Work

Heap management on SPM can be generally divided into static approaches, quasi-static approaches and dynamic approaches. Static approaches treat an SPM as a heap, and implement efficient memory allocator to manage heap data on the SPM Wilson *et al.* (1995); McIlroy *et al.* (2008) to avoid run-time overhead at every memory access. However, such methods may be forced to allocate heap objects to main memory when there are not enough space on SPM. Quasi-static approaches divide execution into time intervals, and bring the most frequently used data into the SPM at the beginning of each interval Dominguez *et al.* (2005). Within the interval, locations of heap data are fixed (thus the name), either in SPM or in the main memory. Such approaches usually rely on profiling, which becomes inaccurate when representative input is absent.

Dynamic approaches change the set of data objects and the location of each object in SPM as program executes, and are the most flexible. Previous dynamic approaches for heap management, including ours, are mostly based on software caching, due to the high dynamism of heap data. They are however different from ours. For example, Moritz et al. implemented a compiler-based software cache on a raw SRAM Moritz *et al.* (2001). The technique relies on fast translation, which requires on additional

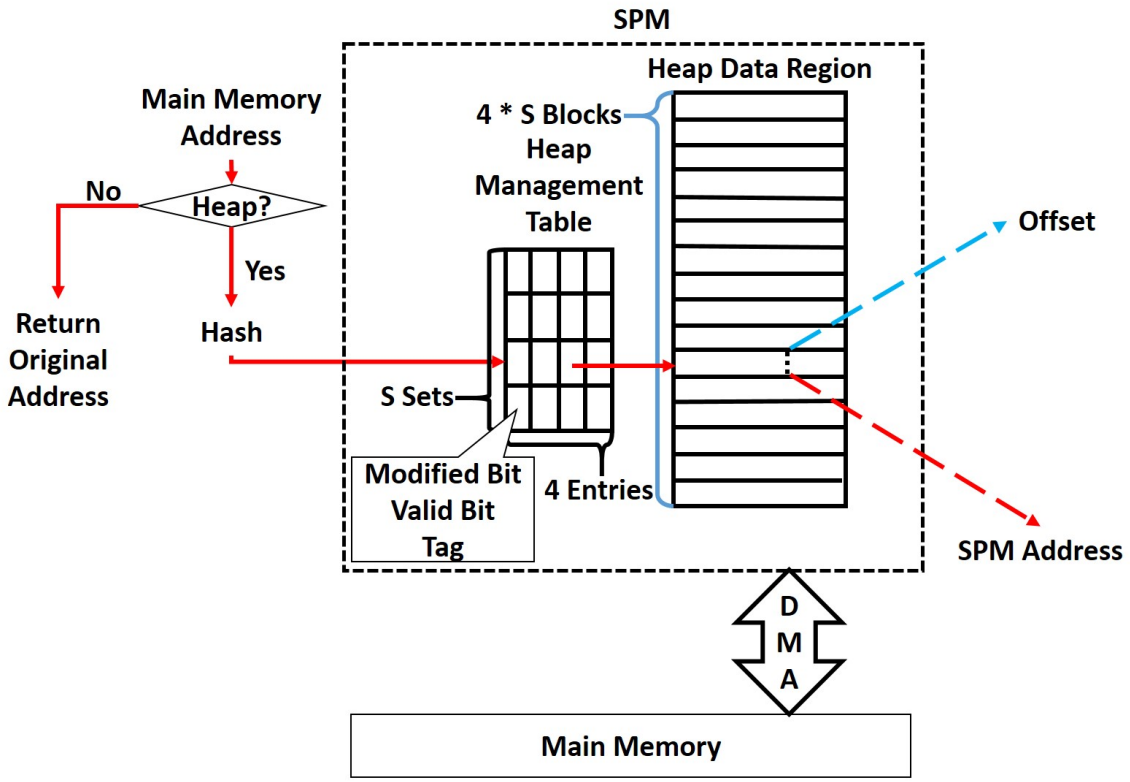


Figure 4.3: The state-of-the-art heap management implements a 4-way set-associative software cache on SPM.

registers, while our technique does not require additional hardware. Chakraborty et al. proposed an array management technique on SPM Chakraborty and Panda (2012) that labels arrays into either statically allocated in SPM or dynamically managed via software cache. The decision relies on the knowledge of array sizes, while our technique does not require array sizes, and can be used to manage any data aggregates, such as linked lists or trees.

The state of the art dynamic heap management for SMM architectures from Bai et al. Bai and Shrivastava (2013) is the most related work to our technique. The state of the art implemented a 4-way set-associative software cache with first-in-first-out (FIFO) replacement policy on SPM. The details of the technique will be explained

shortly. However, it introduces very high management overhead. Our technique can significantly reduce overhead and improve performance.

4.3 Limitation of the State of The Art

The state of the art heap management Bai and Shrivastava (2013) emulates a 4-way set-associative cache on an SPM. The SPM is partitioned into a data region and a heap management table (HMT), as shown in Figure 4.3. The data region stores the actual heap data in fixed-sized blocks, while the management table stores a tags, a modified bit, and a valid bit for each block in the data region, i.e. there is a one-to-one mapping between each block in the data region and each entry in the management table. Every 4 entries in the management table forms a set, with a victim index for round-robin replacement policy.

The *g2l* function in the state of the art takes a main memory address as input, and checks if the given address is in heap. The input address is immediately returned if it is not in heap. Otherwise, the set index of the input address is calculated. A sequential search is done to compare the tag of the input address with the tags of the four entries in the set. The data block at target heap address is brought to SPM if it is not present, before the SPM address is returned.

Although the state of the art correctly manages the heap data of an application, it incurs high performance overhead. Figure 4.4 shows its management overhead on some typical embedded applications. It is important to note that the heap management technique not only significantly increases the execution time of applications, but also inflicts high overhead on the benchmarks without any heap accesses, `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search`.

The high overhead is caused by two main reasons.

i) heap management function *g2l* called before each memory access. Since

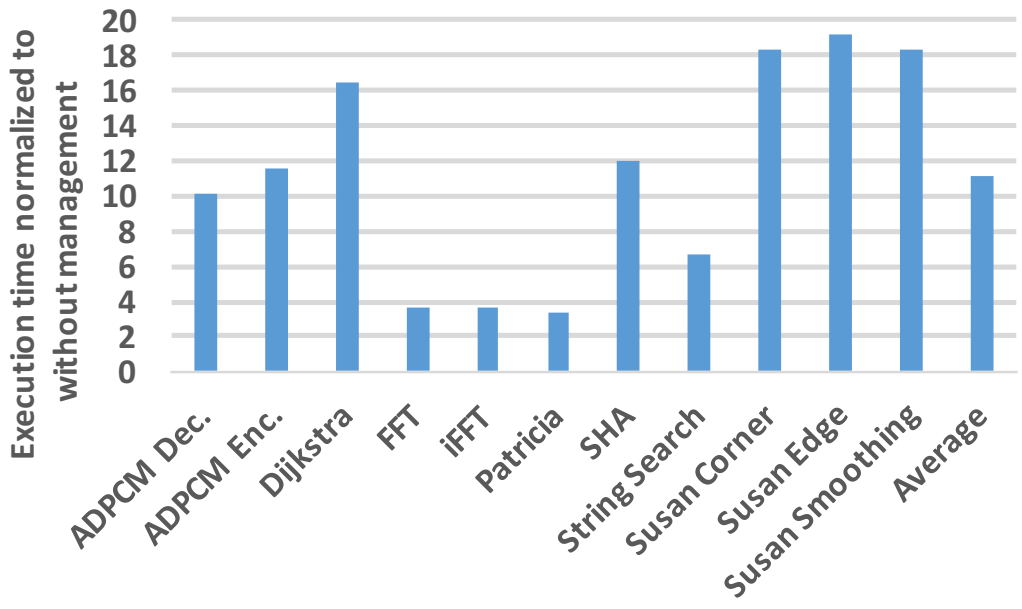


Figure 4.4: Performance overhead caused by the state-of-the-art heap management approach.

the management functions check if an access is to heap at runtime, it has to be called before each memory access (including those are not to heap). The checking is expensive, not only because it happens at every memory access, but also because it involves branch operations, and potentially memory operations.

ii) set-associate heap management. The second major source of overhead comes from the fact the previous technique manages heap data in a set-associative manner. The software implementation of the set-associate structure has to sequentially search all the entries in the set at every heap access. It also complicates the calculation of the set index, and the translation of a main memory address to the corresponding SPM address. The set index of the input main memory address is calculated with the following hash function:

$set\_index = ((mem\_addr >> log(block\_size)) \wedge (mem\_addr >> (log(block\_size) +$

$1)))\&(set\_num - 1)$

where mem_addr is the input main memory address, $block\_size$ is the size of a data block, and set_num is the number of sets. The SPM address is then calculated as:

$spm\_addr = spm\_base + (set\_index * set\_assoc + entry\_index) * block\_size + mem\_addr \% block\_size$

where spm_base is the start address of the data region, set_assoc is the set associativity (4 in this case), and entry_index is the index of the entry in the set specified by set_index. The complexity of the calculations translates to significant instruction overhead.

4.4 Key Ideas of Our Approach

To solve the problems in state of the art, we use a series of optimizations that can greatly reduce the overhead of heap management on SMM architectures:

i) statically detecting heap accesses. This optimization identifies heap access at compile-time and eliminates heap management function $g2l$ when the memory is definitely not a heap accesses, and significantly reduces the number of (unnecessary) management calls at runtime. It also eliminates the runtime checking within the management function.

ii) simplifying management framework. We implemented a direct-mapped cache on SPM. In a direct-mapped cache, it is no longer required to sequentially go through different entries and search for the requested data block for each heap access. In addition, it simplifies the calculation of set index and the SPM address in the management functions. Therefore, this optimization can effectively reduce the number of instructions in each management function.

iii) inlining and combining management calls. Once $g2l$ functions are inserted, we inline the function calls. We also remove the (redundant) heap management functions and execute them once before all the management calls. This optimization is

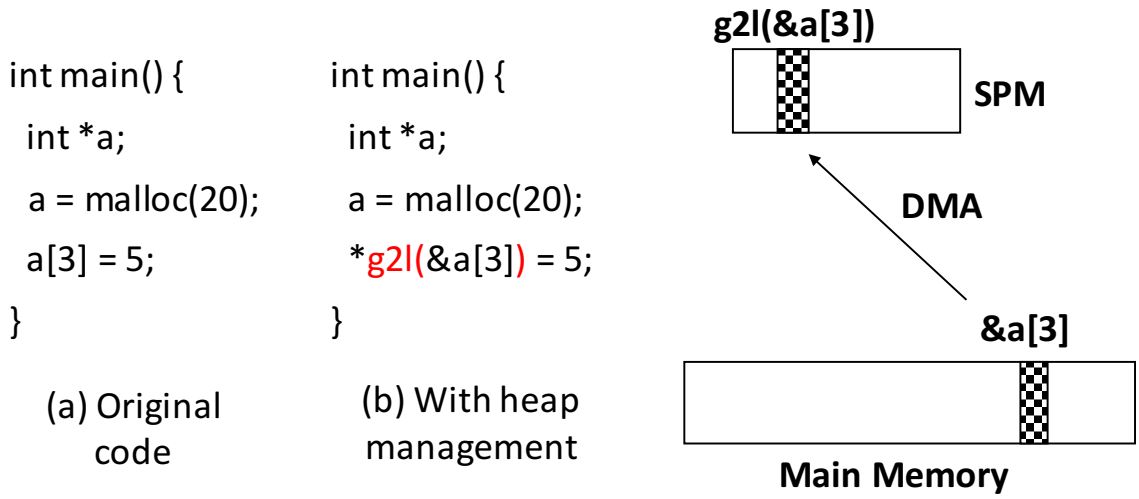


Figure 4.5: The previous approach inserts g2l before every memory access, while our approach tries to identify the heap accesses statically and skip unnecessary g2ls.

particularly beneficial, when management functions are called within loops, and the common operations are hoisted to be outside of the loops.

4.5 Details of Our Approach

4.5.1 Statically detect heap accesses

This optimization identifies heap accesses at compile-time, so that the management function *g2l* can be avoided at memory accesses that are absolutely not to heap. Figure 4.5 illustrates the effect of this optimization. The original program defines a structure, which consists of two integer pointers `a` and `b`. It then creates a global variable `s` as an instance of the structure, and assigns `s->a` with an heap object created by a call to the `malloc` function. The program then points `s->b` to the fourth integer element starting from the address in `s->a`. Later `s->b` is used to access the heap object. The program also defines a pointer `p` that refers to a stack variable.

Even though only `s->a` and `s->b` points to heap data in this program, the previous heap management technique Bai and Shrivastava (2013) will insert a *g2l* call at every memory access unnecessarily as in Figure 4.5(b), including memory accesses via `p` and `s` (not `s->a` or `s->b`), which are to stack and global data respectively. On the other hand, with static detection of heap accesses, we only insert *g2l* before the memory instructions via these two pointers.

To find out heap accesses, we first identify all the the heap pointers. Algorithm 3 explains the method we use to identify heap pointers, which includes both the pointers that directly points to heap objects created by memory allocators (e.g. *malloc* or *calloc*), and their aliases. The analysis starts at `getHeapPtr`. In this procedure, the analysis first executes `getAlloc` procedure, taking as input the *main* function (line 2). The `getAlloc` procedure identifies all the invocation of memory allocators in the input function `F`, and record the pointers that are used to store the created heap objects (line 8 and 9). If `F` calls any other functions `F'`, `getAlloc` recursively accesses and identifies the memory allocations in `F'` (line 11 and 12). Once all the heap pointers that stores the heap objects created by memory allocations are identified, the analysis continues to identify all the possible alias of these heap pointers by executing the `getAlias` procedure on the *main* function (line 4). The `getAlias` procedure goes through each instruction in the input function `F`, and recognizes any instruction that performs pointer arithmetic on a heap pointer and assigns the result to another pointer. The destination pointer of such an instruction is identified as an alias of the heap pointer. Similar to the `getAlloc` procedure, in case `F` calls any other function `F'`, the `getAlias` procedure recursively calls itself on `F'` to identify aliases created in `F'`. Since each iteration of the `getAlias` procedure may recognize new aliases, this procedure is repeated until no new aliases can be recognize (line 3 to 5).

Once all the heap pointers are recognized, we can identify heap accesses and insert

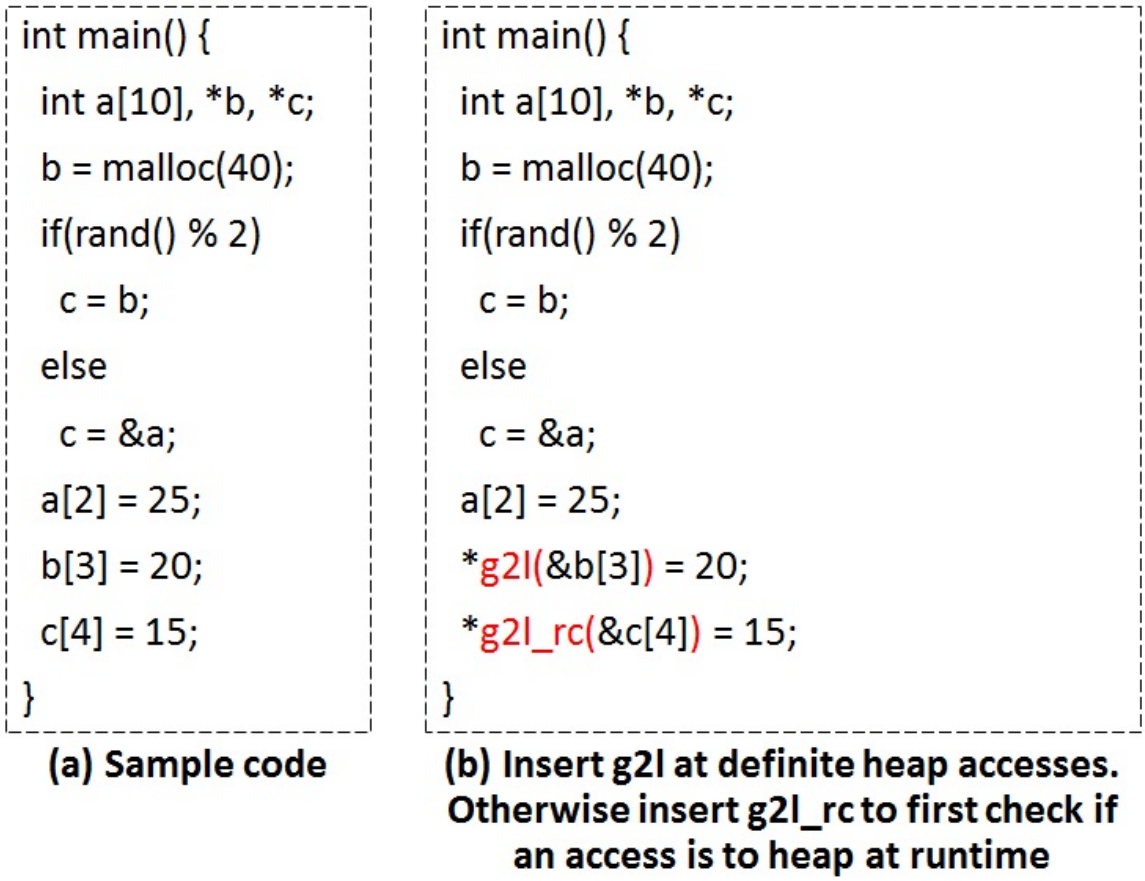


Figure 4.6: When a memory access may be to heap but is not for certain, we check at runtime before managing the access.

g2l function as follow. All the memory access (i.e. loads ans stores) via any of the heap pointers identified in Algorithm 3 are considered as potential heap accesses. An *g2l* function is inserted right before the memory instructions to first translate the memory address to an SPM address. The SPM address is then used to substitute for the original memory address in the instructions.

There are cases when the compiler cannot determine whether a pointer refers to heap data. In Figure 4.6(a), the pointer `c` can either refer to heap data or stack data, depending on the outcome of the call to `rand` function, which returns a random number. We introduce a new management function, called *g2l*`_rc`, that checks at

runtime and see whether the memory address is in heap, similar to the previous work. When we are sure an access is to heap, we call the *g2l* function, which does not have any runtime checking. If an access may be to heap, we call *g2l*_rc. Otherwise, if we are sure an access is definitely not to heap, we do not call any heap management functions. Figure 4.6(b) shows the transformed code with heap management function. We call *g2l* before accessing the data referred by the pointer `b`, because we are sure it is in heap. We call *g2l*_rc before accessing `c`, because it may refer to heap data, but are not for sure. We do not insert any heap management function when accessing `a`, because it is definitely in stack.

4.5.2 Simplify management framework

Whenever a memory access happens, a software-cache based approach has to first calculate the set index of the memory address. The software cache will then sequentially access the entries in the set and compare the tag of the target address with the tags in the entries. Once the data block that contains the target address is located, either already in the SPM in a hit, or first copied from the main memory in a miss, the final SPM address is generated and used to replace the original memory address in the memory access.

Since this process happens within each management function call, it is performance critical to speed up this process. With a direct-mapped cache on software, this process can be noticeably simplified to execute much less instructions at runtime, compared to a set-associative cache. Figure 4.7(a) and Figure 4.7(b) shows two examples using the previous approach and our approach respectively. The edge in both figures specify dependence between two steps. The previous approach as in Figure 4.7(a) calculate set index with the following function:

$set\_index = ((mem\_addr >> log(block\_size)) \wedge (mem\_addr >> (log(block\_size) +$

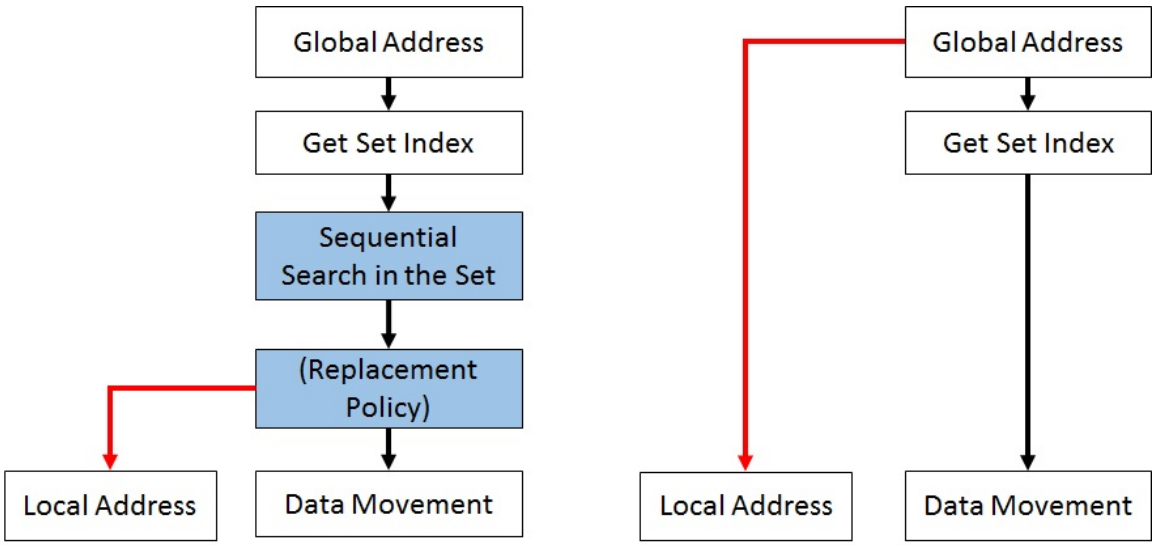


Figure 4.7: (a) The steps of a management in the previous work (b) The steps of a management function in our approach The steps of a management function in the previous work and our approach.

$1)))\&(set\_num - 1)$,

where mem_addr is the input main memory address, $block\_size$ is the size of a data block, and set_num is the number of sets. The software cache then searches the corresponding set for the requested data block. Only after the data block is found (either after a hit or after a miss), can then the SPM address be generated as

$spm\_addr = spm\_base + (set\_index * set\_assoc + entry\_index) * block\_size + mem\_addr \% block\_size$,

where spm_base is the start address of the data region, set_assoc is the set associativity (4 in this case), and entry_index is the index of the entry in the set specified by set_index. Notice this equations required both the index of the set and the index of the entry in the set, which explains the dependence of the calculation of the SPM address on the sequential searching in Figure 4.7(a). On the other hand, our approach

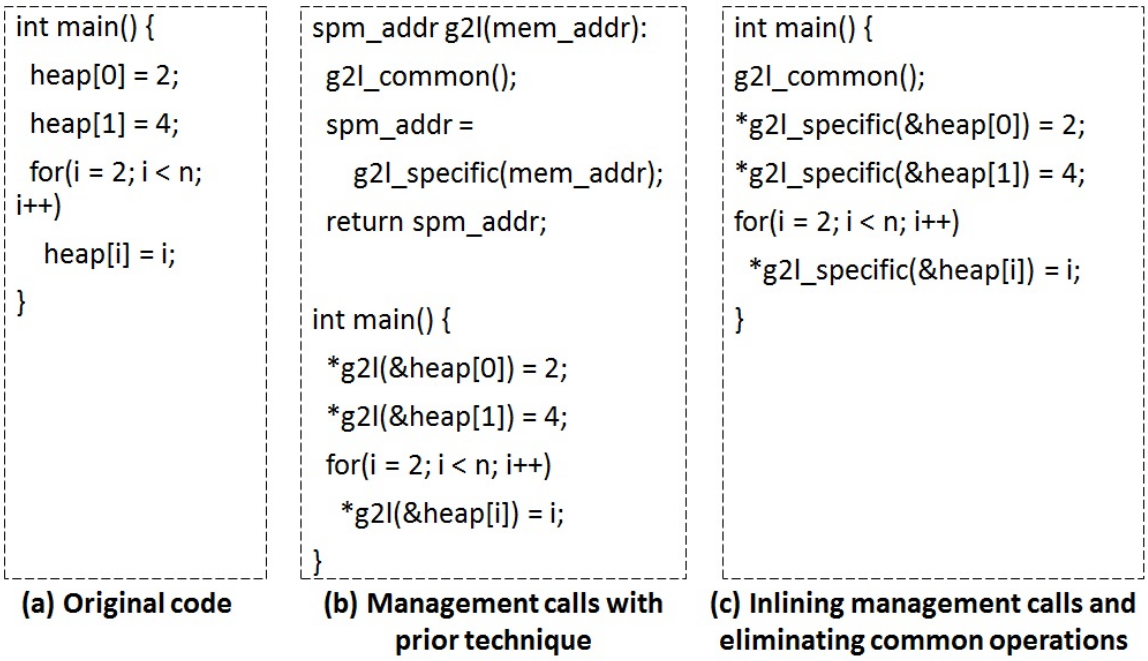


Figure 4.8: We inline management calls and move common operations to the beginning of the caller function.

in Figure 4.7(b) simplifies the calculation of the set index of a memory address into $set\_index = global\_addr >> log(block\_size)\%set\_num$,

Since each set has only one entry, sequential searching is not necessary. The software can simply go ahead and calculate the final SPM address as $spm\_addr = spm\_base + mem\_addr\%(set\_num * block\_size)$.

In addition, the calculation of SPM does not depend on any previous steps. Elimination of such dependence may allow the compiler to have more parallelism when generating and scheduling the machine instructions for the management functions.

4.5.3 Inline and combine management calls

Once *g2l* function is inserted after identifying heap accesses statically, we can reduce the management overhead by inlining the management functions, which enables

further optimization. In Section 5.1.2 we explained the previous approach divided SPM into two memory regions for heap management table and data region. Our approach makes similar usage of SPM space. Every $g2l$ thus has to load the start address of the heap management table and data region at the beginning of its execution, before executing any other call-specific instructions. Therefore, we can move these common instructions outside of the $g2l$ function and execute it once before any $g2l$ calls, so that all the subsequent $g2l$ calls can reuse the results, similar to common subexpression elimination.

Figure 4.8 shows an example. Figure 4.8(a) lists the original code. Figure 4.8(b) is the transformed code before inlining. Each $g2l$ call first executes the common instructions redundantly, and then execute specific instructions for that call. We represent the common instructions and specific instructions in a $g2l$ with function calls $g2l\_common$ and $g2l\_specific$ respective in the example, but they are plain instructions in the actual implementation. In Figure 4.8(c), we inline the $g2l$ calls, move and execute the common instructions at the beginning of the caller function. After the optimization, only call-specific instructions are executed at where a $g2l$ was called. While this optimization generally will improve performance, its importance is maximized when $g2l$ was originally called within loops, as this example shows —instead of repeatedly and excessively executing the common steps in a loop nest, moving these common instructions to be outside can significantly reduce such overhead.

The algorithm of this optimization is shown in Algorithm 4. The compiler goes through every function in the program and inlines $g2l$ calls with call-specific instructions. The common instructions are moved to the beginning of the function.

4.6 Experiments

4.6.1 Experimental setup

We implemented both the state-of-the-art technique Bai and Shrivastava (2013) and our technique as intermediate representation (IR) passes on LLVM 3.8 Lattner and Adve (2004) respectively. We then compiled the same benchmarks with different heap management techniques, ran the executable code on Gem5 Binkert *et al.* (2011a) and compared the performance. The block size in the software cache is set to 64 bytes in both techniques.

We emulated the SMM architecture on Gem5, by modifying the linker script and

Benchmark	Heap Size (KB)
Adpcm Decode	0
Adpcm Encode	0
Dijkstra	6.43
FFT	32
iFFT	32
Patricia	766
SHA	0
String Search	0
Susan Corner	92.16
Susan Edge	42.81
Susan Smoothing	17.35
Typeset	32

Table 4.1: Maximum heap usage of benchmarks

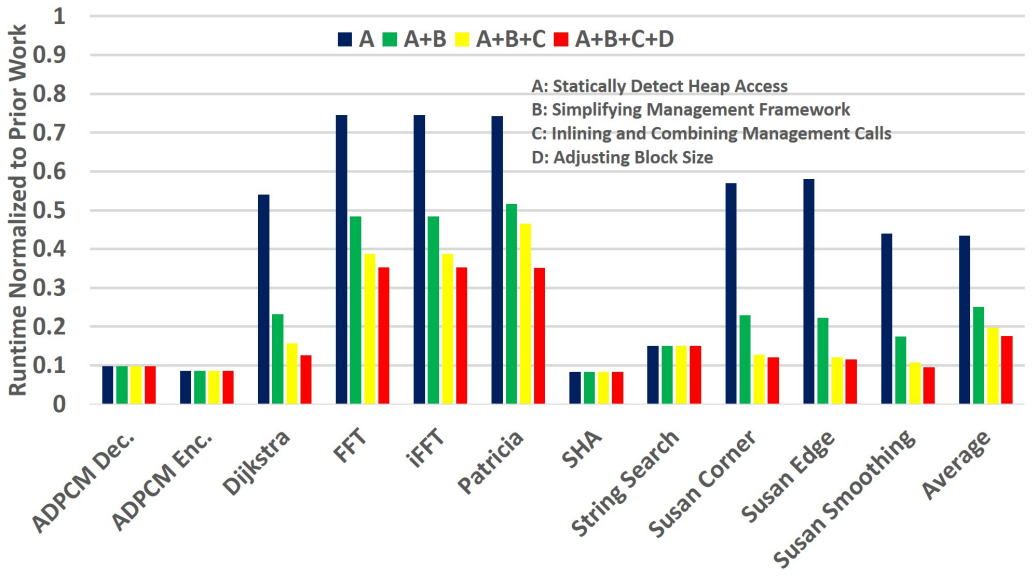


Figure 4.9: The execution time of our approach normalized to the previous work with optimizations incrementally added.

reserving part of the memory address space as the SPM. We implemented an DMA instruction that copies data between the SPM and the main memory. DMA cost is modeled as a constant startup time and the time for actual data movement. The startup time is set to 291 cycles, and the rate for transferring data is set to 0.24 cycles/byte. The CPU frequency is set to 3.2 GHz. All these parameters are based on the IBM cell processor Kistler *et al.* (2006).

We evaluated the proposed technique on benchmarks from Mibench benchmark suite Guthaus *et al.* (2001a). Table 4.1 lists the maximum usage of heap size in the benchmarks, i.e. the maximum sum of sizes of heap objects at any moment. The benchmarks that have zero heap usage do not have any heap accesses.

Benchmark	Before	After
Adpcm Decode	116702082	0
Adpcm Encode	10211280	0
Dijkstra	149209166	19077784
FFT	336608	90188
iFFT	336671	90204
Patricia	3114668	893184
SHA	8350153	0
String Search	2198090	0
Susan Corner	1238553	273717
Susan Edge	2628207	579221
Susan Smoothing	37252034	4891730
Typeset	274118	3826

Table 4.2: Number of g2l calls called before and after identifying heap access statically with the previous technique

4.6.2 Significantly reduces execution time

Figure 4.9 shows the execution time of our approach normalized to the previous work, as optimizations incrementally introduced. Overall, our approach reduces execution time by 80% on average.

We can clearly see from the result in Figure 4.9 that statically detecting heap accesses contributes the largest reduction of execution time, especially in benchmarks that do not have any heap accesses, i.e. `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search`. Overall, statically detecting heap access reduces the execution time by 57% on average. This is because of two reasons: reduced management calls and

Case	Previous Work	Statically Detecting Heap Accesses	Simplifying Management Framework	Inlining and Combining Management Calls
read hit	52	46	19	8
write hit	59	53	23	10
read miss w/o write-back	145	139	41	36
write miss w/o write-back	145	139	44	37
read miss w. write-back	172	166	58	45
write miss w. write-back	172	166	58	45

Table 4.3: Instructions executed per g2l under different cases with optimizations incrementally added.

less executed instructions in each call. Table 4.2 shows the number of calls to the *g2l* function before and after statically detecting heap accesses in the previous work. The management calls is significantly reduced in all the benchmarks. For example, the number of management calls is reduced from 2628207 to 579221 in `Susan Edge`. In the benchmarks that do not have any access, management calls are completed eliminated.

Statically detecting heap accesses also allows us to eliminate runtime checking at *g2l*s, and thus reduces number of instructions. Table 4.3 shows the average number of instructions each *g2l* executes under different cases, after we incrementally introduce

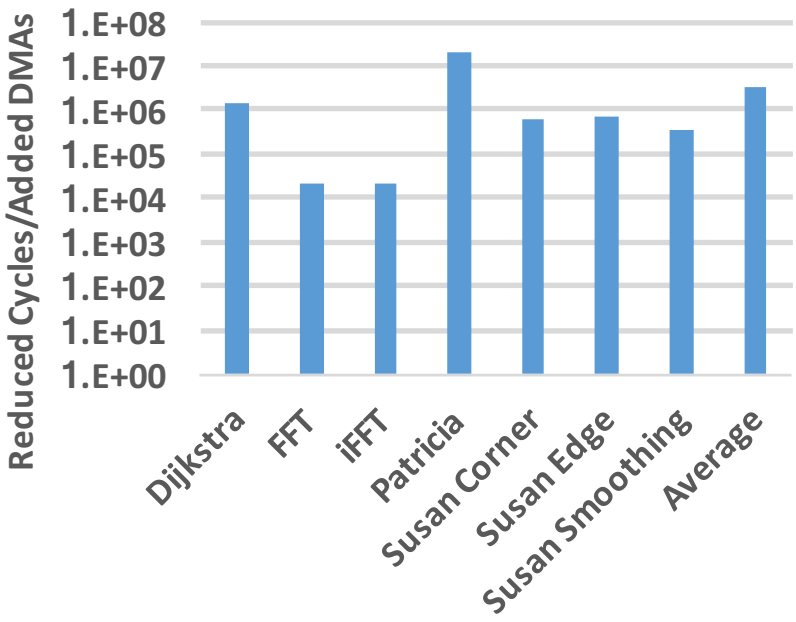


Figure 4.10: Implementing a direct-mapped cache other than a 4-way set-associative cache reduces more execution time than the extra time introduced due to increased cache misses.

the optimizations. There are 3 possible cases when a *g2l* function is called: a cache hit; a cache miss and a clean data block is evicted; a cache miss and a dirty data block is evicted. The memory access may either be a read access or a write access, so there are 6 different cases overall that may happen when calling a *g2l* function. The table shows there is a constant difference of 6 instructions between the Previous Work column and the Statically Detecting Heap Accesses column in any case.

Simplifying management framework, by implementing a direct-mapped software cache instead of a 4-way set-associative cache, reduces execution time by 42% on average (on top of statically detecting heap accesses). This is because average dynamic instruction count of *g2l* calls in all the cases of Table 4.3 is significant reduced. For example, the average instructions executed in the sixth case is reduced from 166 to 58

after simplifying management framework. Since a direct-mapped cache causes more cache misses compared to a 4-way set-associative cache, we also compare the benefit (reduced cycles) due to less management instructions to the penalty (increased cycles) due to increases cache misses. Figure 4.10 shows the reduced CPU cycles thanks to less management instructions normalized to the increased CPU cycles because of more cache misses. The simplification of management framework improves the performance of a benchmark, as long as the quotient of that benchmark is greater than 1. For example, in `Patricia`, the reduced cycles are more than 10000000 times than the increased cycles. The figure shows that the increased cycles almost are ignorable compared to the reduced cycles, in all the benchmarks.

Inlining and combing management calls can further reduce execution time by 21% (on top of statically detecting heap accesses and simplifying management framework), thanks to the removed function calls and redundant operations. For example, as Table 4.3 shows, the average instructions executed in the sixth case is reduced from 166 to 58 after simplifying management framework, and is further reduced from 58 to 45 after inlining and combing management calls. Notice we apply this optimization after statically detecting heap accesses. So if the heap management calls are all eliminated after that step, inlining and combining management calls will not improve performance. For example, the management calls of `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search` are reduced to 0 after the compiler statically finds out there are no heap accesses in these benchmarks. The performance is therefore not further improved after the first optimization.

4.6.3 Scales well with SPM size

In the above experiments, the SPM size is set to 4KB. Figure 4.11 shows the execution time before and after applying the optimizations to the previous technique,

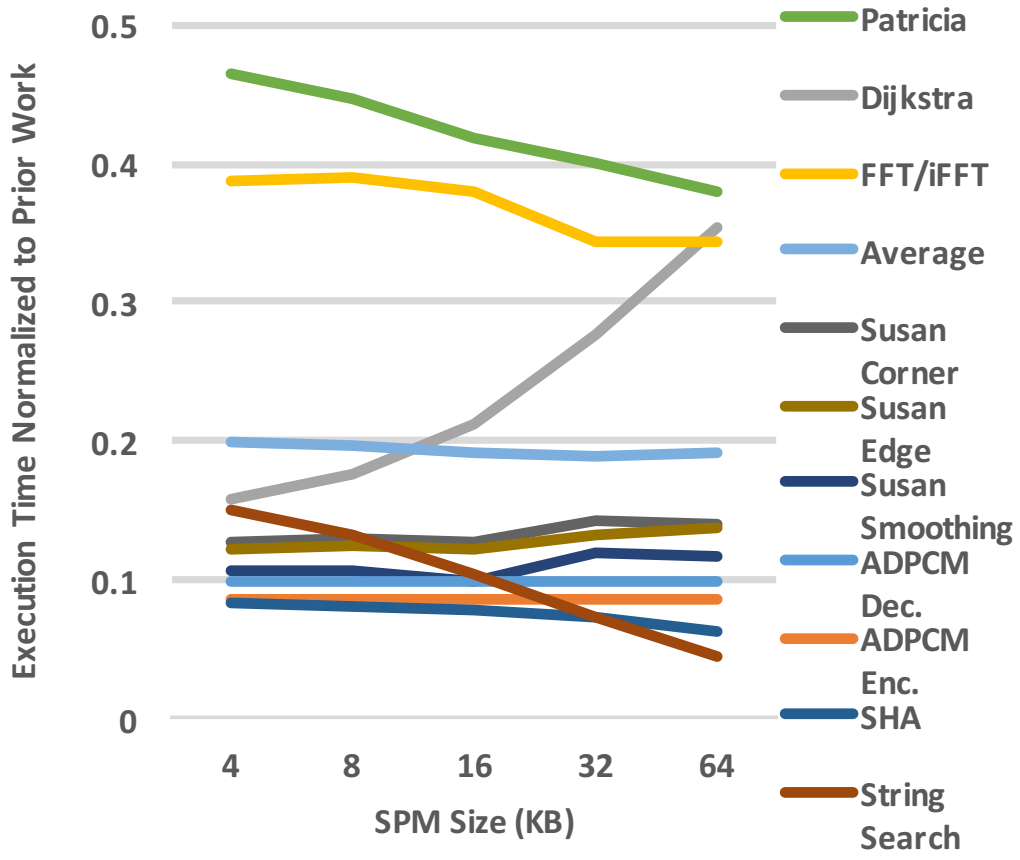


Figure 4.11: Execution time of our approach normalized to the previous work, when the SPM size increases from 4KB to 64KB. The order of the names on the right side is the same as the order of endpoint of the line for the corresponding benchmark.

as the SPM size increases from 4KB to 64KB.

Under any SPM size, our technique achieves significant improvement over the previous work. In addition, as the SPM size increases, the normalized execution time of most benchmarks decreases. The only exception is `Dijkstra`. The reason is because in our experiments, we assume the source code of library functions is not available. Therefore, we can not insert $g2l$ function in the code of library functions. Notice this is a common problem for most if not all the compiler based approaches. As a result, all the modified data blocks in the software cache must be flushed to the

main memory whenever a library function may modified heap data, to conservatively ensure the correctness of execution. In `Dijkstra`, `malloc` and `free` functions are extensively called. These two functions modify heap data, therefore we have to flush the software cache every time either of the functions is called. While both the previous technique and our technique suffer from the overhead, it is worse for our technique. The proof is briefly given as follows. When we flush a software cache, we have to check all the blocks in the cache and write back dirty data. Since the capacity and block size of the cache are the same for both techniques, the number of blocks to check are the same in both techniques. The overheads of flushing software cache are therefore roughly the same in both techniques. Let the time for flushing the software cache be T_{flush} in both techniques. Let the time for the rest of execution be T_1 with the previous technique, and T_2 with our technique. The normalized execution time can be calculated as $(T_{flush} + T_1)/(T_{flush} + T_2)$. As the cache size increases (the data block size remains), T_{flush} becomes higher, since the number of data blocks to check increases. Therefore, the normalized execution time becomes larger. For the similar reason, we can have the source code of library functions, we can eliminate the flushing in both techniques and the normalized execution time will be further reduced.

4.7 Conclusion

Due to the expense of caches, some processor designers have opted to use SPM as an alternative. SPM-only software manged manycore (SMM) processors have been widely used in various area. However, the data management must be explicitly done on SPM, which introduces overhead. In this paper, we propose an efficient management technique for heap. The experimental results show that with the optimizations, we can reduce the execution time by 80% on average compared to the state of the art.

Algorithm 3 Identify heap pointers

```
1: function getHeapPtr
2:     getAlloc(main)
3:     repeat
4:         getAlias(main)
5:     until cannot find new aliases
6: function getAlloc(Function F)
7:     for each instruction I in F do
8:         if I is a call to any memory allocator then
9:             Record destination pointer P as a heap pointer
10:        else
11:            if I is a call to any user function F' then
12:                getAlloc(F')
13: function getAlias(Function F)
14:    for each instruction I in F do
15:        if I is an assignment statement and one of the operands P is a heap pointer
   then
16:            Record destination pointer P' as an alias of P
17:        else
18:            if I is a call to any user function F' then
19:                getAlias(F')
```

Algorithm 4 Inline and combine g2l calls

```
1: function inlineManagementFunction(Function F)
2:     for each function F do
3:         if F has any call to g2l then
4:             insert common operations of g2l at the beginning of F
5:             for each g2l call I in F do
6:                 inline the call
7:                 remove the common operations
```

Chapter 5

SHARED DATA MANAGEMENT

5.1 Introduction

So far, we have introduced management of the local SPM for each core. If tasks running on different cores do not interact with each other, then the presented work is able to run them correctly. However, in the presence of shared data for inter-task/inter-core communication, we need to make sure they are coherence among cores, in order to get expected results.

The challenges of coherence management of SMM architectures are extremely analogous to the coherence management of non-coherent cache (NCC) architectures. NCC architectures are cached-based multicore architectures yet without cache coherence. For example the 8-core TI TMS320C6678 processor features a non-coherent cache memory architecture, or in the more general purpose domain, the 48-core Intel SCC Howard *et al.* (2011) has non-coherent cache memory architecture. NCC architecture require explicit memory transfers to synchronize shared data that are modified and/or accessed by different cores when running parallel programs, just liken any SMM architecure. This problem is illustrated in Figure 5.1. Without explicit communication, modifications of shared data are not propagated properly and cause unexpected problems. The only difference between an SMM architecture and an NCC architecture is that in the NCC architecture, code and data are automatically managed between local caches and main memory, which could be taken care by the management of local SPMs presented earlier. We therefore demonstrate our work in an NCC architecture, which allows us to focus on the coherence management

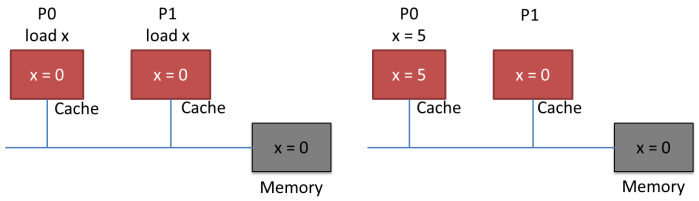


Figure 5.1: Coherence of data has to be managed explicitly on a Non-coherent Cache (NCC) multi-core architecture. Suppose physical memory location x initially contains value 0 and both cores P0 and P1 caches it. If P0 writes a new value 5 at address x, P1 will still stay uninformed and access the stale value.

problem. More importantly, the coherence management work in NCC architectures can be easily exported to SMM architectures.

A lot of research has been done on coherence management of shared data Kontothanassis and Scott (1995); Stenstrom (1990); Reinhardt *et al.* (1994). Most previous work maintains coherence of shared data at fixed granularity, either at block level Scales *et al.* (1996) or page level Carter *et al.* (1991); Bershad *et al.* (1993). Typical approaches either rely on hardware such as the page fault hander of Memory Management Unit (MMU) to identify writes to shared pages and invalidate its copies in all other cores Stenstrom (1990); Carter *et al.* (1991); Bershad *et al.* (1993), or develop software to manage the modifications on each core and do compute-intensive comparisons between the original and the modified copy to apply the change Lee *et al.* (2008). The hardware approaches are not considered in this proposal since modern multi-cores are usually designed without complex hardware to save power. But even with the software approach, a fixed granularity approach may not be the best option for NCC architectures—such approaches usually incur much computation power. For example, when multiple cores write to the same shared page, each writing core needs to create and modify its local copy, and compare the modified

copy to the original copy in order to apply changes to the original copy, to prevent its modifications from overwritting changes done by other cores Lee *et al.* (2008). In traditional multi-processor systems with computationally strong processors and relatively slow inter-process communications, increasing computation to avoid (more expensive) communication makes sense. However, in modern multi-core systems, each core is designed with relatively weak computational power to preserve power efficiency, with relatively fast communication speed. For example, the theoretical peak performance of an accelerator core on the Cell processor is around 20 GFLOPS IBM Technical Library (2005), while that of a conventional Xeon processor reaches over 600 GFLOPS AMD (2012). Therefore, we need to a different solution to adapt to the changes on processor design.

In this chapter, we present a pure software approach at byte granularity on NCC architectures to manage coherence of shared data among cores while reducing the computational overhead caused by the management. Our approach achieves more than 2X performance improvement compared to the state-of-art approach Lee *et al.* (2008) on an 8-core non-cache coherent TI TMS320C6678 processor Texas Instruments (2017). Experiments also show that the performance overhead of managing coherence via our approach is more lower.

5.1.1 Related Work

Maintaining coherence of shared data has been a heavily studied topic for multi-processor systems, e.g., cluster computing, grid computing, and distributed computing, in an effort to provide a Distributed Shared Memory (DSM) to applications Nitzberg and Lo (1991). A DSM manages data coherence between processors and creates an illusion of shared memory over a distributed memory system on multi-processor systems. A compendium of many important approaches to provide

coherence among the processors of a multi-processor system can be found in Tartalja (1997). DSMs are closely related to software cache coherence as both try to provide a single image of memory to all processors. Most software DSMs use page-grain coherence management Protic *et al.* (1995). If a page-grain DSM is implemented by modifying page-fault handler, it is also called Software Virtual Memory, or SVM. All these coarse-grain approaches aim to reduce the communication between processors, even if it results in a slight increase in the computation required.

Although page-grain coherence management reduces communication, it is prone to false sharing. To avoid the adverse effects of false sharing on the performance, some researches propose to manage coherence at variable granularity. Carter et al. Carter *et al.* (1991) introduced a method of managing coherence at size of the data items, through user-defined association between synchronization objects and data items. Their approach relies on Memory Management Unit (MMU) trapping page faults, Scales et al. Scales *et al.* (1996) transparently rewrite the application executable to intercept loads and stores in compiler. Sandhu et. al. Sandhu *et al.* (1993) introduced a program-level abstraction called shared region, and users explicitly call a function that binds a shared region to a set of memory locations with variable sizes, and access shared regions via some provided functions which guarantee the synchronization of different processors. Bershad et. al. Bershad *et al.* (1993) also provides variable granularity coherence management by asking users to explicitly associate data items and synchronization objects.

More recently, providing coherence and consistency in multi-cores has become a more important research topic. Several hardware based schemes have been proposed to assist in coherence implementation, and make it more efficient. A recent proposal from Zhao et al. Zhao *et al.* (2013) proposes two adaptive cache coherence schemes. The first scheme supports adaptive granularity of storage but fixed cache line size,

while the other supports both adaptive granularity of storage and cache line size. Ophelders et al. Ophelders *et al.* (2009) proposed a hardware-software hybrid scheme which places private data in write-back cacheable memory regions and shared data in write-through cacheable memory regions. While these approaches attempt fine-grain coherence management, they are implemented by introducing new hardware, while we are seeking for software based solutions.

Among the software approaches, Kim et. al. Kim *et al.* (2011) proposed a software shared virtual memory for the Intel SCCHoward *et al.* (2011) without cache coherence. Their approach requires modification of page-fault handler and manages coherence at page granularity. However, several NCC multi-core architectures including the TI TMS320C6678 do not have a MMU for each core, therefore page-fault handler based schemes are not applicable for these embedded NCC architectures. The work most closely related to our work is COMIC Lee *et al.* (2008). It is a software approach designed for multi-core processors without hardware cache coherence Gschwind *et al.* (2006a) and MMU in each core. In the experiments section, we will compare the performance of COMIC with our approach. We describe COMIC in more detail in the next section.

5.1.2 Previous Approach

COMICLee *et al.* (2008) proposes a software approach which implements Release Consistency model Gharachorloo *et al.* (1990) at page granularity. Release Consistency consists of two special operations: *acquire* and *release*. The program execution between acquire and release is called *interval*. All memory accesses after an acquire operation should be performed only after the acquire operation has been performed while all memory accesses before a release operation must have been performed by the time when the release operation is performed. Acquire and release operation are

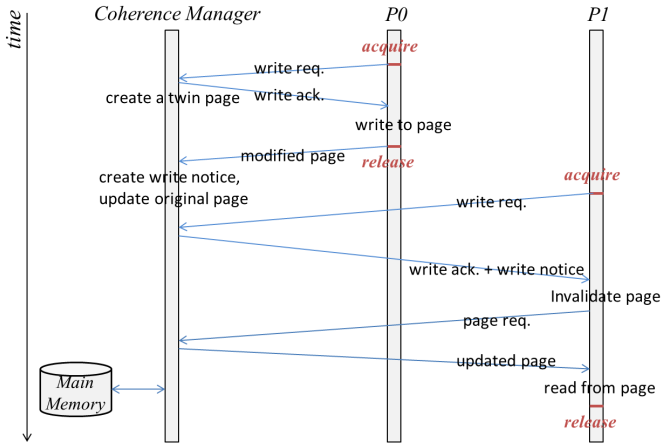


Figure 5.2: The way COMIC works.

performed in program order.

Figure 5.2 illustrates how COMIC works. It requires one core to act as coherence manager, which is the only processing element that can directly access the main memory. All the other cores have to make memory requests via coherence manager. In addition, whenever write requests to a shared page arrive, the coherence manager creates a copy of the page (termed *twin*), and sends the page to all the requesting cores. When the cores are done, they send back the modified pages, and coherence manager then finds out the changes of each core by comparing the unmodified copy to the modified pages. Only the changes are apllied to the original page in the main memory.

5.1.3 Our Approach

Our approach implements Release Consistency model at byte granularity. Figure 5.3 shows an overview of our approach. Whenever a core wants to modified shared data, it first performs an acquire operation. Upon its success, the core creates a private duplicate in main memory and all the subsequent writes go to the duplicate.

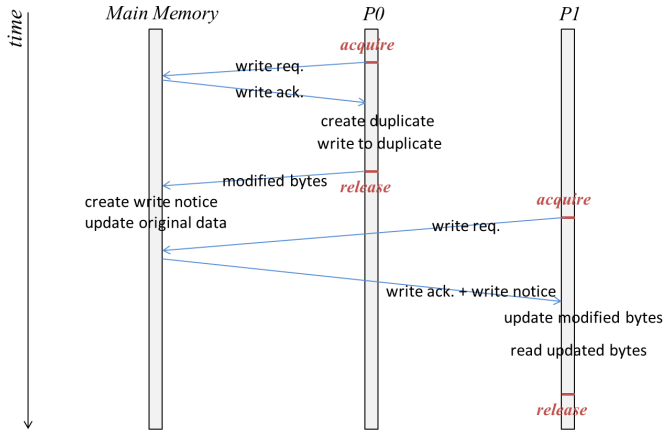


Figure 5.3: The way our approach works.

All writes during the interval are recorded in write notices. Each write notice is a record of the the starting memory location and the number of bytes modified. On the release operation, the core makes all the write notices visible to other cores by pushing them back to the main memory, together with the modified values. The subsequent acquiring core can then read the write notices from the main memory and either update or invalidate its local copy of modified data.

Code Transformation

We implement our own synchronization primitives. Figure 5.4 shows an example of how the code will be changed with our management functions. The *_lock* performs an *acquire* operation once the exclusiveness to the critical region is obtained, while the *_unlock* function performs a *release* operation and releases the lock. The *write_notice_add* function records the start address and size of the modification.

Fine-Grain Coherence via Write-Notice

Write notice is the key component in implementing release consistency model, and the content of a write notice determines coherence granularity. For example, in traditional

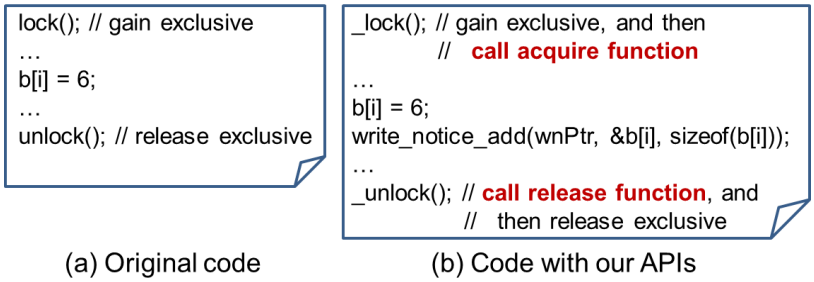


Figure 5.4: Code transformation with our management functions. Figure (a) shows the snippet of original code. Figure (b) shows the transformed code.

page-grain coherence scheme, e.g., COMIC, a write notice will mark which page is updated but not where exactly the page is modified. So whenever any coherence operation (e.g., acquire and release) happens, it either invalidates or updates the entire page, which usually causes unnecessary data transfer. Our coherence scheme works at byte granularity, assuming that no more than one processing elements should access different bits on the same byte simultaneously. In our approach, a write notice records the beginning address and the exact size of the memory locations. By doing so, a core only needs to write back the exact modified part of shared data at the release operation. For example, if a core modified only one word in a cache block, it will write only that word instead of flushing the whole cache line.

Reduced Computation Overhead

Compared to communication pattern of COMIC (Figure 5.2), our approach (Figure 5.3) writes back the exact modified bytes to main memory instead of pages. As a result, when the modified data is written back to main memory, no comparison is needed to figure out the modification. Also, the duplicate and write notices are created by each acquiring core, but not by the coherence manager as in COMIC. By doing so, we avoid compute-intensive tasks of creating twins, comparing different

copies of the same page and applying the difference. Moreover, it also distributes the load to multiple cores and mitigates the throttle of a centralized manager.

Further Optimizations

To further reduce the runtime overhead of our approach, we propose two more ways of optimization. The first optimization aims to reduce the contention to memory locations if multiple cores are trying to create write notices at the same time, while the second one tries to reduce the number of write notices.

Private Write Notices Two types of write notices are used in our design - private write notices and global write notices. Private write notices record modifications by a specific core during an interval, while global write notices keep records of all modifications from all cores. When a core works on an interval, it works on its private write notices. Private write notices will be merged with global write notices at the release operation. Global write notices can be accessed by any core. During acquire operation, modifications done by other cores can thereby be propagated to the subsequent core. [1] With private write notice, we can avoid creating performance bottleneck caused by centralization.

Merging Write Notices Less number of write notices should reduce the number of inquiries and DMA transfers on acquire operation, while a core goes through all the write notices and apply modifications on shared data. Therefore, merging and reducing number of write notices will reduce its overhead and improve performance. To do so, before creating a write notice, we first check if the range of new write completely or partly overlaps with or is adjacent to any existing write notices. If so,

[1] Unless noted, write notices mentioned in the rest of this chapter refers to the global write notices.

we change the existing notice of interest to include the new write instead of creating a new write notice. In particular, at release stage, we also merge private write notices with the existing global write notices.

5.1.4 Experimental Results

Experimental Setup

Our experiment platform is TI TMS320C6678 evaluation board Texas Instruments (2017). The board has a single C6678 processor and a 2GB DDR3 memory. It is based on TI's KeyStone multi-core architecture, and has eight cores on chip, each of which can run at up to 1.25 GHz. Cores are connected by on-chip teraNet with a bandwidth of 2 Tbps. Each core has its private L1 cache and shares an L2 cache.

We took several commonly-used routines in many numerical or multimedia applications. The benchmarks and their input sizes are described in the first two columns

Table 5.1: Benchmarks

Benchmark	**Input Size**	**numIters**
Compress	512x512	2048
Laplace	512x512	2048
Lowpass	512x512	2048
Wavelet	4096x4096	8192
MMT	M=4, N=16384, P=64	-
MV	M=4, N=16384, P=1	-
MM	M=4, N=16384, P=64	-
MT	M=512, N=512	8192

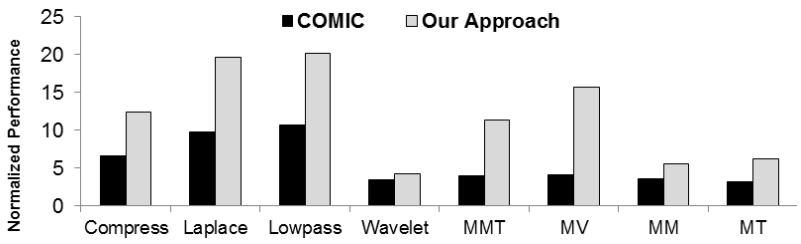


Figure 5.5: Comparison of the performance of our approach and COMIC.

of Table 5.1. MM, MV, and MMT, are matrix-matrix, matrix-vector, and matrix-transposed matrix multiplication, respectively. MT stands for matrix-transpose. All benchmarks are implemented as multi-threaded programs. We divide the computation equally by dividing the output array into equal sub-arrays and making each core be responsible to compute on one of the sub-array, and put a barrier at the end to ensure all cores have finished the computation on its own portion before the final result can be provided as an output.

Better Performance than COMIC

Our metric of performance is execution time. The shorter the execution time the better the performance. In this experiment, we compare the performance of our coherence scheme and COMIC. The numbers are normalized to a baseline approach that disables cache. As shown in Figure 5.5, we achieve over 2X performance improvement compared to the COMIC for all benchmarks.

To better understand the individual performance results of the benchmarks, we analyzed the access patterns of benchmarks and categorized their temporal locality and spatial locality as *strong* or *weak*. A benchmark shows strong spatial locality, if more than one contiguous memory locations are accessed in each iteration of the innermost loop. Otherwise the benchmark has weak spatial locality, if it only accesses discontinuous memory locations. Similarly, a benchmark shows strong temporal lo-

cality if any memory location is accessed more than once over time. Otherwise the benchmark shows weak temporal locality, if it accesses memory locations only once. *Compress*, *Laplace* and *Lowpass* show both strong temporal and spatial locality, since to compute an element in the target array, the program needs to access some surrounding neighbors in the source array, e.g., to calculate b[i][j], the program needs to access a[i-1][j-1], a[i-1][j], a[i-1][j+1], a[i][j-1], a[i][j], a[i][j+1], a[i+1][j-1], a[i+1][j], a[i+1][j+1], and over the time a[i][j] needs to be accessed for the calculation of b[i-1][j-1], b[i-1][j], b[i-1][j+1], b[i][j-1], b[i][j], b[i][j+1], b[i+1][j-1], b[i+1][j], b[i+1][j+1]. *Wavelet* shows strong temporal locality. However, it writes to two discontinuous locations in each iteration, which impairs the spatial locality. Both *MV* and *MMT* access a large region of contiguous memory for many times in a loop, showing both relatively strong temporal and spatial locality. Although MMT provides the same functionality as MM (multiplying two matrices), it has stronger spatial locality. This is because MM accesses matrix in column-wise order, while the data for the matrix is laid out in memory in row-major order (assume arrays are stored in row-major order). As a result, MM does not show as much performance improvement as MMT does. *MT* shows limited locality of reference. Array elements are accessed in a column-wise manner in the transposed matrix, and each element in both the original and transposed matrix is accessed only once and never again.

Note that the performance of our approach over MMT, MV, MM and MT vary significantly, while the performance of COMIC does not show much variation. This is because for every shared data access, COMIC has to check a dirty bit to find out if the page has been invalidated, whereas our approach does not. As a result, the performance gained from locality of references is compromised in COMIC because of the extra memory accesses introduced may poison the cache.

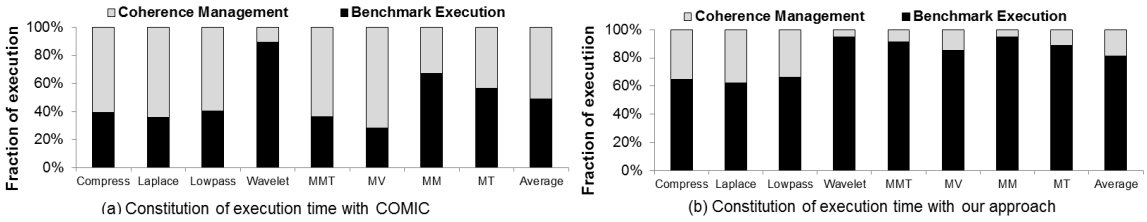


Figure 5.6: The comparison of runtime overhead of our approach and COMIC.

Reduced Runtime Overhead than COMIC

We show other factors that could affect performance in this section. First, note that COMIC dedicates one core to coherence management, so only seven of all the cores were used for actually executing threads. Note that this limitation of being able to execute only one thread on a core is quite a common limitation in several real multi-core architectures, e.g., the IBM Cell processor Gschwind *et al.* (2006a), the 48-core Intel SCC Howard *et al.* (2011), and the TI Keystone architecture Texas Instruments (2017)—our experimental platform. This is to minimize the overhead of operating system on the cores and to allow extremely power-efficient bare-metal execution. Using less number of threads affects the performance of COMIC. However, that is not the only reason for the worse performance of COMIC. Figure 5.6 shows the fractions of coherence management and the actual execution of benchmarks in total execution time for both approaches. Overhead comprises all the actions related to coherence management, e.g., comparison of different copies of the same page. As shown in the figure, COMIC takes up to 51% of overall execution time, while our approach takes only 19% on average. By using our write-notice based approach, we successfully avoid expensive twin-page comparison, and eliminate the checking of dirty-bits of pages for every shared data access.

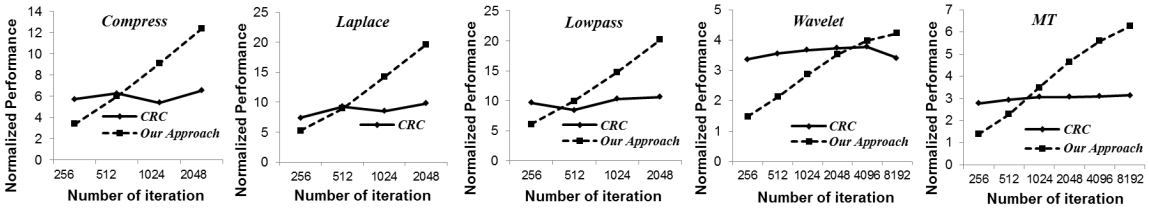


Figure 5.7: Compute-intensity is varied by changing the numIters parameter.

More Scalable Overheads with Increasing Computation-to-Communication Ratio

We compare the scalability of our approach to COMIC as the inter-core computation-to-communication ratio increases. To show that, we increase the workload assigned to each core before the synchronization by increasing loop counters in this experiment. By repeating the same work before the barrier, the workload of each core is increased before it needs to communicate with other cores at the barrier, while the amount of time for inter-core communication at the barrier remains the same, since each core modifies the same shared memory locations and thus only needs to propagate the same amount of information to subsequent accessing cores to these locations. Therefore, it reduces the time spent on inter-core communication in the overall running time. The third column of Table 5.1, i.e., *numIters*, shows the number of iterations the workload will be repeated. The bigger $numIters$ is, the lower the weight of inter-core communication in the overall running time. Figure 5.7 shows the impact of reducing the weight of inter-core communication on the two different approaches. The Y axis shows the performance normalized to the baseline, which disables caches. The figure clearly shows that benchmarks using our approach gets near-linear performance improvement as the number of iterations increases, while their performance suffers from the increasing overhead of COMIC. The overhead of COMIC increases because dirty-bit checking has to be done for every shared data access. Although that bit

checking can be done by a cheap modular operation, it accumulates to a large overhead as the number of iterations increases.

Another important observation is that, when the iteration count is small, the overhead of our coherence management stands out more, and our approach performs worse than COMIC. This is because our approach uses write-update protocol. Under such protocol, each core needs to fetch more data than they actually need to. When there is very little computation in each interval (for example, when each thread mostly updates some elements of array with some constants without any calculation), this drawback will become more prominent. However, the overhead of our coherence management gets amortized off as the amount of workload increases, which should be acceptable as the communication should not be dominant most of the time.

Chapter 6

MY CONTRIBUTIONS

The code management technique in Chapter 2 is based on the published paper *Reducing Code Management Overhead in Software-Managed Multicores* Cai *et al.* (2017). The paper was published in Proceedings of the 2017 International Conference on Design Automation and Test in Europe (DATE), 2017. I conceived the idea of eliminating unnecessary code management functions when function-to-region is given, and implemented the technique.

The stack management in Chapter 3 is based on the published paper *Efficient Pointer Management of Stack Data for Software Managed Multicores* Cai and Shrivastava (2016). The paper was published in Proceedings of the International Conference on Application Specific Systems, Architectures and Processors (ASAP), 2016. I was one of the major contributors to the idea of removing unnecessary pointer management functions in stack management, and implemented the technique .

The heap management technique in Chapter 4 is based on a paper we recently finished. I coauthored the paper and was a major contributor to the idea implemented in the paper. The paper currently for review at the time of writing.

The shared data management technique in Chapter 5 is based on a paper textit-Software Coherence Management on Non-Coherent Cache Multi-cores Cai and Shrivastava (2016) published in International Conference on VLSI Design (VLSID), 2016. I conceived the idea of managing shared data in fine granularity to reduce computation overhead, and implemented the technique.

Chapter 7

SUMMARY

The swiftly increased overhead on area and power of cache coherence makes multi-/many-core system difficult to scale. While we can keep improving the efficiency of cache coherence mechanism, some processor designs have sought to use scratchpad memories instead of caches in memory hierarchy to curb the cost. The scratchpad memory (SPM) is a low-cost SRAM memory —it removes the hardware logics (such as tag array and tag comparators) used in caches to automate the data transfers, and therefore uses 34% less area and consumes 40% less power than a cache of the same capacity Banakar *et al.* (2002). As a result, data management on SPMs is explicitly managed on software, typically automated in compilers to save programmers from manually programming the data transfers. For this reason, SPM-only architectures are also referred as software managed multi-/many-core (SMM) architectures.

SMM architectures consume much less power and area compared to traditional cache-based many architectures (thanks to the using of SPMs), and therefore is easier to scale as number of cores increases. However, efficient software SPM management techniques are still required for applications to deliver high performance on SMM architectures. It would be otherwise meaningless if SMM processors only deliver poor performance, not matter how much power it may have saved. In other words, SMM processors may overcome the scaling issue of multi-/many-core architectures, only if they can deliver high performance with low power consumption. Therefore, it is extremely important to produce software that solves the problem both correctly and —more importantly—efficiently.

This thesis develops different automatic compiler-based SPM management tech-

niques to manage code, stack and heap on SMM architectures. These techniques insert data transfer instructions automatically in programs at compile-time, after proper analysis. They improve performance by 14%, 52%, and 80% on average compared to the state of the art techniques, respectively. The SPM code and stack management techniques can even outperform caches by 9% and 12 % on average respectively. On top of the local data management, a technique for shared data management among different cores in an SMM processor is also developed, achieving more than 2X speedup compared to its predecessor. Overall, this thesis provides an complete solution to manage both the local and shared data on SMM architectures efficiently.

REFERENCES

AMD, "HPC Processor Comparison", URL `http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/ESG-AMD-HPC-Processor-Comparison-Aug-2.pdf` (2012).

Angiolini, F., F. Menichelli, A. Ferrero, L. Benini and M. Olivieri, "A Post-compiler Approach to Scratchpad Mapping of Code", in "Proc. of CASES", (2004).

ARM, "ARM1176JZF-S Technical Reference Manual", http://infocenter.arm.com/ (2004).

Avissar, O., R. Barua and D. Stewart, "Heterogeneous memory management for embedded systems", in "Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems", (2001).

Avissar, O., R. Barua and D. Stewart, "An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems", ACM TECS **1**, 1, 6–26 (2002).

Bai, K., J. Lu, A. Shrivastava and B. Holton, "CMSM: An Efficient and Effective Code Management for Software Managed Multicores", in "Proc. of CODES+ISSS", (2013).

Bai, K. and A. Shrivastava, "Automatic and efficient heap data management for limited local memory multicore architectures", in "Design, Automation Test in Europe Conference Exhibition (DATE), 2013", (2013).

Bai, K., A. Shrivastava and S. Kudchadker, "Stack Data Management for Limited Local Memory (LLM) Multi-core Processors", in "Proceedings of the International Conference on Application Specific Systems, Architectures and Processors (ASAP)", (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 "Proc. of CODES+ISSS", (2010).

Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems", in "Proc. of CODES", (2002).

Bershad, B., M. Zekauskas and W. Sawdon, "The Midway Distributed Shared Memory System", in "Proc. of Compcon Spring", (1993).

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 (2011a).

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** (2011b).

Bournoutian, G. and A. Orailoglu, "Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive Mobile Processors", in "Proc. of CODES+ISSS", (2011).

c. Jung, S., A. Shrivastava and K. Bai, "Dynamic code mapping for limited local memory systems", in "Proc. of ASAP", (2010).

Cai, J., Y. Kim, Y. Kim, A. Shrivastava and K. Lee, "Reducing code management overhead in software-managed multicores", in "Proceedings of the 2017 International Conference on Design Automation and Test in Europe (DATE)", (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).

Carter, J. B., J. K. Bennett and W. Zwaenepoel, "Implementation and Performance of Munin", in "Proc. of SOSP", (1991).

Carter, N. P., A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh and J. Xu, "Runnemede: An Architecture for Ubiquitous High-Performance Computing", in "Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)", HPCA '13 (2013).

Chakraborty, P. and P. R. Panda, "Integrating software caches with scratch pad memory", in "Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems", CASES '12 (2012).

Cho, H., B. Egger, J. Lee and H. Shin, "Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU", in "Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems", LCTES '07 (2007).

Choi, B., R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism", in "Proc. of PACT", (2011).

Cullmann, C., "Cache persistence analysis: Theory and practice", ACM Trans. Embed. Comput. Syst. **12** (2013).

Dominguez, A., N. Nguyen and R. K. Barua, "Recursive Function Data Allocation to Scratch-pad Memory", in "Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems", CASES '07 (2007).

Dominguez, A., S. Udayakumaran and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems", J. Embedded Comput. (2005).

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 "Proc. of CASES", (2006).

Egger, B., S. Kim, C. Jang, J. Lee, S. L. Min and H. Shin, "Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU", IEEE Transactions on Computers **59** (2010).

Egger, B., J. Lee and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an mmu", ACM Trans. Embed. Comput. Syst. (2008a).

Egger, B., J. Lee and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an mmu", ACM Trans. Embed. Comput. Syst. **7** (2008b).

Ferdinand, C. and R. Wilhelm, "Efficient and Precise Cache Behavior Prediction for Real-TimeSystems", Real-Time Syst. **17** (1999).

Francesco, P., P. Marchal, D. Atienza, L. Benini, F. Catthoor and J. M. Mendias, "An Integrated Hardware/Software Approach for Run-time Scratchpad Management", in "Proc. of DAC", (2004).

Garcia-Guirado, A., R. Fernandez-Pascual, A. Ros and J. Garcia, "Energy-Efficient Cache Coherence Protocols in Chip-Multiprocessors for Server Consolidation", in "Proc. of ICPP", (2011).

Gauthier, L. and T. Ishihara, "Implementation of Stack Data Placement and Run Time Management Using a Scratch-Pad Memory for Energy Consumption Reduction of Embedded Applications", IEICE (2011).

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors", in "Proc. of ISCA", (1990).

Gschwind, M., H. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture", IEEE Micro (2006a).

Gschwind, M., H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture", IEEE Micro **26** (2006b).

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 "Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on", (2001a).

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 "Proc. of IWCC", (2001b).

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", Proc. of Workload Characterization pp. 3–14 (2001c).

Howard, J. *et al.*, "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling", IEEE Journal of Solid-State Circuits (2011).

IBM Technical Library, "Cell Broadband Engine Architecture and its First Implementation", URL `http://www.ibm.com/developerworks/power/library/pa-cellperf/` (2005).

Ishitobi, Y., T. Ishihara and H. Yasuura, "Code and data placement for embedded processors with scratchpad and cache memories", J. Signal Process. Syst. (2010).

Janapsatya, A., A. Ignjatović and S. Parameswaran, "A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric", in "Proc. of ASPDAC", (2006).

Jang, C., J. Lee, B. Egger and S. Ryu, "Automatic Code Overlay Generation and Partially Redundant Code Fetch Elimination", ACM Trans. Archit. Code Optim. **9** (2012).

Jia, Z., Y. Li, Y. Wang, M. Wang and Z. Shao, "Temperature-aware data allocation for embedded systems with cache and scratchpad memory", ACM Trans. Embed. Comput. Syst. (2015).

Kandemir, M. T., J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif and A. Parikh, "Dynamic Management of Scratch-Pad Memory Space", in "Proc. of DAC", pp. 690–695 (2001).

Kannan, A., A. Shrivastava, A. Pabalkar and J.-e. Lee, "A Software Solution for Dynamic Stack Management on Scratch Pad Memory", in "Proceedings of the 2009 Asia and South Pacific Design Automation Conference", ASP-DAC '09 (2009).

Kim, J., S. Seo and J. Lee, "An Efficient Software Shared Virtual Memory for the Single-chip Cloud Computer", in "Proc. of APSys", (2011).

Kim, Y., D. Broman, J. Cai and A. Shrivastava, "WCET-aware dynamic code management on scratchpads for Software-Managed Multicores", in "Proc. of RTAS", (2014).

Kistler, M., M. Perrone and F. Petrini, "Cell multiprocessor communication network: Built for speed", IEEE Micro (2006).

Kontothanassis, L. and M. Scott, "Software Cache Coherence for Large Scale Multiprocessors", in "Proc. of HPCA", (1995).

Lattner, C. and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in "Proc. of CGO", (2004).

Lee, J., S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim and S. Han, "COMIC: a Coherent Shared Memory Interface for Cell BE", in "Proc. of PACT", (2008).

Levine, J. R., *Linkers and Loaders* (Morgan Kaufmann Publishers Inc., 1999).

Li, L., L. Gao and J. Xue, "Memory coloring: a compiler approach for scratchpad memory management", in "14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)", (2005a).

Li, L., L. Gao and J. Xue, "Memory Coloring: A Compiler Approach for Scratchpad Memory Management", in "Proc. of PACT", (2005b).

Lu, J., K. Bai and A. Shrivastava, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)", in "Proceedings of the 50th Design Automation Conference (DAC)", (2013).

Lu, J., K. Bai and A. Shrivastava, "Efficient code assignment techniques for local memory on software managed multicores", ACM Trans. Embed. Comput. Syst. **14**, 4 (2015a).

Lu, J., K. Bai and A. Shrivastava, "Efficient Code Assignment Techniques for Local Memory on Software Managed Multicores", ACM Trans. Embed. Comput. Syst. **14** (2015b).

Mamidipaka, M. and N. Dutt, "On-chip Stack Based Memory Organization for Low Power Embedded Architectures", in "Proc. of DATE", pp. 1082–1087 (2003).

McIlroy, R., P. Dickman and J. Sventek, "Efficient Dynamic Heap Allocation of Scratch-pad Memory", in "Proceedings of the 7th International Symposium on Memory Management", ISMM '08 (2008).

Moritz, C. A., M. I. Frank and S. Amarasinghe, *FlexCache: A Framework for Flexible Compiler Generated Data Caching* (2001).

Nguyen, N., A. Dominguez and R. Barua, "Memory Allocation for Embedded Systems with A Compile-time-unknown Scratch-pad Size", in "Proc. of CASES", pp. 115–125 (2005).

Niar, S., S. Meftali and J. L. Dekeyser, "Power consumption awareness in cache memory design with SystemC", in "Proceedings. The 16th International Conference on Microelectronics, 2004. ICM 2004.", (2004).

Nitzberg, B. and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", Computer **24**, 52–60 (1991).

Olofsson, A., "Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip", CoRR (2016).

Ophelders, F., M. J. Bekooij and H. Corporaal, "A Tuneable Software Cache Coherence Protocol for Heterogeneous MPSoCs", in "Proc. of CODES+ISSS", (2009).

Pabalkar, A., A. Shrivastava, A. Kannan and J. Lee, "SDRM: Simultaneous Determination of Regions and Function-to-region Mapping for Scratchpad Memories", in "Proc. of HiPC", (2008).

Panda, P. R., N. D. Dutt and A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications", in "Proceedings of the 1997 European Conference on Design and Test", EDTC '97, pp. 7– (IEEE Computer Society, 1997).

Panda, P. R., N. D. Dutt and A. Nicolau, "On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems", ACM Trans. Des. Autom. Electron. Syst. (2000).

Park, S., H.-w. Park and S. Ha, "A Novel Technique to Use Scratch-pad Memory for Stack Management", in "Proc. of DATE", (2007).

Poletti, F., P. Marchal, D. Atienza, L. Benini, F. Catthoor and J. M. Mendias, "An Integrated Hardware/Software Approach for Run-time Scratchpad Management", in "Proc. of DAC", (2004).

Protic, J., M. Tomasevic and V. Milutinovic, "A survey of distributed shared memory systems", in "Proc. of the Hawaii International Conference on System Sciences", vol. 1, pp. 74–84 vol.1 (1995).

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 (2014).

Reinhardt, S. K., J. R. Larus and D. A. Wood, "Tempest and Typhoon: User-level Shared Memory", SIGARCH Comput. Archit. News (1994).

REX Computing, Inc., "THE NEO CHIP", http://rexcomputing.com/ (2014).

Sandhu, H., B. Gamsa and S. Zhou, "The Shared Regions Approach to Software Cache Coherence on Multiprocessors", in "Proc. of PPoPP", (1993).

Scales, D. J., K. Gharachorloo and C. A. Thekkath, "Shasta: a Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory", SIGPLAN Not. (1996).

Sjödin, J. and C. von Platen, "Storage Allocation for Embedded Processors", in "Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems", (2001).

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 "Proc. of ISSS", (2002a).

Steinke, S., L. Wehmeyer, B.-S. Lee and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction", in "Proc. of DATE", (2002b).

Stenstrom, P., "A Survey of Cache Coherence Schemes for Multiprocessors", Computer (1990).

Tartalja, I., *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions* (IEEE Computer Society Press, Los Alamitos, CA, USA, 1997).

Texas Instrument, "TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. E)", http://www.ti.com (2014).

Texas Instruments, "TMS320C6678", URL `http://www.ti.com/product/tms320c6678` (2017).

Udayakumaran, S. and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems", in "Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems", CASES '03 (2003).

Udayakumaran, S. and R. Barua, "An integrated scratch-pad allocator for affine and non-affine code", in "Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings", DATE '06 (2006).

Udayakumaran, S., A. Dominguez and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions", ACM Trans. Embed. Comput. Syst. (2006a).

Udayakumaran, S., A. Dominguez and R. Barua, "Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions", ACM Trans. Embed. Comput. Syst. **5** (2006b).

Udayakumaran, S., A. Dominguez and R. Barua, "Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions", ACM TECS **5**, 2, 472–511 (2006c).

Verma, M. and P. Marwedel, "Overlay techniques for scratchpad memories in low power embedded processors", IEEE TVLSI **14** (2006).

Verma, M., S. Steinke and P. Marwedel, "Data Partitioning for Maximal Scratchpad Usage", in "Proceedings of the 2003 Asia and South Pacific Design Automation Conference", ASP-DAC '03 (2003).

Verma, M., L. Wehmeyer and P. Marwedel, "Cache-Aware Scratchpad Allocation Algorithm", in "Proc. of DATE", (2004).

Wilson, P. R., M. S. Johnstone, M. Neely and D. Boles, *Dynamic storage allocation: A survey and critical review* (1995).

Xu, Y., Y. Du, Y. Zhang and J. Yang, "A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs", in "Proc. of ICS", (2011).

Zhao, H., A. Shriraman, S. Kumar and S. Dwarkadas, "Protozoa: Adaptive Granularity Cache Coherence", in "Proc. of ISCA", (2013).