Compiler and Runtime for Memory Management on Software Managed Manycore Processors

by

Ke Bai

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

Approved February 2014 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Karamvir Chatha
Guoliang Xue
Chaitali Chakrabarti

ARIZONA STATE UNIVERSITY

May 2014

ABSTRACT

We are expecting hundreds of cores per chip in the near future [19]. However, scaling the memory architecture in manycore architectures becomes a major challenge. Cache coherence provides a single image of memory at any time in execution to all the cores, yet coherent cache architectures are believed will not scale to hundreds and thousands of cores [20, 22, 28, 68]. In addition, caches and coherence logic already take 20-50% of the total power consumption of the processor and 30-60% of die area [20]. Therefore, a more scalable architecture is needed for manycore architectures.

Software Managed Manycore (SMM) architectures emerge as a solution. They have scalable memory design in which each core has direct access to only its local scratchpad memory, and any data transfers to/from other memories must be done explicitly in the application using Direct Memory Access (DMA) commands. Lack of automatic memory management in the hardware makes such architectures extremely power-efficient, but they also become difficult to program. If the code/data of the task mapped onto a core cannot fit in the local scratchpad memory, then DMA calls must be added to bring in the code/data before it is required, and it may need to be evicted after its use. However, doing this adds a lot of complexity to the programmer's job. Now programmers must worry about data management, on top of worrying about the functional correctness of the program - which is already quite complex.

This dissertation presents a comprehensive compiler and runtime integration to automatically manage the code and data of each task in the limited local memory of the core [10, 11, 12, 13, 14, 15, 40, 49]. In Chapter 4, we firstly developed a Complete Circular Stack Management [12] to manage stack frames between local memories and the main memory. Then we optimize it by proposing Smart Stack Data Management (SSDM) [49]. In this work, we formulate the stack data management problem and propose a greedy algorithm for the same. Heap data is dynamic in nature and therefore it is hard to manage it. We provide a fully automatic scheme in Chapter 5 to manage unlimited amount of heap data in constant sized region in the local memory [10, 12, 14]. Later on, we propose CMSM heuristic for code mapping problem [11] (Chapter 6). Finally, in addition to those separate schemes for different kinds of data, we also provide a memory partition methodology in Chapter 9.

DEDICATION

I dedicate my dissertation work to my family, who are the foundation upon which everything else is built.

To my wife and best friend, Jing Lu: You are my backbone and the origin of my happiness. Your love and regretful support have enabled me to complete my Ph.D. Thanks to your join to our lab, Compiler Microarchitecture Lab, in the second year of your graduate life. Because of this, we can stay up several nights for conference deadlines, and eventually strive together after a lofty ideal. Without you, I would be a very different person today. Still today, learning to love you and to receive your love make me a better person. I owe my every achievement to you, Jing Lu.

To my father Xinghua Bai, my mother Guilan Tian, my father-in-law Jianping Lu and mother-in-law Guoli Zhou: Your love, understanding, support and trust encourage me to study abroad and work diligently. Your experience and personalities have affected me to be steadfast and be indomitable to difficulties. You are always proud of me, which inspirit me to do all my best in my life.

To my son, Peilin: Your perseverance affects me to move forward, no matter how challenging the life is going to be. You are a continuing source of courage. At the end of a day when I am exhausted, coming home and seeing you smiling at me eliminate all sources of stress.

ACKNOWLEDGEMENTS

Pursuing a Ph.D. is a both painful and enjoyable experience. It's like a long trip in the unforeseen sea, day by day, accompanied with bitterness, arduousness, frustration, encouragement, and trust with personal and practical support of numerous people. When I arrive at the destination, I realize that I am not lonely and teamwork helps me go through the whole trip. Although it is never enough to express my gratitude in words to my parents, my wife, all my friends and my companions, I would still like to give my many and sincere thanks to all these people for their love, support, and patience over the last few years.

I would like to give my sincere gratitude to my honorific supervisor, Professor Aviral Shrivastava. Without his inspirational guidance, his encouragements, his enthusiasm, his generous help and support throughout my graduate studies, I could never finish my dissertation at Arizona State University. Professor Aviral Shrivastava brought and motivated me into the filed of compiler and micro-architectures. He has helped me to see life and science in their full depth, and taught me how to appreciate and enjoy the good work that help other researchers to build on it. For me, he is not only a professor, but also a lifetime friend and advisor.

One of the most inspiring classes I ever had at Arizona State University was Programming for Multicore Processors by Professor Karam S. Chatha. His enthusiasm and work in the area of multicore processors finally attract me into the same field. I am grateful to Professor Chatha for his support and guidance in shaping my dissertation.

I would like to thank Professor Guoliang Xue, for introducing me to algorithm design and many classical optimization problems. All those knowledge inspire me throughout all my research work. I also acknowledge my other committee member, Professor Chaitali Chakrabarti. Her useful and insightful conversation helped to improve the quality of my thesis. I specially thank Professor Guoliang Xue and Professor Chaitali Chakrabarti for being on my committee and taking interest in my work at Compiler Microarchitecure Laboratory.

Being a member of the Compiler Microarchitecture Lab is a great experience in my life. I am lucky to be surrounded by friends rather than colleagues. Not only we studied and relaxed well together, but also they were even willing to read some portions of this dissertation and thus provided some very useful input. I wish our friendship will not die down after we are apart, but will continue and possibly flourish further. I take this opportunity to thank my friends: Reiley Jayapaul, Yooseong Kim, Jian Cai, Mahdi Hamzeh, Seungchul Jung, Fei Hong, Di Lu, Bryce Holton, Jared Pager, Abhishek Rhisheekesan, Shri Rajendran Radhika and Dipal Saluja. From weekly meetings that go on for hours in technical discussions in the lab, to team lunches, and hour long talk on our lives, I am very proud to have been with these enthusiastic researchers. Thank you for your support along the way.

I gratefully acknowledge support by grants from NSF CCF-0916652, IIP-0856090, NSF I/UCRC for Embedded Systems, Microsoft Research, Raytheon, SFAz and Stardust Foundation.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

1.1 Number of Cores is Increasing

The continuing need for higher performance is undeniable. All computing devices, from sensor nodes, watches, eyeglasses, cell-phones, tablets, laptops, desktops, servers, data centers need higher performance. However, higher performance can no longer be simply obtained by scaling up the operating frequency. This is because power increases cubically with frequency of operation, and most computing systems are limited by power, energy and thermal constraints. Embedded computing systems are often designed around battery capacity, high performance and data centers are designed around the total power draw, and the rest in the middle are designed around thermal constraints.

To improve performance without much increase in the power consumption is the goal of computing system design, and parallelism is the most promising way forward. Figure 1.1 from Intel gives an insight into the reason. The pair of blue and gray bars in the middle show the performance and power respectively of a single core system. The two bars on the left show the performance and the power of the system, if we increase the operating frequency of the cores by 20%. The bars indicate that the performance increases by about 13%, but power increases by 73%. This is again because power depends cubically on frequency, while performance is only linearly proportional – at best. The two bars on the right show the performance and power of a dual core system, when the operating frequency of the cores is reduced by 20%, and the application divided into two perfectly parallel parts, that run on the two cores. The performance of this dual core system is then 73% higher than the single core system, but the power is almost the same. In conclusion, parallelism provides a way to improve performance without much increase in power consumption. Therefore, the irrevocable trend of computer design in the near future is to increase the number of cores, while reducing the operating voltage and frequency. The new interpretation of Moore's law states that the number of cores will double every two years. We already have 2-4 cores in our phones, 6-8 cores in tablets and laptops, up to 32 cores in servers and data centers. Soon, we will have processors that have hundreds of cores! Experts from industry project over a thousand cores per chip in about a decade [19].

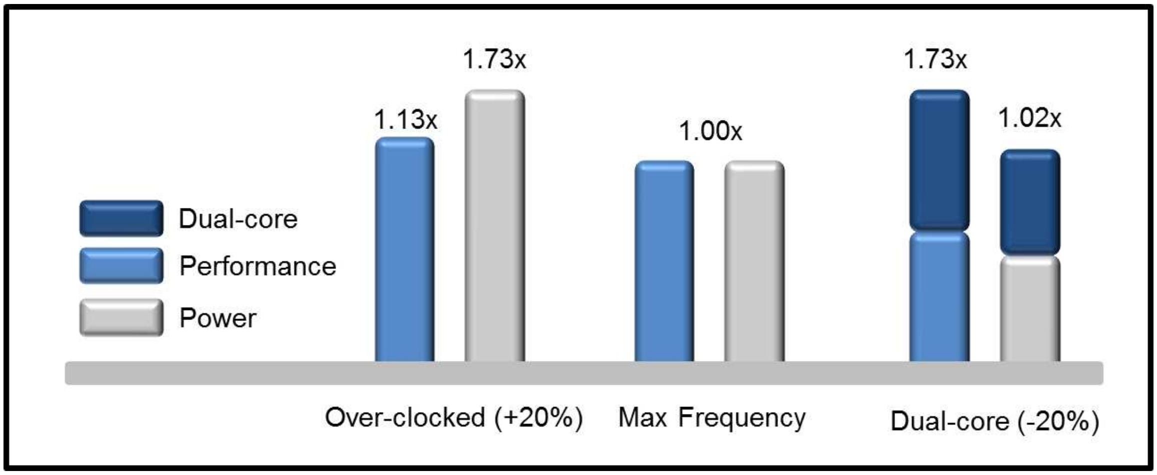


Figure 1.1: Multicore energy efficient performance.

1.2 Hundred Core Processor: Is It Just a Matter of Putting Together More Cores?

The question is – how do we build a processor with hundreds and thousands of cores? Can we design them just like our few-core processors of today, and that it is just a matter of putting more number of cores on the die, or do we have to think anew? This dissertation argues that current processor designs will not scale to hundreds and thousands of cores for two main reasons.

First is that existing cores consume too much power. For example, the 4-core Intel Core i7-3770 (Ivybridge, 22nm) consumes 77W at 3.4GHz [3]. If we use these cores to design a 100-core processor, then the power consumption of the processor would be $77 * 100/4 = 1925$W, which is clearly unsustainable [1]. Even with technology scaling, the power would not sustainable. We have to sacrifice performance of single core, if we want to put hundreds and thousands of cores on a chip. The design metric cannot be performance, it has to be power-efficiency, namely, *performance/power*.

The second reason is that the current designs of coherent-cache architectures will not scale for hundreds and thousands of cores [20, 22, 28, 68]. Coherence is implemented in hardware by mainly two mechanisms: i) snooping and ii) directory based. Snooping schemes [29] simplify the coherency problem by enforcing a unique global ordering of memory events in the processor through the use of global bus. The global bus broadcasts messages to all the cores. However, when we try to scale the design to hundreds of cores, this broadcast bus becomes the bottleneck. Directory based protocols [21, 33, 46, 58] scale better with the number of cores. The basic idea here is to keep a directory entry

[1] The current thermal cap of packaging technologies is about 250W.

for every cache block to identify the cache that contains the most up to date copy of the block. For a 1024-core processor, this directory entry will be 1024/8 = 128 bytes (in full map implementation), which is also the typical cache block size. At 100%, this is overwhelming overhead. An important point to note here is that this extra transistor requirement (for the directory) not only causes area overhead, but also significant power and performance overheads. Although there are some schemes that attempt to mitigate the space overhead of the directory, but they do so by making the directory structure distributed and hierarchical. This increases latency, but more importantly, it makes the the coherence protocol distributed. Distributed cache coherence schemes are notoriously hard to design and verify [6, 61].

In conclusion, designing processors with hundreds and thousands of cores is not an obvious extension of the processor design today. The cores need to be made more power-efficient, and the coherency problem needs to be solved in higher layers of system design.

1.3 Software Managed Manycore (SMM) Architecture

Architects and system developers are in search of designs that will scale to processors with hundreds and thousands of cores. Many experiments are being performed, both in the research space and industrial spheres.

First, in order to bypass the coherence wall, architects are experimenting with non-coherent cache architectures. The 48-core Intel Single Chip Cloud computer (Intel SCC) is a prime example [34]. Since coherence is not supported in Intel SCC, applications written in the multithreading paradigm do not work. Message passing paradigm, or scatter gather (where there is no communication between tasks) are a natural fit for such an architecture. However since multithreading is a very popular way of writing parallel programs, correct execution of multithreaded programs must be enabled; and to do that, communication management must be handled by software layers [35, 56].

Even if we use non-coherent cache architecture, the power challenge still stops us from scaling to hundreds of cores. The Intel SCC consumes 125W at 1.14V, out of which 87.7W is spent on cores [63]. If we scale the number of cores to 1000, the power consumption of the processor will be $87.7/48 * 1000$ = 1827W, which is clearly prohibitive. To enable scaling to thousands of cores, we need to reduce the power consumption of the cores by 10X.

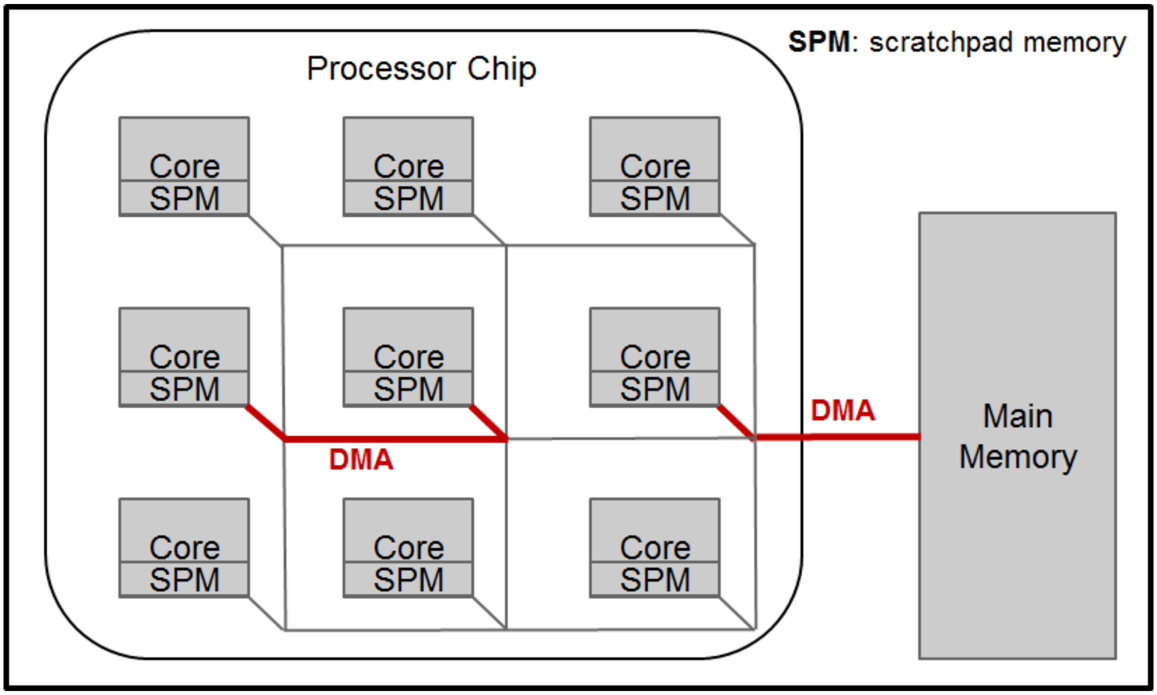


Figure 1.2: SMM architecture

Therefore, Software Managed Manycore (SMM) architecture emerges as one of the most promising scalable manycore design. Inspired by the IBM Cell processor [27], we present SMM architecture as shown in Figure 1.2. SMM architecture has only scratchpad memory (SPM) [18] in the cores and all cores on the processor share a main memory. Scratchpad memory (SPM) is fast, low power, raw memory closed to the core. As shown in Figure 1.3, SPM is quite similar to cache. The cache is primarily an SRAM array. The tag array and comparator logic in hardware is used to perform the checks to locate a recently used data. As the check is active for every memory access made by the program, the cache consumes a lot of power. On the other hand, the SPM does not use any extra logic, and only has a data array supplemented by the required address decoding logic as shown in Figure 1.3. Therefore, it consumes 30% less area and power than a direct mapped cache of the same effective capacity [18]. Each core has direct access to only its local SPM. Since SPM has no automatic hardware management as cache, all data transfers to and from other SPMs have to be explicitly specified in the software through Direct Memory Access (DMA) commands. As all the management and intelligence is relegated to software, SMM architectures prove to be extremely power-efficient, if the software does the management properly. The IBM Cell processor is a very good example of such architecture [27]. Thanks to the SPM based memory architecture, the Cell processor can compute at a power-efficiency of about 5 GFlops per

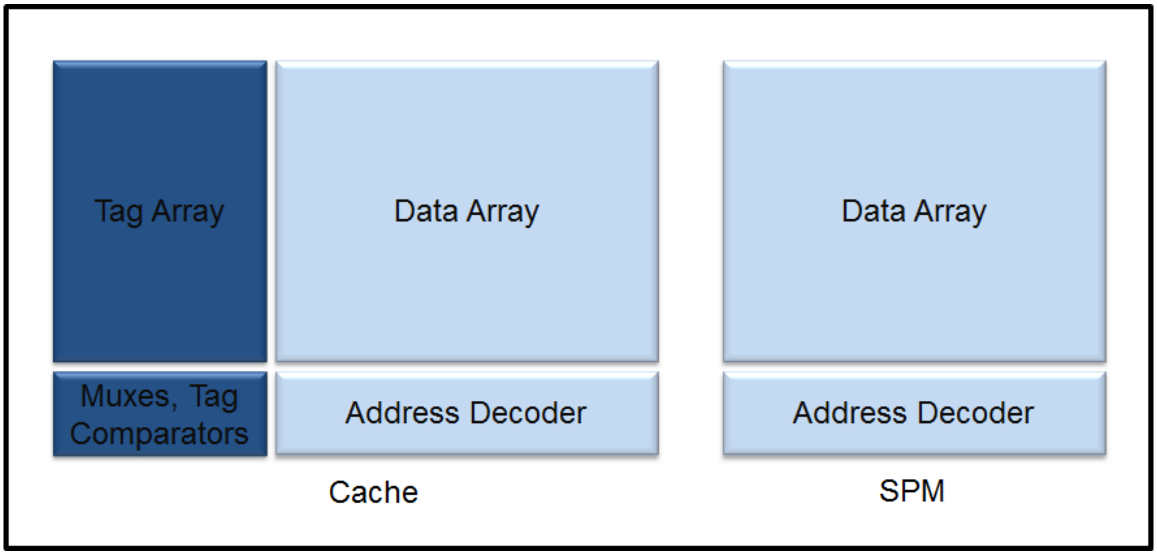


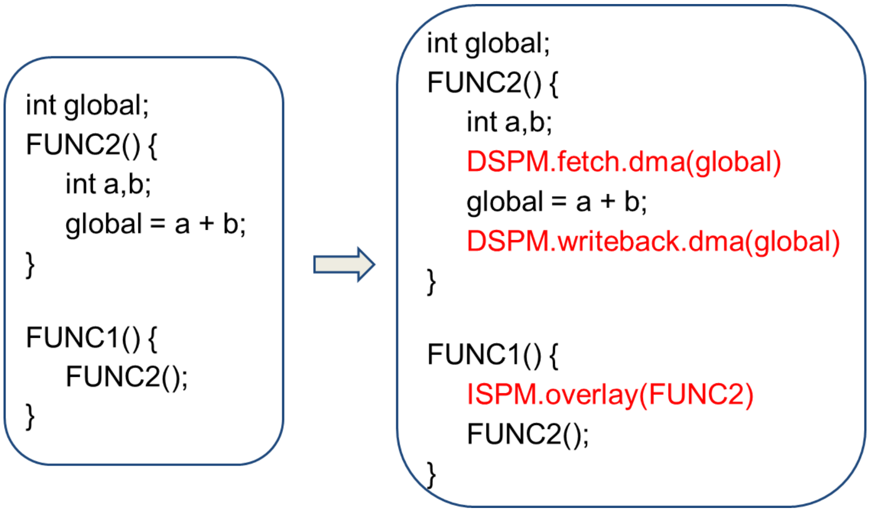
Figure 1.3: Comparison between cache and SPM

watt. In contrast, Intel i7 4-core Bloomfield 965 XE can only achieve a power-efficiency of 0.5 GFlops per watt [4, 32].

1.4 Need Advanced Compiler Technology to Enable SMM Architecture

SMM architectures are scalable memory designs, and are potentially extremely power-efficient. However, one main challenge in using SPM-based memories is that the data of the program must be managed explicitly in the software. Figure 1.4 shows that the program on the left will execute on a cache based architecture, even if the variable *global* is not in the cache at the beginning of the program. This is because cache management will bring the variable *global* from the “main memory" to the cache to execute the instruction. This automatic data movement is not provided in an SMM architecture. For such architectures, the program must be modified to as shown in the right hand side box. The *global* variable must be brought into the local SPM on each core before it is needed, and can be sent back after it is used.

The data management is explicit in the program [26, 53]. Data management can be done extremely efficiently in an SMM architecture by a programmer. This is because a programmer can have a very good understanding of when a data is needed, and when not. However, doing this adds a lot of complexity to the programmer’s job. Now programmers must worry about data management, on top



```
int global;
FUNC2() {
    int a,b;
    global = a + b;
}

FUNC1() {
    FUNC2();
}
```

```
int global;
FUNC2() {
    int a,b;
    DSPM.fetch.dma(global)
    global = a + b;
    DSPM.writeback.dma(global)
}

FUNC1() {
    ISPM.overlay(FUNC2)
    FUNC2();
}
```

Figure 1.4: Data management on caches is automatic, but in SPM-based manycore processors, data has to be managed explicitly in the program. To make a program run on an SMM architecture, the program must be changed as shown. Here global is a global variable.

of worrying about the functional correctness of the program - which is already quite complex. To better understand this complexity, it is important to understand how the scratchpad memory is allocated and managed. As shown in Figure 1.5, the application's code and data share the whole local scratchpad memory. The area below _end is the programs code and data sections, and the top of the local memory is dynamic storage. This dynamic storage is usually used for two purposes, the stack and the *malloc* heap. The stack grows downward (from high addressed memory to low addressed memory), and the *malloc* heap grows upward. They both change during the execution time. Since the local memory is a limited resource and lacks hardware-enabled protection, it is possible to overflow the space and therefore corrupt the program's code or data or both. This often results in hard to debug problems because the effects of the overflow are not likely to be observed immediately. Now, the programmer must not only be aware of the local memory available in the architecture, but also be cognizant of the memory requirement of the task at every point in the execution of the program. Estimating the memory requirement is difficult for C/C++ programs, since stack and heap sizes may be variable and input data dependent. This difficulty of programming these SMM architectures has been the biggest roadblock in the success of extremely power-efficient SMM architectures.

In summary, we expect a compiler that will automatically perform efficient data management of the application, through automatic analysis, and then also provide an interface to the programmer to

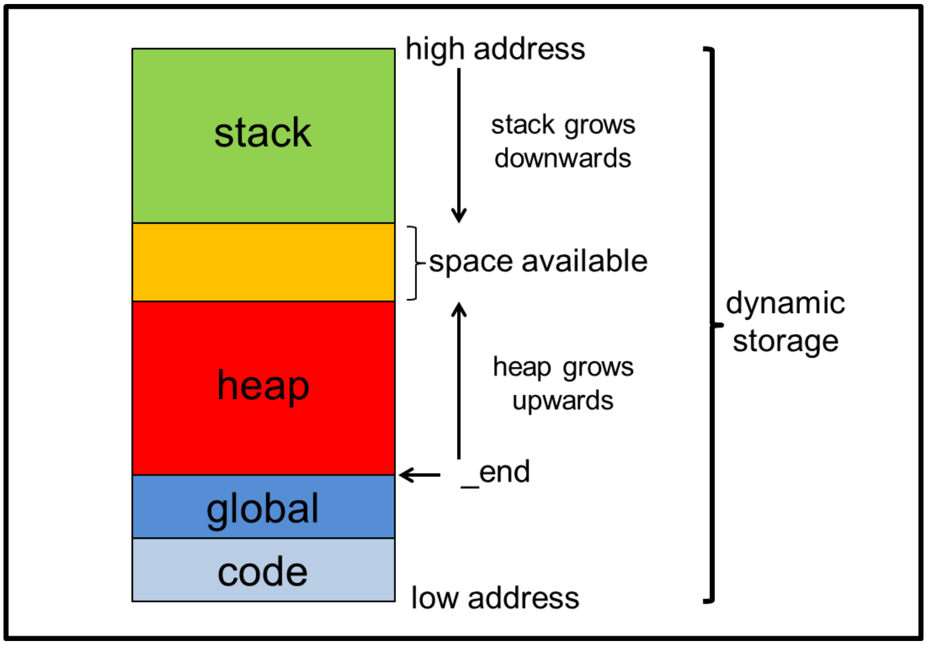


Figure 1.5: Local scratchpad memory and the anatomy of the compiled program.

improve it even further, if they need, or have time. This is the objective of our compiler and runtime system, and this dissertation.

Chapter 2

CONTRIBUTIONS OF THIS DISSERTATION

The compiler and runtime infrastructure in this dissertation automates the task of explicit memory management in the software on Software Managed Manycore (SMM) architectures. It relieves the pain of memory management by the programmer. The compiler analyzes the memory requirements of the application and inserts memory management instructions at appropriate points in the program, so that the execution is both correct and efficient.

Specifically, the contributions of this dissertation are summarized below:

- We design and implement a Complete Circular Stack Management (CCSM) framework [15] for SMM architectures (Chapter 4, Section 4.3). Although it enables the execution of programs on SMM architectures, we further optimize it by proposing another approach called Smart Stack Data Management (SSDM) [49] (Chapter 4, Section 4.5). It manages stack frames at the whole stack space granularity. It encapsulates all management functionality in five library functions. In addition, the compiler in SSDM needs to efficiently place these library functions for users. We formulate this problem, and propose a greedy algorithm for the same. Stack pointer solution in CCSM works but is further optimized with a systematic scheme (Chapter 4, Section 4.5.4).

- We implement a framework to manage heap data for SMM architectures (Chapter 5). It consists of a modified compiler and a runtime library [14]. The library contains three management functions and they are automatically inserted by our compiler. They are implemented in a low associative heap cache data structure. In addition, multi-level heap pointers are supported. Experimental results show that heap cache with low associativity performs better than heap cache with full associativity [12].

- A correct cost estimation method for code mapping is first proposed, and then a better mapping scheme called CMSM is provided [11, 40] (Chapter 6). The estimation algorithm updates the cost when determining the code mapping with any mapping algorithms. Our mapping heuristic manages code at the granularity of function object, enabling the efficient execution of applications with large code size.

- A main memory management scheme is proposed to address the problem of memory overflow on the main core (Chapter 7). It not only allocates and deallocates space efficiently, but also eliminates the explicit effort of programmers.

- An estimation scheme is proposed to partition memory space among stack data, heap data and code (Chapter 8). We can get the best partitioning among them by conducting extensive simulations, however, this takes unrealistic long time. Experimental results show that our scheme reduces this time, and generates a partitioning with which the application has comparable performance.

Chapter 3

OVERVIEW OF COMPILER AND RUNTIME INFRASTRUCTURE

Our infrastructure consists of a optimized compiler, a runtime library, and a code overlay script generating tool. Figure 3.1 shows the flow of our infrastructure. The rectangles containing italic words are our modules which could and must be embedded in the compilation of C programs. Arrows connecting components represent the flow of the component dependencies.

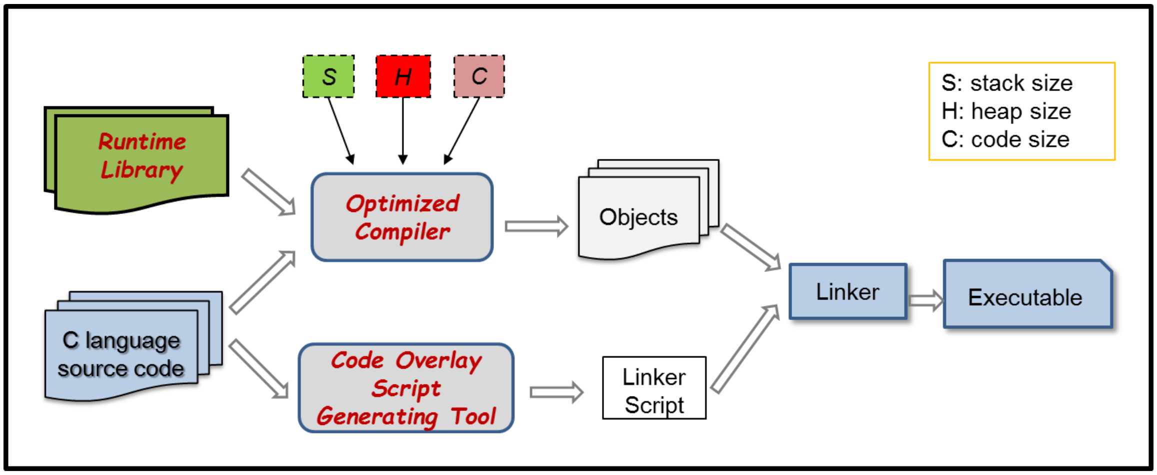


Figure 3.1: Overview of compiler and runtime framework for memory management on SMM architectures.

Our scheme has several benefits. Firstly, it increases the programmability of Software Managed Manycore (SMM) architectures. The programmers can be unaware of the memory constraints and keep programming as it is for unlimited memory space. Secondly, it extends the capability of precious works that map codes to small SPM. With our compilation framework, the codes containing dynamic memory uses on the local memory can execute safely and efficiently. Finally, the whole process is transparent to users. The memory management operations are automatically and efficiently inserted by our modified compiler.

Table 3.1 shows the interfaces that we provide to users and our infrastructure. The first two functions correspond to memory allocations and deallocations. *_sstore* and *_sload* functions manage function stack frames. *__ovly_load* is responsible for loading “to-be-execute" instructions from main memory to the local scratchpad memory. The last three functions process stack pointers and heap pointers in applications. One thing deserves to be mentioned is that all these functions will be automati-

Table 3.1: Runtime library on data and code management

Library	Functionality
_malloc	allocates space in local memory and main memory, and eventually returns a global address
_free	frees space in main memory
_sstore	uses DMA instruction to evict some or all stack frames from local memory to main memory
_sload	uses DMA instruction to fetch needed stack frame(s) in the previous stack state back to local memory
__ovly_load	load function instructions from main memory to the local memory
_g2l	translates a global address to a local address; gets the value from main memory if object misses
_l2g	translates a local address to a global address
_wb	updates data to main memory

cally inserted by our compiler at the right places. The function implementation details and their locations to be placed in the managed applications are explained in Chapter 4, Chapter 5, and Chapter 6.

Chapter 4

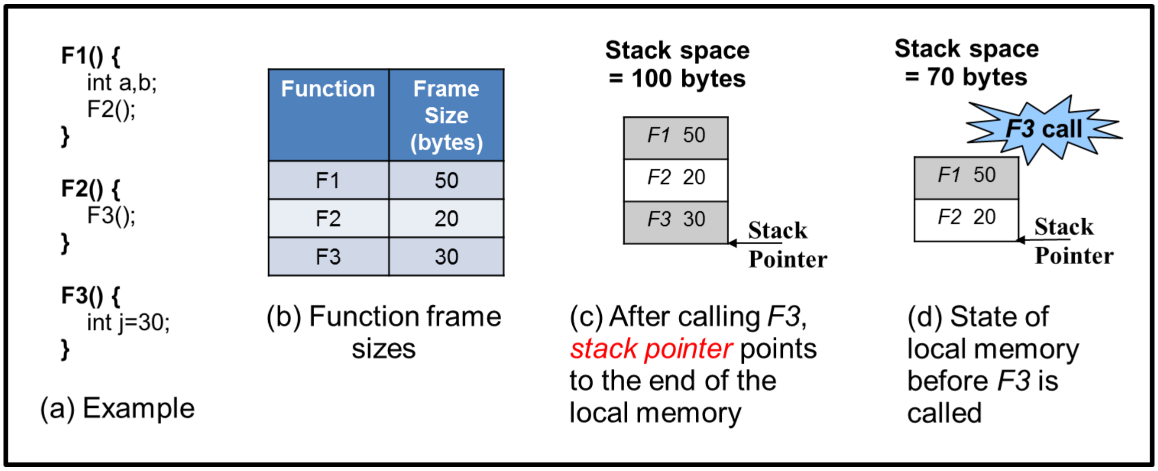
INTELLIGENT STACK DATA MANAGEMENT

In this chapter, we focus on stack data management. An efficient stack data management is crucial for program's performance, since about 64% of memory accesses in multimedia applications are to stack variables [30]. The rest of the chapter is organized as follows. Motivation is firstly discussed in Section 4.1 and then we study related work about stack data management on scratchpad memory based processors in Section 4.2. Section 4.3 present the complete stack data management scheme. Thereafter, we present challenges of efficient stack data management in Section 4.4. In Section 4.5, we present our smart stack data management. Finally, our techniques are demonstrated in Section 4.6.

4.1 Motivation

Only the local memory is accessible to the execution core and this small memory is shared by text code, stack data, global data and heap data of the thread executing on the execution core. All data should be present in the local memory when used. As a result, only a fraction of the local memory is available for managing stack data. Managing stack data is challenging as its size is non-determinable at compilation time. Stack data is dynamic in nature, namely, function frames get allocated and de-allocated at runtime, as functions are called and returned. Furthermore, the total stack size requirement of a thread may not even be known statically, for example, when recursive functions exist in the program.

The need of stack data management in a fixed sized space in the local memory is illustrated by an example in Figure 4.1. The example in Figure 4.1 (a) has three functions, whose stack frame sizes are shown in Figure 4.1 (b). Figure 4.1 (c) shows that if we have 100 bytes to manage stack data, we do not need to do anything. The application will work correctly and use up the entire space. However, if we only have 70 bytes to manage stack data, Figure 4.1 (d) shows the state of the stack just before calling function *F3*. There is no more space in the local memory, and a space of 30 bytes must be created in the local memory for allocating the stack frame of function *F3*. Without management, stack data can grow and overwrite heap data or code, and cause application crash in the best case, or simply an incorrect output in the worst.



```
F1() {
    int a,b;
    F2();
}
F2() {
    F3();
}
F3() {
    int j=30;
}
```

(a) Example

Function	Frame Size (bytes)
F1	50
F2	20
F3	30

(b) Function frame sizes

(c) After calling *F3*, *stack pointer* points to the end of the local memory

(d) State of local memory before *F3* is called

Figure 4.1: Suppose we want to execute the thread shown in (a) on the execution core with local scratchpad memory. The function frame sizes are shown in (b). (c) shows that we can easily manage the stack data of this thread in 100 bytes, however, trying to manage it in only 70 bytes (d) requires some management.

4.2 Related Work

Local memories in Software Managed Manycore (SSM) processors are raw memories that are completely under software control. They are very similar to the Scratch Pad Memories (SPMs) in embedded systems. Banakar et al. [18] noted that the majority of power in the processor was consumed by the cache hierarchy (more than 40% in StrongARM 1110). He demonstrated that this compiler controlled memory could result in performance improvement of 18% with a 34% reduction in die area. As a result, SPMs are extensively used in embedded processors, for example, the ARM architecture [1]. In SPM-based embedded processors, code and data can all be managed to use SPM, so that the application can be optimized in terms of performance and power efficiency. Schemes have been developed to manage code [7, 24, 38, 52, 54, 64], global variables [9, 41, 47, 52, 54, 64], stack data [23, 64] and heap data [55] on SPMs.

While all these works are related, they are not directly applicable for local memories in SMM architectures. This is because of the differences of the memory architecture of SPMs in embedded systems and that in SMM architectures. Figure 4.2 illustrates the major difference. It shows that embedded processors have SPMs in addition to the regular cache hierarchy. This implies that applications can execute on embedded processors without using the SPM. However, frequently needed data can be mapped to the SPM to improve performance and power, since it is faster and consumes less power [18]. On the

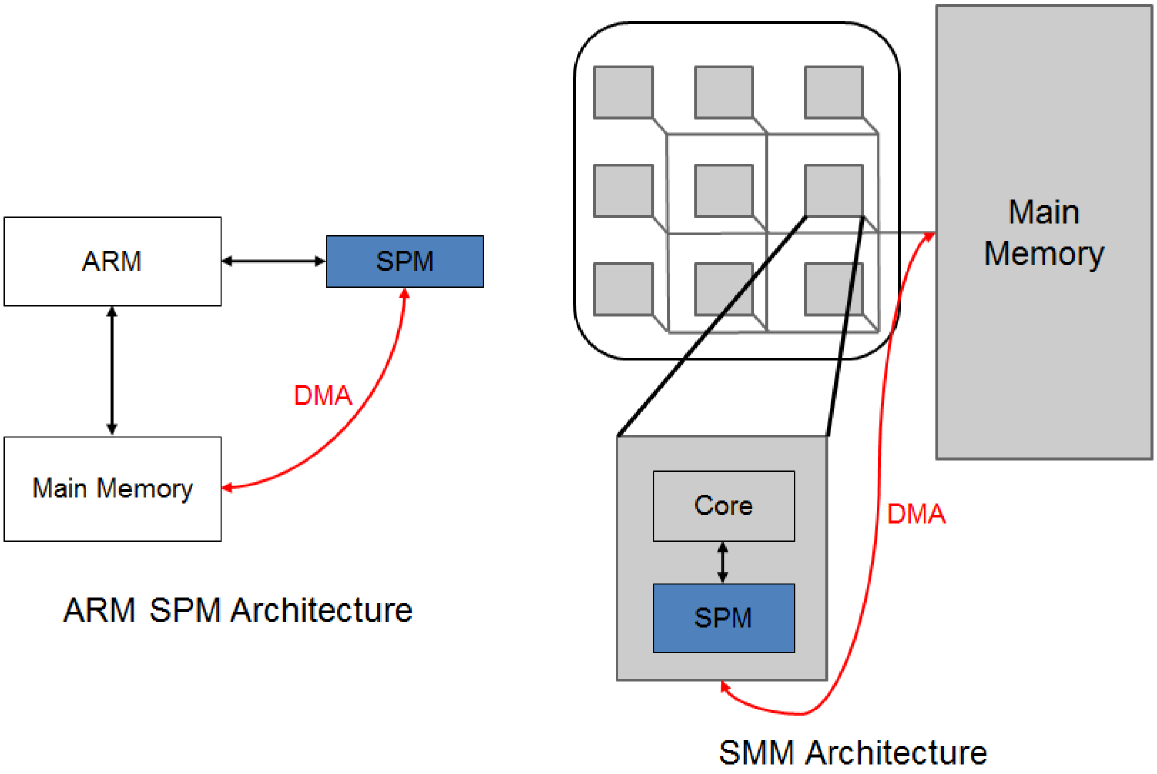


Figure 4.2: In the ARM architecture, SPM is in addition to the regular memory hierarchy, while in SMM architecture (for example, the IBM Cell processor [27]), the local memory is an essential part of the memory hierarchy on the execution core.

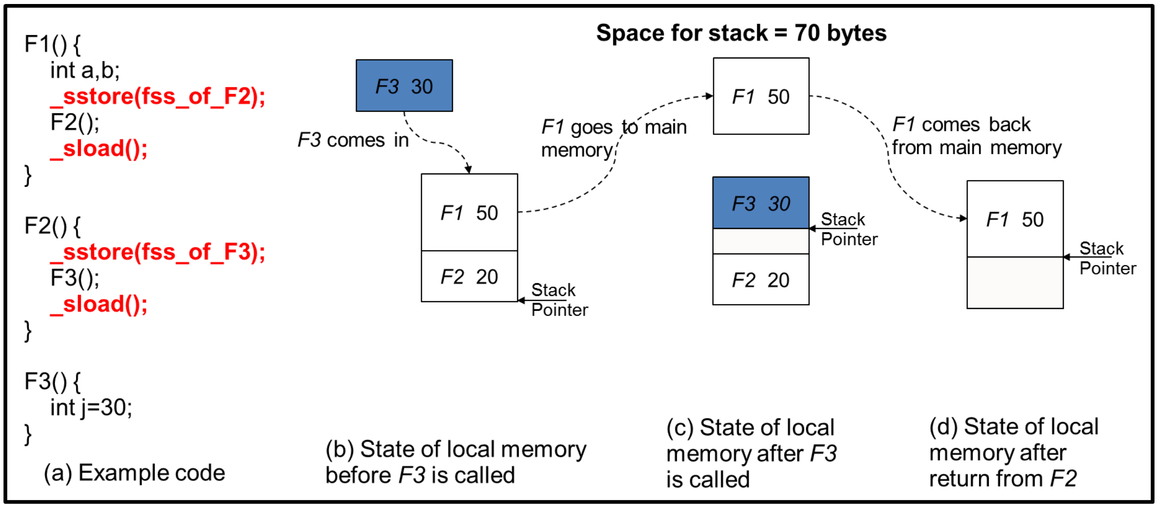
other hand, local memory is the only memory hierarchy of the core on an SMM processor. Consequently, using SPM is not an optimization problem, but is mandatory. The execution core can only access the local memory, and the data it needs must be brought into the local memory before it is accessed, or the application will not work correctly.

The stack data management techniques proposed for embedded systems in works [23, 64] only map some of the frequently accessed function frames to the SPM, and leave the rest to go through the cache hierarchy. Only the Circular Stack Management (CSM) scheme in [44] maps all stack data to the SPM, and will therefore work for SMM architectures. In this dissertation, we identify and fix several limitations of the CSM technique, to improve its applicability and generality. Section 4.3 presents the details of the complete stack management scheme for SMM architectures. We then discuss challenges of efficient stack data management in Section 4.4. Beyond that, a more efficient stack data management approach is proposed in Section 4.5. Finally we show the results in Section 4.6.

4.3 Complete Circular Stack Management

The Complete Circular Stack Management (CCSM) scheme operates at the level of function frames [15, 44]. The basic technique is to export function frames to the main memory if there is no more space on the local scratchpad memory. Figure 4.3 illustrates the functioning of CCSM. Consider the same application and function frame sizes as in Figure 4.1, and the problem is to manage stack data of the application in 70 bytes of space on local memory. Figure 4.3 (b) shows that the local memory is full after F1 calls F2, and therefore there is no more space for stack frame of F3. To make space for F3, CCSM evicts the stack frame of F1 to the main memory. This is shown in Figure 4.3 (c). After there is enough space for function frame of F3, it can execute. When F3 returns, the function frame of its ancestor F2 is in the local memory, and therefore it can execute fine. However, after F2 returns, execution returns to F1, whose function frame must be brought back into the local memory. This is shown in Figure 4.3 (d).

The eviction and fetch of function frames are achieved by using stack management Application Programming Interface (API) functions *_sstore* and *_sload*, that need to be inserted just before and after every function call. Figure 4.3 (a) shows these functions inserted in the original program in Figure 4.1 (a). The stack data management API function *_sstore(fss)* makes sure that there is enough space to accommodate the stack frame with the size *fss*. If not, it evicts as many oldest functions as required to



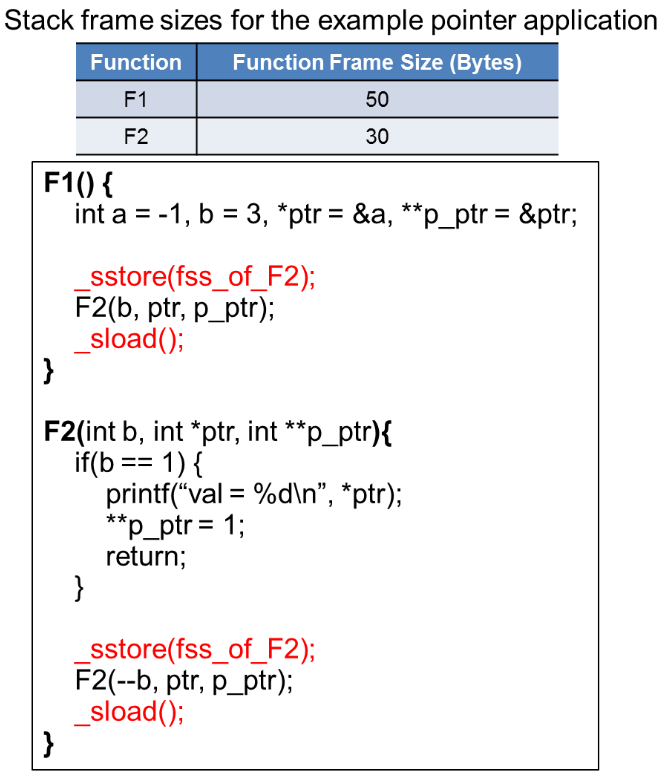
```
F1() {
    int a,b;
    _sstore(fss_of_F2);
    F2();
    _sload();
}

F2() {
    _sstore(fss_of_F3);
    F3();
    _sload();
}

F3() {
    int j=30;
}
```

Space for stack = 70 bytes
F3 30
F3 comes in
F1 50
F2 20
Stack Pointer
F1 goes to main memory
F1 50
F3 30
Stack Pointer
F2 20
F1 comes back from main memory
F1 50
Stack Pointer
(a) Example code
(b) State of local memory before F3 is called
(c) State of local memory after F3 is called
(d) State of local memory after return from F2

Figure 4.3: Complete circular stack management: The function frames can be managed in a constant amount of space in local memory using a circular management scheme. If we have only 70 bytes of space on the local memory to manage stack data, frame F1 must be evicted to the main memory to make space for F3. Before the execution returns to F1, it must be brought back to the local memory.



Stack frame sizes for the example pointer application

Function	Function Frame Size (Bytes)
F1	50
F2	30

```
F1() {
   int a = -1, b = 3, *ptr = &a, **p_ptr = &ptr;

   _sstore(fss_of_F2);
   F2(b, ptr, p_ptr);
   _sload();
}

F2(int b, int *ptr, int **p_ptr){
   if(b == 1) {
      printf("val = %d\n", *ptr);
      **p_ptr = 1;
      return;
   }

   _sstore(fss_of_F2);
   F2(--b, ptr, p_ptr);
   _sload();
}
```

Figure 4.4: An example application contains pointers to other function frame.

make enough space. Similarly, the API function *_sload()* makes sure the stack frame of the caller is in the local memory. If not, it is brought back from the main memory.

4.4 Challenges of Circular Stack Management

4.4.1 Pointer Threat

CCSM works efficiently for applications that do not have pointer references to any previous frames. However, if a function frame has a pointer reference to a variable in the evicted function frame, there is a problem. We will succinctly explain this challenge by constructing a simple program which is recursive in nature.

As shown in Figure 4.4, *a* is a local variable in function *F1*. *F1* also declares a single-level pointer, *ptr*, which points to *a*. Now this *ptr* is passed as the second parameter to *F2*. The pointer to *a* in the third argument is passed as a two-level pointer reference. The function *F2* is a recursive function.

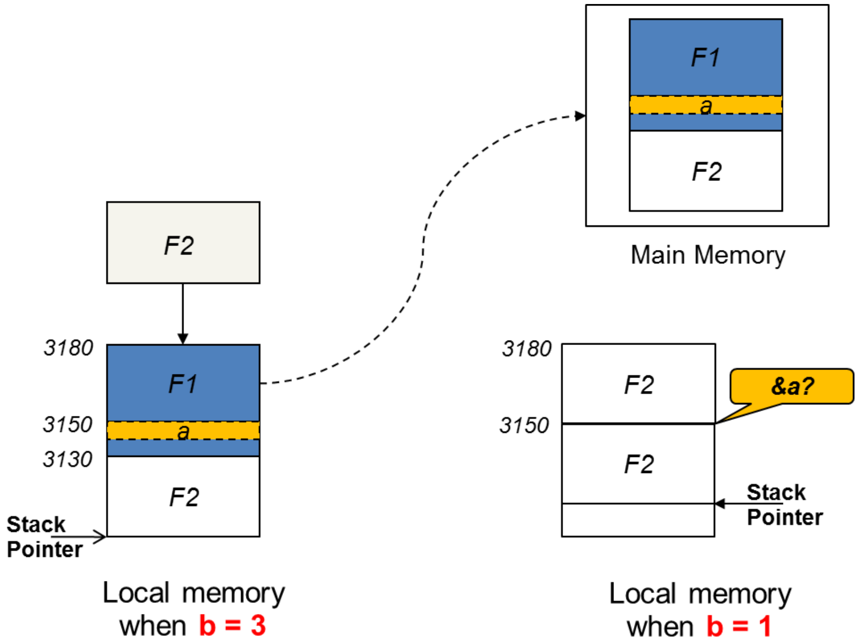


Figure 4.5: Pointer threat of the example application in Figure 4.4: When the frame of F1 is evicted to the main memory and F2 comes in, the pointer ptr in F2 which refers the local variable a in the frame of F1 cannot be referenced, as the variable does not exist in local memory. This can cause the program to crash or give wrong results.

At the tail of the recursion, the local variable *a* is accessed through pointers inside *F2*. This example uses the common programming practice of using pointers to local variables and reading/writing to them in other functions. Essentially, the function stack for the active function accesses data in other stack frames in its call path. The stack frame sizes of the functions in the example application are shown above the figure. Let us assume the SPM size be 80 bytes. Now consider executing this application with $b = 3$. The total stack space required for this application will be $50 + 30 \times 3 = 140$ bytes, which is larger than the available stack space. Therefore, we need stack data management. When *F1* is called, its function frame is created in the stack, with a location for *a*. Figure 4.5 illustrates the pointer threat of the example application in Figure 4.4. Suppose the frame of function *F1* starts at address $0x3180$, and space is allocated for *a* at $0x3150$. Then after the assignment, *ptr* contains the value $0x3150$, which is the address of *a* in the local memory. Now all goes fine until the first call to *F2*. At this point, the function frames of both functions *F1* and *F2* are in the stack. Now when *F2* (with $b = 3$) calls another instance of *F2* (with $b = 2$), the CCSM function *sstore* will remove *F1* out of the local memory, and relocate it to the global memory. When the execution calls the third instance of *F2* ($b = 1$), it falls into the base case, where *a* is accessed. They all access the contents of local memory address $0x3150$. This is clearly wrong, since the variable *a* of function *F1* is actually in main memory, and not in the local memory. If the program returns to *F1*, then the original value of *a* will be loaded – however, this is the lesser problem. This assignment will corrupt the stack frames of previous invocations of *F2*, and can lead to all kinds of failures and crashes.

The challenge here is that, the kind of code illustrated in Figure 4.4 is all too common, and is not even considered as bad programming, and this pointer problem will show up in any data management solution, and is not specific to CCSM. One way out of this is to give up, and let programmers "use pointers at their own risk". However, if we do not want to curb programmer's productivity and creativity, we need to resolve the pointer addresses, and this is not trivial. The problem is that threads are written (and should be written), assuming infinite local memory. Therefore, pointer to a variable will contain local address of the variable. If that variable is relocated to the main memory, then two things are needed to resolve the pointer correctly. First is to know that the variable has been moved out, and this is relatively easy, and can be implemented using a management table. The second problem is to find its global address. This is not easy, since we are trying to find a global address from a local address, but the relation from local address to global address is "one-to-many". The same local address may map to several global addresses. Since local memory is limited, over time several variables will be mapped to the same local memory address, but as they are relocated to the main memory, they will have different addresses there. Shrivastava et al. [57] only partially solve the second problem at several assumptions. First, functions will access data from other stack frames only through the use of direct pointers passed as parameters to it, for example, the second parameter *ptr* in the function *F2* is the one works under this assumption. However, for multi-level pointers like the third parameter *p_ptr*, their extension needs to update the address at each level of dereference. Second, pointers to stack data must not be passed within other structures. In summary, although [57] solved the pointer issue to some extent, a more comprehensive solution is required. Note that one way to solve the pointer problem is to just increase the size of local memory used to manage stack data, but then the challenge is to find how much stack space is needed. In extremely embedded contexts, the pointer safe local memory size for a given program inputs may be empirically determined by repeated simulations with several stack space allocations, and observing when execution fails or starts giving wrong results. In not-so-embedded setting, it is difficult to statically determine the maximum stack space needed because the call graph of an application may not be statically determinable, for instance, in the presence of function pointers and recursion.

4.4.2 Library Complexity

In CCSM, when there is no space for the incoming function in the local memory, the oldest function frames from the top are evicted to make space which is barely enough for the incoming function. As a result, the new function frame is instantiated as soon as enough space is available. Figure 4.3 shows that although this results in a judicious usage of local memory space for stack management, this may cause the stack space to be fragmented after some time. Consequently, to track the status of the stack, this scheme requires the book-keeping of complicated information, such as the stack size of each function, the start and end address of the free slots, etc. All those information need to be checked and updated each time the management library functions are called, which therefore slows down the application.

4.4.3 Management Granularity

Not only in SMM architectures, but also in all multicore architectures, as the number of cores increases, the memory latency of a task will be very strongly dependent on the number of memory requests. This is because memory pipelines are becoming longer, and a large part of latency is the waiting time to get the chance to access memory. Therefore, it is better to make small number of large requests, than large number of small memory requests. We expect a coarser management granularity than CCSM does, which therefore results in better performance.

4.4.4 Management Function Reduction

In CCSM, the function *_sstore()* and *_sload()* are inserted before and after each function call. Many times, these functions will not result in any data movement. For example, if there is space for the stack frame of the to-be-called function, then no DMA is required, only some book-keeping happens. Much of the overhead is due to calling these functions, even though they are not needed. An algorithm to analyze the application for judiciously placing *_sstore()* and *_sload()* functions is needed.

4.5 Smart Stack Data Management

4.5.1 Overview

Section 4.3 provides a complete solution for stack data management on SMM architectures, called Complete Circular Stack Management (CCSM). In this section, we propose another approach called Smart Stack Data Management (SSDM) built upon it [49]. In this new scheme, we further optimize CCSM by optimizing the insertion of management functions (both stack frame management functions and stack pointer management functions).

Although we have a high-level description of our whole infrastructure in Chapter 3, we delve into more details about the SSDM framework here. Figure 4.6 shows the flow of our infrastructure. Firstly, it takes the application and generates its weighted call graph (WCG). Then the SSDM greedy algorithm takes the weighted call graph and the given size of local stack space *S* as inputs and determines the right locations to insert *_sstore* and *_sload* in the managed program. Meanwhile, the results generated by SSDM are passed on to our stack pointer analyzer, which makes use of this information to figure out where to insert pointer management functions. Finally, our modified compiler GCC 4.1.1 produces the executable with our runtime library and library placement information.

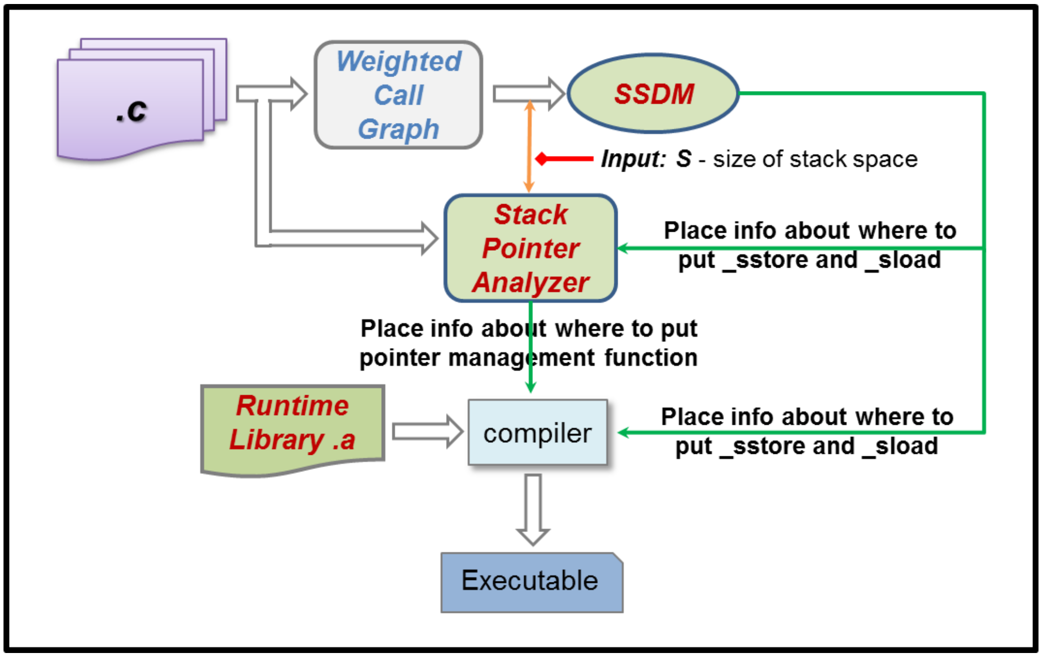

.c
Weighted Call Graph
SSDM
Input: S - size of stack space
Stack Pointer Analyzer
Place info about where to put _sstore and _sload
Place info about where to put pointer management function
Runtime Library .a
compiler
Place info about where to put _sstore and _sload
Executable

Figure 4.6: An overview of SSDM infrastructure

To optimize the stack data management, we redesigned a stack data management library with less management overhead. Different from CCSM which manages stack data at function-level granularity, SSDM performs stack data management at the whole stack space granularity. In other words, instead of evicting individual functions out of local scratchpad memory each time, our management library empties the local stack space by evicting all the contents in the scratchpad memory to the main memory. Similarly, when returning from the last frame in the local memory, the whole previous stack state is copied from the main memory to the local scratchpad memory. Performing stack data management at the whole stack level has several advantages. First is that the management complexity is largely reduced. Namely, the management library (*_sstore* and *_sload*) becomes simpler, since now the scratchpad is managed as a linear queue, rather than a circular queue. The second is that the granularity of stack data management is much coarser (than function level), and therefore there will be fewer DMA calls.

Even with so many advantages of newly implemented management library, high overhead may still happen in this scheme, if we do not judiciously place the management functions. One extreme example of such situations is that of thrashing. This happens when the stack space is full just before entering a loop with high execution count in which another function is called. Then every time the function is called, the stack state will be written back to the main memory, and reloaded on return. However, this can be avoided by carefully placing the functions *_sstore* and *_sload* in the program. In Section 4.5.2 we formulate the problem of optimal placement of these stack data management functions. We show that the management function placement problem can be described as that of finding an optimal cutting of a weighted call graph (WCG). As this problem is tractable, we then propose a heuristic (SSDM) to solve this problem efficiently.

In the following sections, our management function placement scheme is presented, and then the superiority of SSDM is demonstrated through experiments in Section 4.6.

4.5.2 Problem Formulation

We formulate our library function placement problem by using an input called *weighted call graph* (WCG). The WCG integrates flow information, control information, function stack frame sizes, and the number of times a function gets called in one place. The formal definition of WCG can be found in Definition 1.

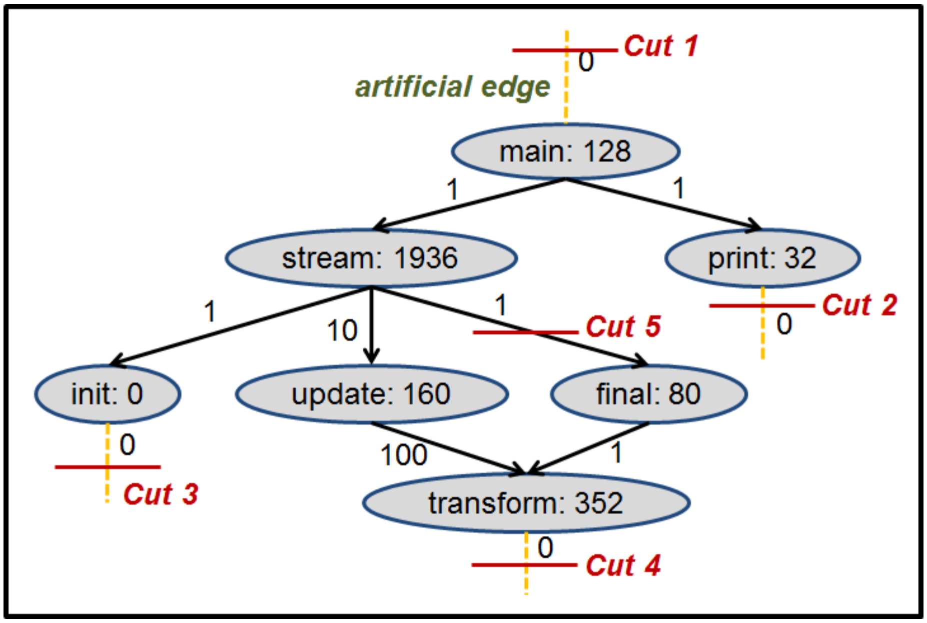


Figure 4.7: WCG with cuts of benchmark SHA: The edge with dashed yellow color represents an artificial edge for root node or leaf node.

Definition 1 *(Weighted Call Graph). A weighted call graph (V, E, W, T) contains a function node set V and a directed edge set E. Each node represents a function, and each directed edge pointing from the caller to the callee represents the calling relationship between two functions. Weight set $W = \{w_{f_1}, w_{f_2}, ...\}$ represents stack sizes of function nodes. Value on each edge e_{ij} ($e_{ij} \in E$) from the value set $T = \{t_1, t_2, ...\}$ corresponds to the number of times function node v_i calls v_j.*

Figure 4.7 shows an example of weighted call graph. Besides, we clarify several related concepts as follows.

- **root node**: A *root node* in WCG is the node with no in-coming edges. There is only one root node in the weighted call graph, which is usually the “main" function in a C program.
- **leaf node**: A *leaf node* is the node that has no out-going edges. Those are functions that do not call any other functions. Without loss of generality, however, an *artificial* in-coming edge to the root node with value 0 and an *artificial* out-going edge from each leaf node with value 0 are added.
- **root-leaf path**: A *root-leaf path* is a sequence of nodes and edges from the root to any leaf node. For example, *main-stream-init* is a root-leaf path in Figure 4.7.

- **cutting of WCG**: A *cutting of the graph* is defined as a set of cuts on graph edges. A *cut* on an edge e_{ij} ($e_{ij} \in E$) corresponds to a pair of function *_sstore* and *_sload* inserted right before and after function v_i calls function v_j, respectively. As shown in Figure 4.7, a set of cuts have been added on *artificial edges* in advance.

- **segment**: A *segment* is a list of nodes which represents the collection of nodes on a root-leaf path between two cuts. In Figure 4.7, the segment between cut 1 and cut 2 is $<$*main*, *print*$>$. A node can belong to multiple segments, e.g., node *stream* can be in both segment $<$*main*, *stream*, *init*$>$ and $<$*main*, *stream*, *update*, *transform*$>$.

As the total function frame sizes in the local scratchpad memory cannot exceed the size limit of stack space, a positive weight constraint $\mathbb{W}$ (the size of stack space) is imposed on each segment so that the total weight (stack sizes) of functions in a segment will not exceed $\mathbb{W}$. Therefore, given a segment $s = \{f_1, f_2, ...\}$ with function weights $\{w_{f_1}, w_{f_2}, ...\}$, the total weight must satisfy the weight constraint:

$$\sum_{f_i \in s} w_{f_i} \leq \mathbb{W} \tag{4.1}$$

The cost of smart stack data management (SSDM) for each segment s comprises of two components: The first one is the running time spent on extra instructions caused by *_sstore* and *_sload* function calls, and the second one is the time spent on data movement between the main memory and the local scratchpad memory. Let us assume a segment $s = \{f_1, f_2, ...\}$ is formed with two cuts on edges e_{start} and e_{end}, the functions in this segment have weights $\{w_{f_1}, w_{f_2}, ...\}$, and the two edges have values t_{start} and t_{end} (the number of function calls). Then the first part of the cost can be represented as

$$cost_1 = t_{end} \times \tau_0 \tag{4.2}$$

where τ_0 is a constant which represents the average execution time for extra instructions in the runtime library (in both *_sstore* and *_sload* function). The time spent on data movement can be estimated as linearly correlated to the size of DMA, which equals to the total function stack sizes in a segment. Therefore, the second cost can be represented as

$$cost_2 = t_{end} \times 2 \times (\tau_{base} + \tau_{slope} \times \sum_{f_i \in s} w_{f_i}) \tag{4.3}$$

where τ_{base} is the base latency for any DMA transfer, τ_{slope} is the additional latency increasing rate with data size, and 2 shows the consideration for DMA data transfer *in* and *out*.

As a result, the total cost for each segment s can be calculated as

$$cost_s = cost_1 + cost_2 \tag{4.4}$$

For a set of cuts on a Weighted Call Graph (WCG) that forms a set of segments $S = \{s_1, s_2, ...\}$, the total cost can be represented as

$$cost_{WCG} = \sum_{s_i \in S} cost_{s_i} \tag{4.5}$$

It should be noted that we treat each recursive function as a single segment and always assign a cut to it to ensure a pair of *_sstore* and *_sload* is placed right before and after recursive function calls.

Definition 2 *(Optimal Cutting of a Weighted Call Graph) An optimal cutting of a weighted call graph G contains a set of cuts that forms a set of segments, where each segment satisfies the weight constraint and the total cost of the segments is minimal.*

4.5.3 Our Heuristic: SSDM

In this section, we present our heuristic SSDM, which solves the cutting problem. The basic idea behind the algorithm is quite straightforward. At the beginning, every edge is placed with a cut. Then the algorithm gradually removes as many edges as possible one by another, until no more edge can be removed without increasing the management overhead or violating the space constraint.

In particular, when we are considering of removing a cut, we first need to check if removing this cut will violate the memory constraint of stack space. To do this, we search upward to get its nearest neighboring upstream cuts, and downward to get its nearest neighboring downstream cuts, through each *root-leaf path*. The functions between this cut and any of its neighboring cut forms a segment. If we remove this cut, the functions between any pair of upstream cut and downstream cut will form a new segment. If any of the new segment violates the memory constraint of stack space, the cut should not be removed. If the memory constraint is not violated, we will calculate how much benefit we can gain by removing this cut.

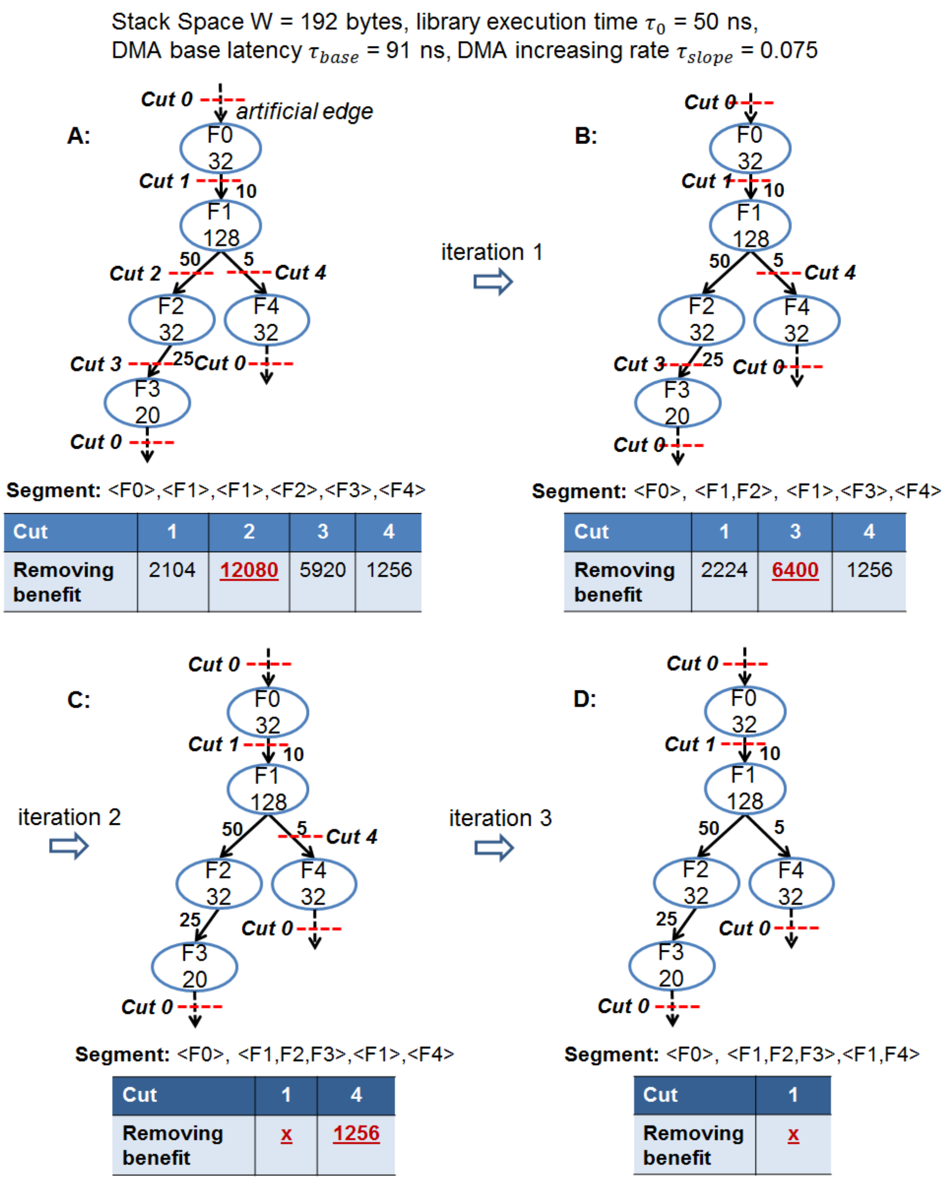
We first calculate the total management cost of all the segments associate with the cut with Equation 4.2-4.5. Then we assume this cut is removed, and construct new segments by combining upward segment and downward segment in the same *root-leaf path*, and calculate their total management

Algorithm 1 SSDM(WCG(V,E))

```
1:  Place cuts on recursive edges, if there are recursive functions.
2:  Define vector $\mathcal{C}$, in which $x_{ij}$ indicates if a cut should be placed on edge $e_{ij}$ ($e_{ij} \in E \setminus E_{recursive}$). set all
    $x_{ij} = 1$.
3:  while true do
4:      Define vector $\mathcal{B}$ to store removing benefit of each cut.
5:      for $x_{ij} == 1$ do
6:          Set boolean violate to false, it shows if removing this cut would violate the weight.
7:          constraint.
8:          Define total cost $cost_{before} = 0$.
9:          for segment $s\_old_i$ that are associated with $x_{ij}$ do
10:             Calculate cost $cost\_old_i$ with Equation 4.2-4.5.
11:             $cost_{before} += cost\_old_i$
12:         end for
13:         Assume the cut of $x_{ij}$ is removed, and get a new set of associated segments.
14:         Define total cost $cost_{after} = 0$.
15:         for new associated segment $s\_new_i$ do
16:             Check weight constraint with Equation 4.1.
17:             if weight constraint is violated then
18:                 violate = true
19:                 break
20:             end if
21:             Calculate cost $cost\_new_i$ with Equation 4.2-4.5.
22:             $cost_{after} += cost\_new_i$
23:         end for
24:         if violate then
25:             continue
26:         end if
27:         Calculate the benefit of removing the cut as $B_{ij} = cost_{before} - cost_{after}$.
28:         if $B_{ij} > 0$ then
29:             Store $B_{ij}$ into vector $\mathcal{B}$.
30:         end if
31:     end for
32:     if $\mathcal{B}$ contains no element then
33:         break
34:     end if
35:     Find out the largest benefit $B_{max}$ from $\mathcal{B}$, and set the corresponding cut $x_{max} = 0$.
36: end while
```

cost in the same way. By subtracting the newer one from the older one, we can get the *removing benefit* of this cut. We calculate the *removing benefit* of all other cuts through the same fashion. When all calculations are done, SSDM picks the largest one and indeed removes the cut associated with it. It keeps removing the cuts on WCG until no more cuts can be eliminated.

Algorithm 1 describes the complete algorithm for placing *_sstore* and *_sload* library functions. In Line 1, all recursive edges are placed with a cut. Since *_sstore* and *_sload* are statically placed at compile time and recursive function calls itself, this pre-processing eliminates the nondeterminacy of recursive functions. In line 9-12, we find out the segments that are associated with each cut x_{ij} on edge e_{ij} ($e_{ij} \in E$). To do this, we need to find out all root-leaf path P_i, where $e_{ij} \in P_i$. Then we search upward


Stack Space W = 192 bytes, library execution time τ_0 = 50 ns,
DMA base latency τ_{base} = 91 ns, DMA increasing rate τ_{slope} = 0.075
A:
B:
C:
D:
Cut 0
Cut 1
Cut 2
Cut 3
Cut 4
artificial edge
F0
32
F1
128
F2
32
F3
20
F4
32
10
50
5
25
iteration 1
iteration 2
iteration 3

Segment: <F0>,<F1>,<F1>,<F2>,<F3>,<F4>

Cut	1	2	3	4
Removing benefit	2104	12080	5920	1256

Segment: <F0>, <F1,F2>, <F1>,<F3>,<F4>

Cut	1	3	4
Removing benefit	2224	6400	1256

Segment: <F0>, <F1,F2,F3>,<F1>,<F4>

Cut	1	4
Removing benefit	x	1256

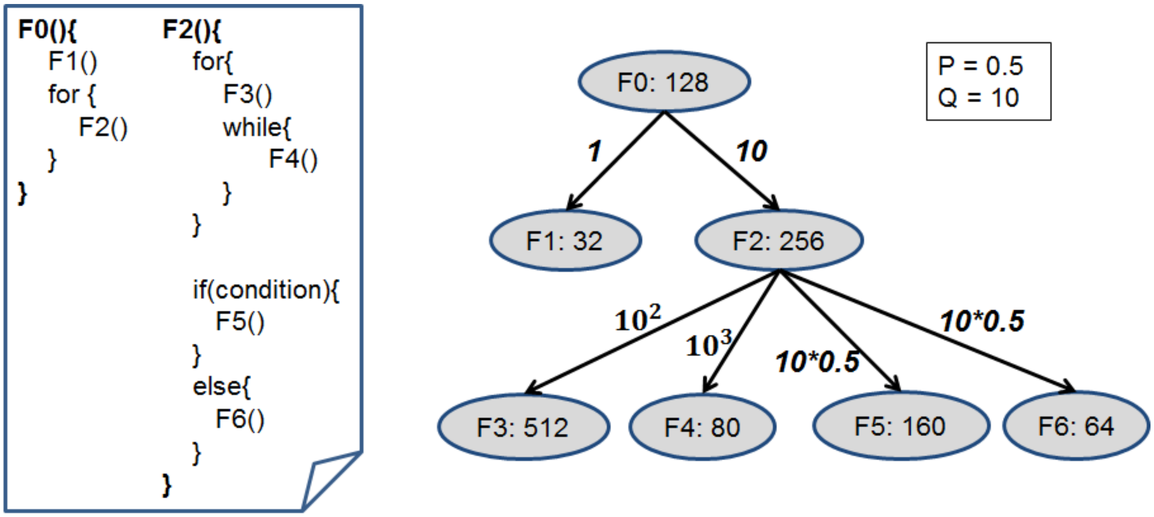
Segment: <F0>, <F1,F2,F3>,<F1,F4>

Cut	1
Removing benefit	x

Figure 4.8: Illustration of SSDM heuristic: the values on edges are the numbers of function calls.

through each P_i, until we meet a cut x_{up}. Similarly, we search downward through each root-leaf path P_i, until we meet a cut x_{down}. The segment between x_{ij} and x_{up} or x_{down} is defined as associated with x_{ij}. For example, in Figure 4.7, the segments that are associated with cut 5 is the segment <*main*, *stream*> and the segment <*final*, *transform*>. Then we calculate the cost of each segment with Equation 4.2-4.5, and the total cost by summing up the cost of all the associated segments. In Line 13-23, we assume the cut is removed, and we can get a new set of associated segments. Those segments are formed by merging the segment between x_{ij} and x_{up} with the segment between x_{ij} and x_{down} on each root-leaf path P_i. As an edge might belong to several root-leaf paths, there might be many x_{up} and x_{down} accordingly. In Figure 4.7, after removing the cut 5, the two associated segments are merged into one segment, which is <*main*, *stream*, *final*, *transform*>. Similarly, we can calculate the cost of each new segment with Equation 4.2-4.5, and the total cost of all associated segments after removing the cut. Line 16-20 check if weight constraint is satisfied by removing this cut. If the constraint is violated, this cut will not be considered to be removed (line 24-26). Line 35 removes the cut with largest positive benefit among all the cuts whose removal will not violate the weight constraint. Line 32-34 is the exit condition of the WHILE loop. The procedure stops until no more cut can be removed from the graph. At this point of time, the rest cuts either have negative removing benefit, or cannot be removed due to weight constraint (Equation 4.1).

Let us look into the illustration of SSDM algorithm shown in Figure 4.8. In this example, we are trying to manage the stack frames of the example WCG (A) in a 192 bytes stack space. When calculating the stack management cost with Equation 4.2 and Equation 4.3, we use 50 ns for τ_0, 91 ns for τ_{base}, and 0.075 for τ_{slope}. As stated before, artificial edges were added for this WCG and an artificial cut was attached for each artificial edge as well. At the initialization stage of SSDM heuristic (line 2 in Algorithm 1), we put cuts on all edges (cut 1-cut 4). Next we check the *removing benefit* of all existing cuts, except artificial cuts (line 5-29). Let us take cut 1 as an example to show how to calculate the *removing benefit*. Before removing cut 1, its associated segments are $<F_0>$, $<F_1>$ (between cut 1 and cut 2) and $<F_1>$ (between cut 1 and cut 4). The cost for $<F_0>$ is $2368 = 10 \times 50 + 10 \times 2 \times (91 + 0.075 \times 32)$ (Equation 4.2-4.4), the cost for $<F_1>$ (between cut 1 and cut 2) is $12560 = 50 \times 50 + 50 \times 2 \times (91 + 0.075 \times 128)$, and the cost for $<F_1>$ (between cut 1 and cut 4) is $1256 = 5 \times 50 + 5 \times 2 \times (91 + 0.075 \times 128)$. Therefore, the $cost_{before}$ is $16184 = 2368 + 12560 + 1256$ (line 9-12). If we assume cut 1 were removed, its associated segments become $<F_0,F_1>$ (between cut 0 and cut 2) and $<F_0,F_1>$ (between cut 0 and cut 4). We could again calculate $cost_{after}$ (line 13-23) as 14080. Thereafter, we get the removing benefit of cut 1 as $2104 = 16184 - 14080$. Similarly, we could get all removing benefit of all cuts, and form the benefit table below WCG (A). As highlighted with *underline*, we know that the largest benefit comes from removing cut 2. Then we can remove it and get WCG (B). Similarly, we can remove cuts one by one through WCG (B) to WCG (D). When we reach WCG (D), we found that we can no longer remove cut 1, as the removal of cut 1 violates our weight constraint (line 16-20), i.e., the total stack size of segment $<F_0,F_1,F_2,F_3>$ is larger than predefined 192 bytes of stack space. Till now, our SSDM stops, and therefore WCG (D) is



```
F0(){
   F1()
   for {
      F2()
   }
}
```

```
F2(){
   for{
      F3()
      while{
         F4()
      }
   }

   if(condition){
     F5()
   }
   else{
     F6()
   }
}
```

(a) Example code

(b) WCG of example code

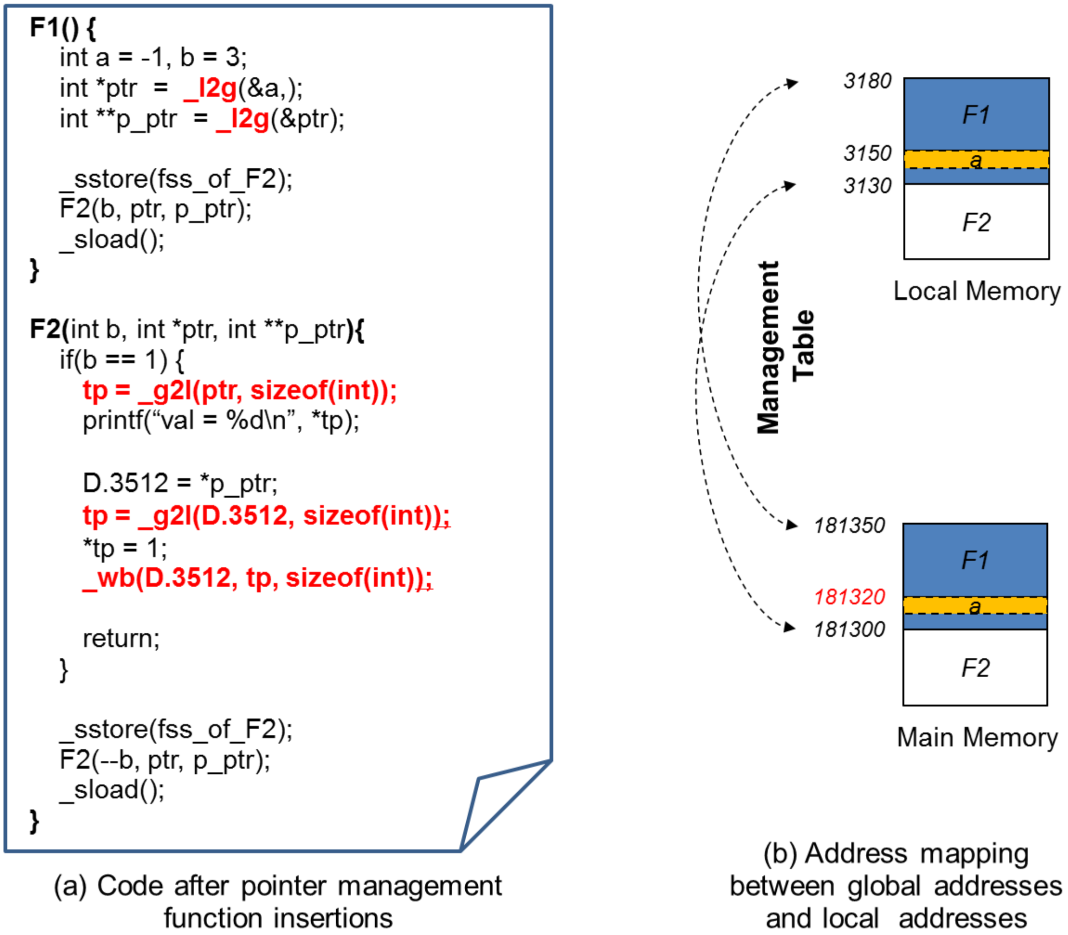
Figure 4.9: An example shows static edge weight assignment.

the final result. It indicates that the stack management function *_sstore* must be placed before F_1 gets called, and *_sload* must be placed right after F_1 returns.

4.5.3.1 Static Edge Weight Assignment of WCG

We have finished the main algorithm in the previous section. In this section, we will discuss the way to assign the value on weighted call graph (WCG). Basically, there are two ways to achieve this, static or profiling. Profiling means the numbers are obtained by running applications with inputs. Those numbers are accurate, yet this simulation based method is time consuming. Besides, we need to run the application each time a new input is given. As our goal in this dissertation is to design automatic compilation techniques, in this section, we choose to construct WCG statically.

Our construction methodology works as follows. Firstly, the basic blocks of the managed application are scanned for the presence of loops (back edges in a dominator tree), conditional statements (fork and join points) and function calls (branch and link instructions). If a function is called within a nested loop, we save the number of loops (nl) nested for that function. After capturing these information, we assign the weights on the edges by traversing WCG in a top-down fashion. Initially, they are assigned to unity. When a function node is encountered, the weight on the edges between the node and its descendants are multiplied by a fixed constant, *loop factor* Q^{nl}. This ensures that a function which is called inside a deeply nested loop will receive a greater weight than other functions. If the edge is



```
F1() {
   int a = -1, b = 3;
   int *ptr  =  _l2g(&a,);
   int **p_ptr  = _l2g(&ptr);

   _sstore(fss_of_F2);
   F2(b, ptr, p_ptr);
   _sload();
}

F2(int b, int *ptr, int **p_ptr){
   if(b == 1) {
      tp = _g2l(ptr, sizeof(int));
      printf(“val = %d\n”, *tp);

      D.3512 = *p_ptr;
      tp = _g2l(D.3512, sizeof(int));
      *tp = 1;
      _wb(D.3512, tp, sizeof(int));

      return;
   }

   _sstore(fss_of_F2);
   F2(--b, ptr, p_ptr);
   _sload();
}
```

(a) Code after pointer management function insertions

(b) Address mapping between global addresses and local addresses

Figure 4.10: Solution to pointers to stack frame: (a) Pointers are resolved by library functions. First the offset of the variable is computed using the stack start address in the local memory. The offset is used to move relatively in the main memory to reach the pointer location. (b) Our function insertions are done at the GIMPLE IR level, where multi-level pointers are dereferenced to single-level pointers, e.g., p_ptr is dereferenced with the help of another single-level pointer D.3512.

either a *true* path or a *false* path of a condition, the weight will be multiplied by another quantum, *taken probability* P. In this dissertation we assume that both paths for a condition will be executed ($P = 0.5$), which is very similar to branch predication [59]. In addition, we choose $Q = 10$. Figure 4.9 shows the resulted WCG of an example code with our static assignment scheme. In this example, the edge between function F2 and function F4 is assigned to 10^3, since F4 is in a 3-level folded loop.

4.5.4 Stack Pointer Resolution and Optimization

In this section, we first provide our solution to stack pointer threat problem, and then further optimize it when possible. The most important point in pointer resolution is that, it is not possible to resolve a pointer using local address. Thus, whenever a pointer is set, it must be set to a global address, rather than a local address. Figure 4.10, illustrates the mechanism of our pointer resolution approach. We need to change the addressing mechanism whenever a pointer to a stack variable is used. Figure 4.10 (a) shows two kinds of modifications in the application program that was shown in Figure 4.4.

The first kind of change is that the initialization of the pointer ptr is changed to $\_l2g(\&a)$ and p_ptr is changed to $\_l2g(\&ptr)$. The function *_l2g* converts the local address of a variable into a global address by first finding which function stack frame the pointer belongs to (in this example, $F1$). Then it computes the *offset* of the pointer variable (in this example, only $\&a$ is picked to show our pointer management) as the relative displacement from the start address of the frame ($F1$) in the local memory to the local pointer address. Finally, it returns a global address, which can be calculated by first getting the global start address of this function frame ($F1$) that is stored by *_sstore* function before $F1$ is called and then subtracting the displacement. Figure 4.10 shows that the stack top is at the local address $0x3180$, which is stored in the Stack Management Table, or SMT. When *ptr* is initialized, it will get the global address of the variable by the help of *_l2g* function. This is done by firstly computing the local address $0x3150$ for a. Then the *offset* is computed as $0x3180 - 0x3150 = 0x30$. The start global address of the function frame of $F1$ is looked up from the SMT, and is $0x181350$. Using these, we can compute the global address of the variable a as $0x181320$.

The second kind of change is that *_g2l*, *_l2g* and *_wb* are inserted automatically right before and after each reference. *_g2l* works directly with a global address and returns with a local address. In contrast, *_l2g* translates a local address back to a global address belonging to this pointer. If the stack data pointed by the pointer is not in the local memory, access to main memory through DMA calls is needed. If the statement contains a write operation, *_wb* updates the content in that global address. If it is a read operation, *_g2l* needs to firstly fetch the value by explicit DMA call to a buffer, and then returns its local address. When some other pointers are read/write, this buffer will be overwritten. For example, the content pointed by *p_ptr* is modified to the value 1. *_wb* function updates it directly in the main memory. Only by performing this direct main memory transaction, we do not create any data coherency problems.

One thing deserves to be mentioned is that, our compiler can process multi-level pointers in the application, utilizing the existing functionalities provided by *gcc* [2]: i) The operations containing multi-level pointers in *C* are broke down to operations containing only single-level pointers in GIMPLE

Algorithm 2 StkPtrLibPlacing(global CFG, call graph)

```
find all definitions of stack pointers and put them to $\mathbb{S}$.
do AliasAnalysis($\mathbb{S}$) and get must-alias $\mathbb{S}_t(p)$ and may-alias $\mathbb{S}_y(p)$, $\forall$ stack pointer $p \in \mathbb{S}$.
for stack pointer $p \in \mathbb{S}$ do
    boolean $flag = false$
    for $p_i \in \mathbb{S}_y(p) \cup \mathbb{S}_t(p)$ do
        $d$ = distance($L(p)$, $L(p_i)$), where $L(p)$ and $L(p_i)$ are the functions where pointers locate.
        if $d > \mathbb{W}$ or existCut($p$, $p_i$), where $\mathbb{W}$ is the size of stack region then
            $flag = true$
            break
        end if
    end for
    if flag $==$ true then
        use _l2g at $L(p)$ and _g2l & _wb accordingly for $L(p_i)$ ($\forall p_i \in \mathbb{S}_y(p) \cup \mathbb{S}_t(p)$) as shown in Figure 4.10.
    end if
end for
```

Intermediate Representation (IR), with artificial pointers generated by the compiler. An example of transformation from C to GIMPLE IR is shown in Figure 4.10 (a), where *p_ptr* is a pointer-to-pointer in *C*. In the example, a pointer write statement is transformed to two statements in GIMPLE IR, with an artificial pointer *D.3512* generated by compiler. ii) The symbol table contains abundant information about every operand in any statement. We can differentiate the type of each operand, and insert stack pointer management library only around memory references. For example, no management functions are placed around the statement “*D.3512=*p_ptr;*". It's because they both are recognized by compiler as *var_decl* type in this statement.

Although the basic solution guarantees the correctness of the program, to avoid performance degradation caused by redundant library functions, we need to only insert stack pointer management functions when necessary. Once the writeback (and reload) function placement is known, pointer management can be optimized. This is because it will become possible to know whether the function – which accesses a stack variable of an ancestor function – and its ancestor function will be in the scratchpad memory at the same time or not. We propose a systematic solution for pointer library insertions. It firstly recognizes all pointers to stack data and then utilizes classic alias analysis algorithm to collect *must-alias* set and *may-alias* set for each stack pointer [5]. Later, it calculates the total stack sizes between the pointer *define* place (or function) and *use* place in a *root-leaf path*. If the size is smaller than the predefined memory limit and no cut is found on the edge between two functions, no management is required. Otherwise, pointer management is needed.

Algorithm 2 describes the details about our systematic approach to address this problem. We use the traditional alias analysis approach to collect *must-alias set* and *may-alias set* for each stack pointer p on line 2 [5]. The *must-alias* means that two pointers are guaranteed to always point to the same memory object. The *may-alias* is used whenever two pointers might refer to the same object. Then, we utilize function *distance*[1] to calculate stack sizes of functions between p and all its alias (line 6). If one of them is larger than the size of stack space $\mathbb{W}$, then stack pointers must be managed in a fashion as shown in Figure 4.10 (line 8-16). Another situation that requires management is handled by the function *existCut*. This function takes in the stack pointer and its alias, and then checks whether there exists a *cut* between the functions where pointer and its alias locate on *root-leaf path*. When there is one, namely, the memory object pointed by these pointers are moved to the global memory, *existCut* returns *true*; otherwise, *false* is returned.

4.6 Experimental Results

4.6.1 Experimental Setup

We demonstrate the need and effectiveness of our approach by experiments on the Sony PlayStation 3 with Linux Fedora 9. It gives access to 6 of the 8 Synergistic Processing Elements (SPEs), whose local scratchpad memory size is 256KB [27]. We implemented our approach as a library with the GCC $4.1.1$. We compile and run benchmarks from the MiBench suite [30]. Their details are listed in Table 4.1. These benchmarks are not multi-threaded; we have made them multi-threaded by keeping all the input

[1] If the function that contains the *use* of stack pointer p is a recursive function (self-recursion or nonself-recursion), *distance* returns ∞.

Table 4.1: Benchmarks, the number of nodes and edges in their WCG, their stack sizes, and the scratchpad space we manage them on.

Benchmark	Nodes	Edges	Stack Size (B)	SPM Size (B)
BasicMath	7	6	400	512
Dijkstra	11	12	1712	1024
FFT	22	21	656	512
FFT_inverse	22	21	656	512
SHA	13	12	2512	2048
String_Search	11	10	992	768
Susan_Edges	8	7	832	768
Susan_Smoothing	7	6	448	256

Table 4.2: Result of stack pointer management: (a) *BasicMath* and *SHA* cannot run with the minimum stack size without our pointer management, but can run with a larger stack size after many fails of simulations. (b) Our technique resolves the pointer problem of CSM.

Benchmark	No Pointer Management				Pointer Management	
	SPM Sz (B)	Runtime (us)	SPM Sz (B)	Runtime (us)	SPM Sz (B)	Runtime (us)
BasicMath	168	CRASHES	218	1575747	168	1582033
SHA	1944	CRASHES	2024	1084	1944	1104

and output functionality of the benchmark in the main thread on Power Processing Element (PPE). The core functionality of the benchmark is executed on the SPE. Thus, each benchmark has two threads: one running on the PPE and the other on SPE. In our last experiment on scaling, we run multiple threads of the same functionality on the SPEs. The runtime for PPE was counted by *_mftb()* and the runtime for SPE was counted by *spu_decrementer()*, which are provided as the library with IBM Cell SDK 3.1.

4.6.2 Increase in Applicability

Our technique promises to run any application in the least amount of stack on SMM architectures. Given a benchmark, we find out the size of the largest stack frame, and also find out the maximum stack depth by profiling. We then run these benchmarks using space on local memory equal to the size of the largest function frame plus the maximum size of stack management table. This minimum stack size is shown in the second column of Table 4.2. We also show the runtime of the application, if it fails, *CRASHES* is printed. We can observe that benchmarks *BasicMath* and *SHA* crash without stack pointer management scheme. The sixth column lists the minimum space on the local memory required by our schemes, and the seventh column shows the time required to execute the application with this size. The main observation is that our technique successfully resolves the pointer problem, and therefore works for a wide range of benchmarks. In addition, our stack data management can work with less space on the local memory. To run an application with less stack space is important, since it provides greater flexibility for managing other data and might therefore generate a better overall performance.

4.6.3 Impact of Stack Space

In this section, we conducted another set of experiments that evaluates SSDM technique under tight size constraints. The benchmark *Dijkstra* contains many nested function calls within loop structures, making it a good candidate for showing the impact of different stack region sizes. We expanded the region size

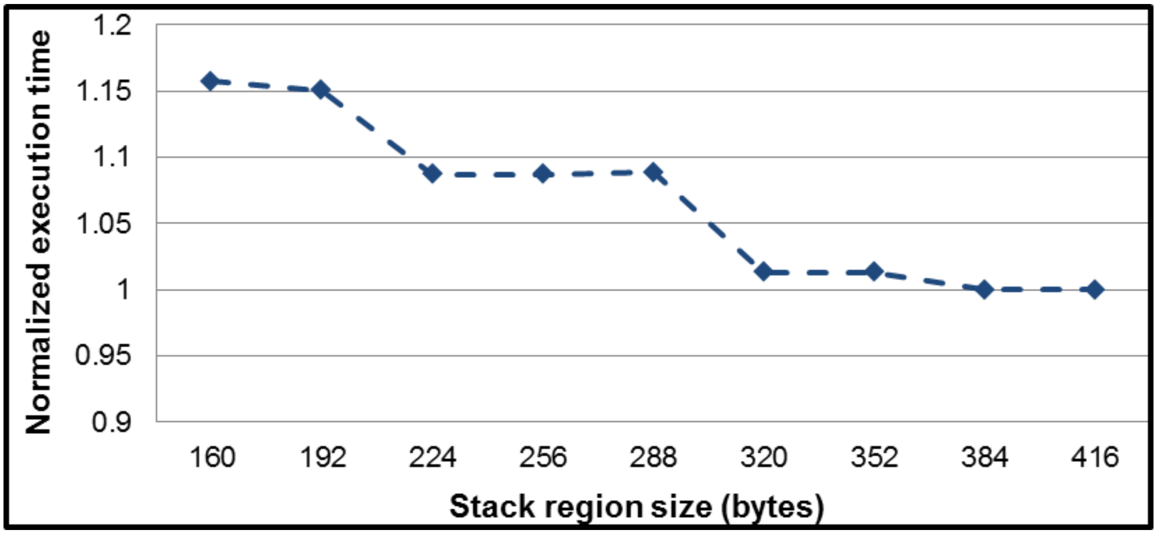


Figure 4.11: Performance - different stack region sizes.

from 160 bytes to 416 bytes with the step size of 32 bytes. The resulted performances are demonstrated in Figure 4.11, where the execution time with different stack region sizes were normalized to the smallest one. The execution time decreases when we increase stack region size. When the size reaches 384 bytes, the performance hardly improves. The primary reason is that we conservatively manage the recursive function by always placing a pair of library function around all its call sites. Therefore, although the region size is large enough, no more benefit can be obtained as only the insertion for recursive function *print_path* is left.

4.6.4 Scalability of SSDM

Figure 4.12 shows the scalability of SSDM heuristic. In the experiment, we executed the same application on different number of cores, and normalized the execution time of each benchmark to its execution time with only one SPE, and show them on y-axis. This is very aggressive, since DMA transfers occur almost at the same time when stack frames need to be moved between the global memory and the local memory. This will lead to the competition of DMA requests. As shown in Figure 4.12, the execution time increases gradually as we scale the number of cores, but no more than 1%. Benchmark *SHA* increases most steeply, as there are many pointers accessing stack data in this program. Managing pointers to stack data incurs more data transfers than general data management, because objects pointed by those stack pointers need to be transferred between the main memory and the local memory.

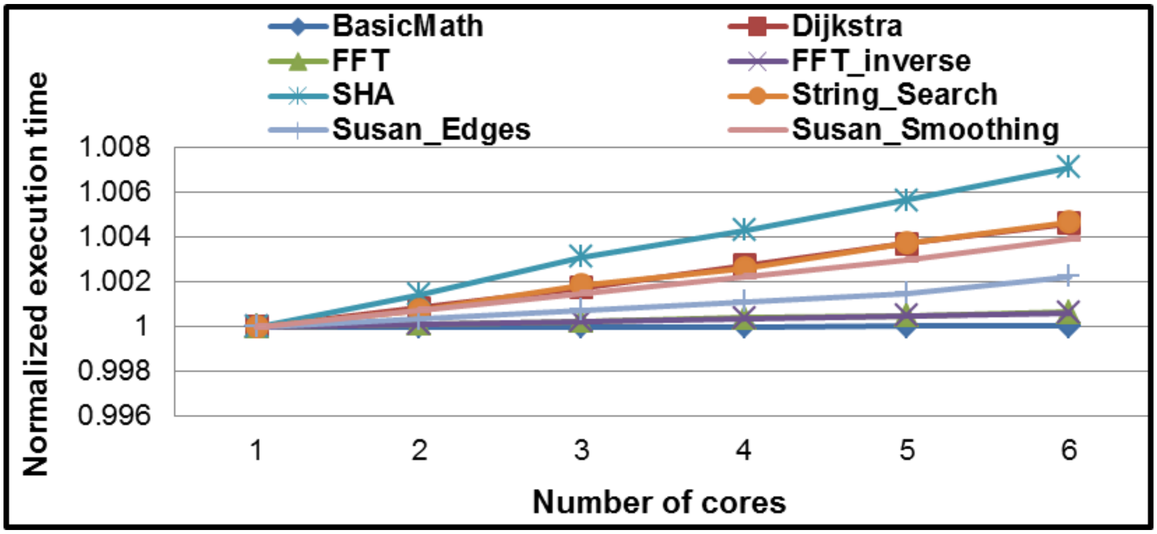


Figure 4.12: Performance - different number of cores.

4.6.5 Thorough Comparison between CCSM and SSDM

4.6.5.1 Overall Comparison

The experiment for each application in this section is conducted under the scratchpad size specified in Table 4.1. The efficiency of SSDM technique is evaluated by comparing it against CCSM presented in Section 4.3 [15]. We first utilize PPE and 1 SPE available in the IBM Cell processor and compare our SSDM performance against the CCSM result [15]. The y-axis in Figure 4.13 stands for the execution time of each benchmark normalized to its SSDM_P result. The number of function calls used in Weighted Call Graph (WCG) is estimated from profiling information for SSDM_P. In SSDM_S, we used a compile-time scheme to assign weights on edges. As observed from Figure 4.13, both the non-profiling-based scheme and the profiling-based scheme achieve almost the same performance. Compared with the CCSM scheme, SSDM demonstrates up to 19% and an average 11% performance improvement.

The overhead of the management comprises of i) time for data transfer, ii) execution of the instructions in the management library functions. Figure 4.14 compares the execution time overhead of CCSM and SSDM. Results show that when using CCSM, an average 11.3% of the execution time was spent on stack data management, while the overhead of approach SSDM is reduced to a mere 0.8% – a reduction of 13X. The gain of performance comes from several aspects. In the following subsections, we break down the overhead and explain the effect of our techniques on its different components.

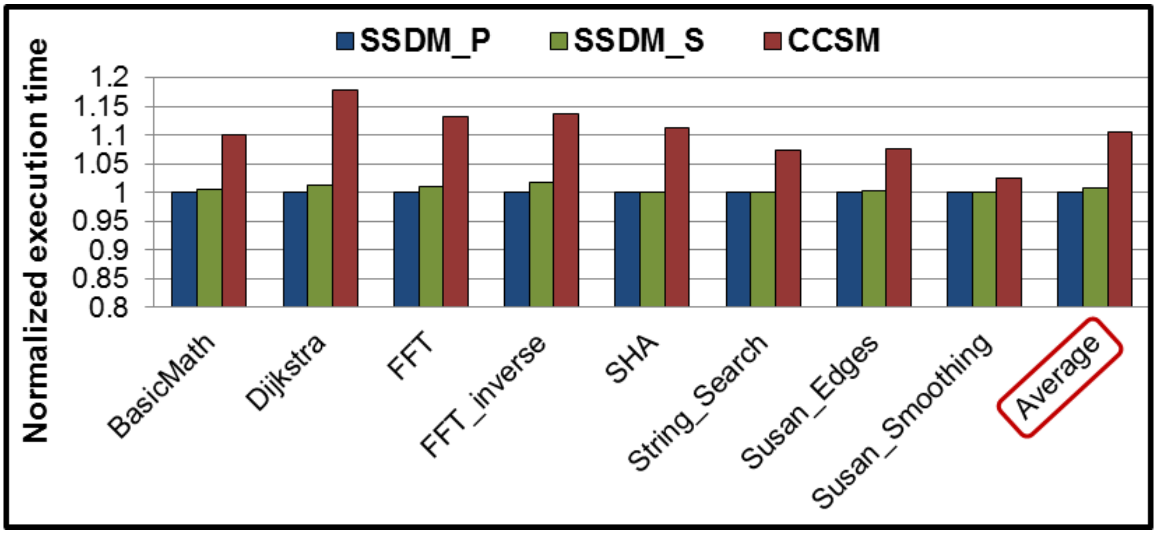


Figure 4.13: Performance comparison between SSDM and CCSM.

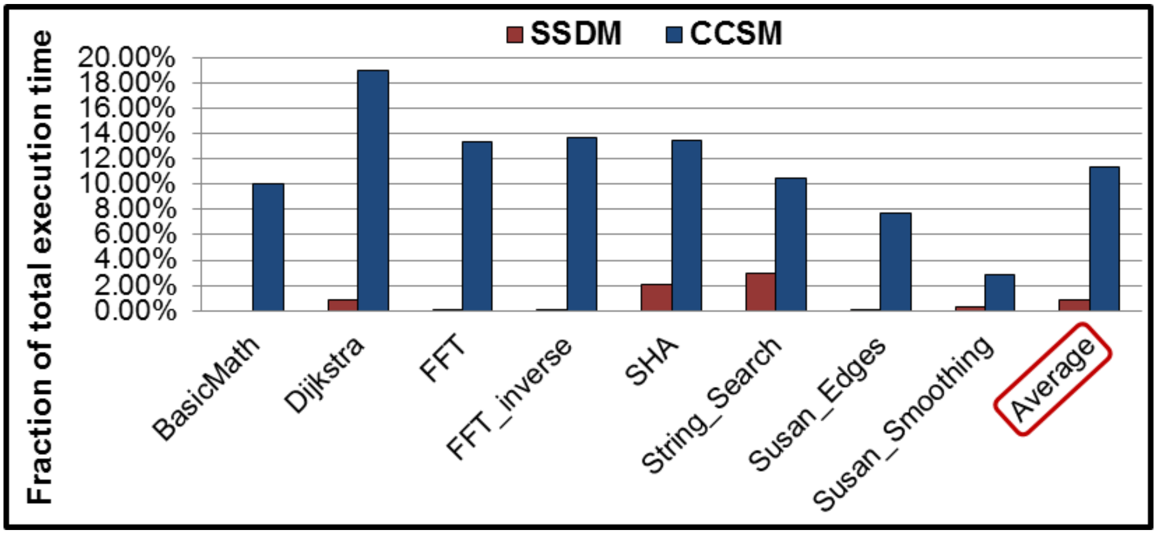


Figure 4.14: Overhead comparison between SSDM and CCSM.

4.6.5.2 Management Library Size

CCSM needs to handle memory fragmentation, while SSDM doesn't have this circumstance. Consequently, SSDM library is simpler than that of CCSM. In particular, the library functions of SSDM contain fewer instructions than that of CCSM. Table 4.3 compares the function footprint between SSDM and CCSM, from which we can find SSDM library has much smaller code size than CCSM does. Small footprint is of importance for improving the performance in two ways. First, because the management

Table 4.3: Library code size of stack manager (in bytes)

	_sstore	_sload	_l2g	_g2l	_wb
CCSM	2404	1900	96	1024	1112
SSDM	184	176	24	120	80

Table 4.4: Comparison of number of DMAs

Benchmark	CCSM	SSDM
BasicMath	0	0
Dijkstra	108	364
FFT	26	14
FFT_inverse	26	14
SHA	10	4
String_Search	380	342
Susan_Edges	8	2
Susan_Smoothing	12	4

algorithm is simpler, the execution time spent on a single management function will be less, and thus the total management overhead is reduced. Second, stack frames will obtain more space in the local memory if the library occupies less space. More space for stack data will therefore improve the management performance, which can be seen from the result in Section 4.6.3.

4.6.5.3 Management Granularity

SSDM scheme manages stack data at the stack space level granularity, which is different from the management scheme of CCSM which manages data at the function level granularity. Therefore, the number of DMA calls in SSDM is reduced. Table 4.4 shows the number of DMAs in both stack data management schemes. Note that because the whole stack of *Basicmath* fits into the local stack space, no DMA is required for this benchmark. SSDM performs well for all benchmarks, except for *Disjkstra*. This is because it contains a recursive function *print_path*. CCSM will perform a DMA only when the stack space is full of recursive function instantiations, while SSDM has to evict recursive functions every time with unused stack space. This also implies that SSDM does not perform very well on recursive applications. However, since many embedded programs are non-recursive, we leave the problem of optimizing for recursive functions as a future work.

Table 4.5: Number of *_sstore* and *_sload* calls

Benchmark	_sstore		_sload	
	CCSM	SSDM	CCSM	SSDM
BasicMath	40012	0	40012	0
Dijkstra	60365	202	60365	202
FFT	7190	8	7190	8
FFT_inverse	7190	8	7190	8
SHA	57	2	57	2
String_Search	503	143	503	143
Susan_Edges	776	1	776	1
Susan_Smoothing	112	2	112	2

4.6.5.4 Redundant Management Elimination

Thanks to our compile-time analysis, SSDM scheme can greatly reduce the number of library function calls. In Table 4.5, we compare the number of *_sstore* and *_sload* function calls in SSDM and CCSM. We can observe that SSDM has much less number of library function calls. The main reason is that SSDM considers the thrashing effect discussed in Section 4.5.1. Consequently, it tries to avoid (if possible) placing *_sstore* and *_sload* around a function call that executes many times, for example, within a loop. On the contrary, CCSM always inserts management functions at all function call sites.

The management overhead can be measured by extra instructions cause by stack management functions. Table 4.6 compares the average additional instructions incurred by each library call across all the benchmarks. As demonstrated in Table 4.6, SSDM performs much better than CCSM. *hit* for *_g2l* and *_wb* means the accessing stack data is residing in the local memory when the function is called, while *miss* denotes the case when stack data is not in the local memory. In CCSM approach, more instructions are needed for the *hit* case than the *miss* case in the function *_wb*. It is because the library directly writes back the data to the main memory when *miss*, but looking up the management table is required to translate the address. More importantly, as the table itself occupies space and therefore needs to be managed, CCSM may need additional instructions to transfer table entries.

4.6.5.5 Stack Pointer Management Optimization

Stack pointer management is properly managed in SSDM, while CCSM might manage all pointers excessively. Table 4.7 shows the results of four benchmarks *with* and *without* stack pointer optimization

Table 4.6: Dynamic instructions per function

	_sstore		_sload		_l2g	_g2l		_wb	
	F	NF	F	NF		hit	miss	hit	miss
CCSM	180	100	148	95	24	45	76	60	34
SSDM	46	0	44	0	6	11	30	4	20

* F: stack region is full when function is called; NF: stack region is enough for the incoming function frame.

Table 4.7: Number of pointer management function calls

	_l2g		_g2l		_wb	
	CCSM	SSDM	CCSM	SSDM	CCSM	SSDM
BasicMath	37010	0	123046	0	89026	0
SHA	2	2	163	158	68	68
Susan_Edges	1	0	515	0	514	0
Susan_Smoothing	1	0	515	0	514	0

technique. They are the only four applications among our eight applications that contain pointers to stack data. We can observe that our scheme can slightly improve the performance of *SHA*, and totally eliminate the pointer management functions for other three benchmarks.

4.7 Summary

In this chapter, we proposed a technique for stack data management on Software Managed Manycore (SMM) architectures, with function libraries *_sstore*, *_sload*, *_g2l*, *_l2g*, and *_wb*. Smart Stack Data Management (SSDM), is built upon Complete Circular Stack Management (CCSM). It manages stack frames at the whole stack space granularity. In addition to having reduced the complexity of runtime library, we formulated the problem of efficiently placing library functions at the function call sites. Finally, we proposed a heuristic algorithm to generate the efficient function placement. As for pointers to stack data, a proper scheme was presented to further reduce the management cost. Our experimental results show that SSDM generates function placement which leads to significant performance improvement compared to CCSM.

Chapter 5

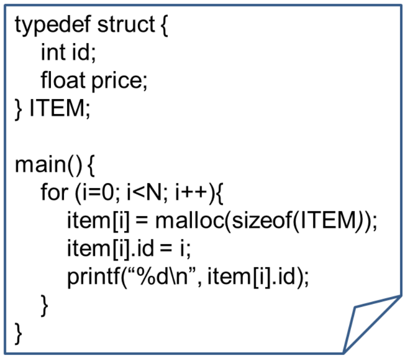
AUTOMATIC HEAP DATA MANAGEMENT

In this chapter, we will discuss how to manage heap data on SMM architectures. Unlike other data and code, heap data has no access pattern and therefore managing heap data is not trivial. We will firstly see an example about what is heap data and why it is needed to be managed in Section 5.1. Then we study the state of the art about heap data management on scratchpad memory based processors in Section 5.2. Thereafter, we propose heap data management scheme in Section 5.3. Finally, our technique are demonstrated in Section 5.4.

5.1 Motivation

In general, the local scratchpad memory on each execution core is conceptually divided into four segments by the compiler: stack data region, heap data region, global data region, and text region. Function frames reside in the stack region, starting from the top of the memory and growing downwards, while heap variables (defined through *malloc* in C language) are allocated in the heap region, starting from the top of code region and growing upwards. The text region is where the compiled code of the program itself resides. The four segments share the limited memory resource of local memory. Because the local memory lacks any hardware protection, heap data can easily overflow into the stack region and corrupt the program state.

Figure 5.1 shows an example, where heap data are defined and used. When the upper bound *N* in the loop is small, the program will execute correctly, but large values of N can cause catastrophic



```
typedef struct {
   int id;
   float price;
} ITEM;

main() {
   for (i=0; i<N; i++){
      item[i] = malloc(sizeof(ITEM));
      item[i].id = i;
      printf("%d\n", item[i].id);
   }
}
```

Figure 5.1: Outline of a program on SMM architectures: on each core, some ITEM structures are allocated and accessed.

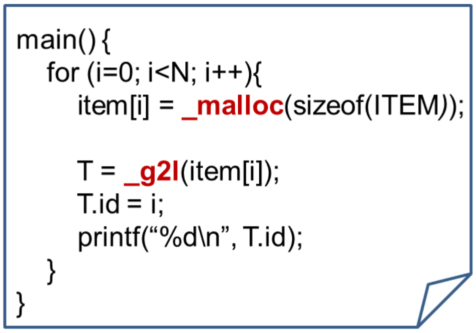
failures. For example, the application crashes, or the execution core goes into an infinite loop. However, the worst situation is that the output is just slightly incorrect. This error is hard to be observed. One way to avoid these problems is to avoid using heap variables, however, we believe that this is very limiting on both the creativity and the productivity of programmers. What is needed is a scheme that SMM architecture programmers can use to efficiently and automatically manage heap data of the application.

5.2 Related Work

As we mentioned in Section 1.3, local memory in each core of SMM architectures is a raw memory under software control. They are very similar to scratchpad memory (SPM) popular in embedded systems. Techniques have been proposed to manage stack data [9, 48, 52, 55, 64], global data [9, 42, 43, 48, 60, 64, 65, 67], and code [7, 17, 24, 25, 38, 52, 60, 64, 65, 67, 66] on the SPM, but little work has been done towards managing heap data [23, 50, 55], not even to techniques for SMM architectures.

This work only focuses on managing heap data on local memories of SMM processors, and fundamentally differs from the existing work on SPMs. The difference originates from the use of SPMs in embedded systems and local memories in SMM processors. Namely, while the problem of using SPMs in embedded systems is that of optimization, the problem of using local memory/SPM in distributed memory multicore processors is to enable the execution of applications. Therefore, previous SPM researches [23, 50, 55] have focused on the question of “what to map” on the SPM. The “what to map" is not even an option for SMM processors. Important questions in using local memories in SMM processors are: i) What API is needed to automatically manage all kinds of data in a constant amount of space in local memory? ii) How to optimize the library functions to reduce the runtime overhead? iii) How to automatically apply the API to the application?

The rest of this chapter is organized as follows. In Section 5.3, we present a fully-automatic heap data management framework in Section 5.3. Then, we compare our technique with the state of the art in Section 5.4.



```
main() {
    for (i=0; i<N; i++){
        item[i] = _malloc(sizeof(ITEM));

        T = _g2l(item[i]);
        T.id = i;
        printf("%d\n", T.id);
    }
}
```

Figure 5.2: Using our approach to manage heap data: (a) We redefine *_malloc* and *_free*. (b) Our modified GCC compiler automatically inserts a call to *_g2l* function before accessing each heap variable. (c) A temporary variale T is used, therefore local address does not need to be translated back to global address for identification.

5.3 Fully-automatic Heap Data Management

5.3.1 Overview

The objective of our approach, fully-automatic heap data management (FHDM), is to hide the additional complexity in managing heap data in a constant space in the local scratchpad memory. It is composed of a modified GCC compiler (GCC $4.1.1$), and an optimized heap data management runtime library. Figure 5.2 shows the pseudo-code of how our heap management works on the example shown in Figure 5.1. By the code transformation by the compiler, we could manage heap data correctly without translating local address to global addresses. The compiler creates a temporary pointer *T* and replaces the original heap pointer with *T* in the corresponding statement. After using our framework, users do not need to consider the redistribution of heap data; they can continue to program as if each execution core has enough memory to manage (almost) unlimited heap data. They even do not need to insert the function *_g2l* before and the function *_l2g* after any access to heap variables with our modified GCC compiler.

The fundamental challenge in Software Managed Manycore (SMM) architectures is that every variable can have two addresses, a *global address* and a *local address*, depending on where the variable is located. Our heap data management approach exposes both addresses of variables to applications. With the local address, the program can directly access the variable. If a required variable is not in the local memory, the library function *_g2l(ga)* brings it from the global address ga to the local memory and returns the local address la of the variable. Besides introducing this newly implemented function, we also re-implement two existing functions, *_malloc* and *free*. If there is enough memory space in the heap region of the local memory, the *_malloc* function can directly return a pointer. Otherwise, the function will first evict the oldest heap variable(s) to the main memory to make enough space for the coming heap variable, and then return a pointer. One important point to note here is that even if the *_malloc* function may allocate space from the local memory, it still returns the global address of the allocated heap variable each time. This is because different heap variables can have the same local memory address, but definitely have unique global addresses. As a result, we should always access heap variables through global addresses. The *_free(global_addr)* function also uses the global address of the variable and frees up space in the main memory.

Our automation is complete, and also works for multi-level pointers. To reduce the runtime overhead of the heap data management, we explore several software cache design parameters (block size, and associativity), and software cache design options (victim cache), and find efficient implementation.

5.3.2 Compiler Implementation

Our extension of compiler is based on GCC $4.1.1$. The compiler support is implemented as a pass at the GIMPLE level [2]. The reason that GIMPLE rather than AST (Abstract Syntax Tree) is chosen is that we need a common IR (Intermediate Representation) to make our pass work for all languages. In GCC, there is no single AST shared by all front-ends. There is another common IR in GCC called RTL (Register Transfer Language), but RTL is not appropriate as well. RTL is a low-level IR, and therefore high level information is lost to some extent. For example, pointer information is needed in our implementation, but the information is lost in RTL. GIMPLE is a three-address IR with tuples of no more than 3 operands (with some exceptions like function calls) [2]. It is a language independent IR and obtained by removing language specific construct from ASTs.

5.3.2.1 Insertion of Function _g2l

Applications can have three kinds of pointers – heap pointers, stack pointers, and function pointers. Stack pointers are pointers that point to its residing function frame or its ancestor function stack frames. For example, p in the statement “p = &a;" is a stack pointer. Differentiating a data pointer (stack pointer or heap pointer) and a function pointer is trivial for any compiler. Therefore, we will not delve into this part in this section. Here, we only discuss the insertion of *_g2l* for data pointers.

Algorithm 3 Insertion of Function *_g2l*

```
1:  for each basic block $bb \in \mathbb{B}$ do
2:      for each statement $s \in \mathbb{S}_i$, such that block statement list $\mathbb{S}_i$ is for $bb$, and $\mathbb{S}_i \subset \mathbb{S}$, $\bigcup \mathbb{S}_i = \mathbb{S}$ do
3:          if s contains multi-level pointers then
4:              break down to single level pointers with artificial variables,
5:              transform $s$ to statements with only single level pointers.
6:          end if
7:      end for
8:  end for
9:  for each basic block $bb \in \mathbb{B}$ do
10:     for each statement $s \in \mathbb{S}_i$, such that block statement list $\mathbb{S}_i$ is for $bb$, and $\mathbb{S}_i \subset \mathbb{S}$, $\bigcup \mathbb{S}_i = \mathbb{S}$ do
11:         if s is a modification expression then
12:             analyzeStmt ($s$)
13:         end if
14:     end for
15: end for
16:
17: procedure analyzeStmt(stmt)
18: l $\leftarrow$ getLeftOperand(stmt); r $\leftarrow$ getRightOperand(stmt)
19: /* T is a single level pointer with the same type as the pointer in the stmt, and it is created by our compiler */
20: if TREE_CODE(r) is a reference then
21:     create statement “T = _g2l($r$)"; substitute $r$ with T in stmt
22: end if
23: if TREE_CODE(l) is a reference then
24:     create statement “T = _g2l($l$)"; substitute $l$ with T in stmt
25: end if
26: insert new statement into statement list right before stmt
27: end procedure
```

As shown in Algorithm 3, the pass traverses each statement in every basic block of the application. When a memory reference is detected in line 11, *_g2l* insertion will be achieved in function *analyzeStmt* in line 12, whose body is from line 17 to line 27. When the right operand of the statement *stmt* is a reference, it will be checked in line 20. Otherwise, our pass will check whether the left reference is a reference in line 23. In both cases, our pass creates a statement “T = _g2l(ptr)"(ptr may be l or r) and inserts the statement into the statement list right before *stmt*.

Insertion of *_g2l* is conducted in a conservative way. Namely, all data pointers in the program will be added a *_g2l* function. However, our pass will not change the correctness of the application, due to the insertion for stack pointers. Our runtime library can be deployed to distinguish heap pointers and stack pointers. Because stack pointers and heap pointers are initialized with global addresses in separate regions, *_g2l* takes in the parameter and checks which region it falls into. When it is a stack pointer, the function does all work for stack pointers. In addition, as the management granularity is at

```
                   D.2348 = *ptr;
val = **ptr;   ⇨   val = *D.2348;
```

(a) C statement (b) GIMPLE IR

Figure 5.3: One example of the transformation from C code to GIMPLE IR by using our modified compiler

least one heap object, our framework will not be affected if a heap object contains a function pointer as its element. Take the statement "H→func = testFunc;" as an example. We use "H" as the parameter of function *_g2l* instead of "H→func", where *func* is a function pointer element in heap object H.

5.3.2.2 Multi-level Pointer Support

As stated before, our pass in GCC can process multi-level pointers in the application. This is achieved by breaking down the statement with multi-level pointers in C to operations containing only single-level pointers in GIMPLE IR. An example of transformation from C to GIMPLE IR is shown in Figure 5.3, where *ptr* is a pointer-to-pointer in C. In this example, a pointer read statement is transformed to two statements in GIMPLE IR, with an artificial pointer D.2348 generated by compiler. By this transformation, every statement in the GIMPLE IR only has one single-level reference. One thing needs to be mentioned is that, although D.2348 and *ptr* are both pointers, macros TREE_CODE of them return *var_decl* for D.2348 but *indirect_ref* for the latter one [2]. Because of this functionality in GCC, library calls will only be added for *ptr*.

5.3.3 Improved Management Data Structure

In this section, we propose a new low associativity heap management data structure. Compared to the fully associative data structure in the previous work [12], the new one has lower runtime overhead. It is because of that fully associative data structure adopted a complex replacement policy (LRU) for heap object replacement. In order to find the least recently used heap object for each eviction, it requires a sequential table walking to find the valid match. This operation is expensive and may happen many times, which therefore degrades the performance.

5.3.3.1 Heap Cache Data Structure

Figure 5.4 shows the design of our new heap cache data structure. It has S sets, N heap blocks and a hash function. The data structure consists of an array of S entries in heap management table (HMT) and

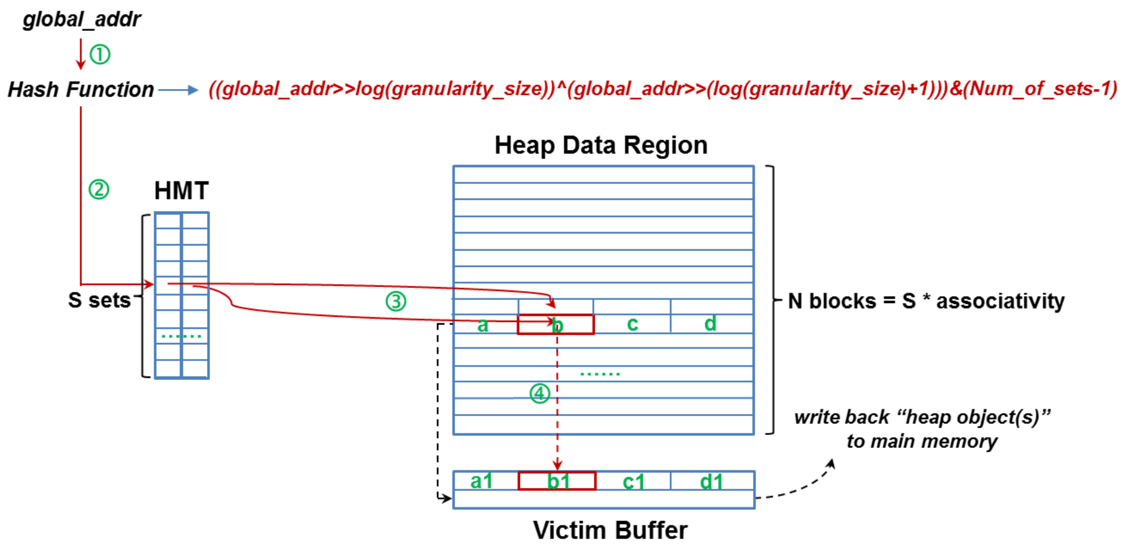

global_addr
①
Hash Function
((global_addr>>log(granularity_size))^(global_addr>>(log(granularity_size)+1)))&(Num_of_sets-1)
②
HMT
Heap Data Region
S sets
......
③
a
b
c
d
......
④
N blocks = S * associativity
write back "heap object(s)" to main memory
a1
b1
c1
d1
Victim Buffer

Figure 5.4: Processes of looking up heap objects in a 2-way associative heap cache: (a) step 4, 5, 6 might happen. (b) When an old heap block needs to be evicted, replacement runs in a round robin fashion.

an array of N heap blocks. As a set can contain several blocks, N is therefore equal to the number of sets (S) multiplies the number of associativity (A) (this figure shows a 2-way associative heap cache and we support A-way associative heap cache, where $A = 1, 2, 4, 8$). *granularity_size* and *Num_of_sets* in hash function are configured by users before compilation. Each entry in HMT contains a tag bit, a valid bit, and a modified bit and high bits of global addresses. There is a “one-to-one" static mapping between the entries in HMT and the heap blocks. Therefore, it has the property that the number of entries in HMT is equal to the number of heap blocks (N). We do not manage HMT between the main memory and the local memory, which can decrease the number of DMAs caused by moving HMT entries.

Besides the customization of heap blocks (N), heap associativity (A) and heap sets (S), we also provide a victim buffer for heap cache in the local memory. It can be used to relieve the thrashing of heap objects. When a heap object needs to be swapped out of the heap cache, our library will not directly transfer it to the main memory, but locate it to the victim buffer. By doing this, when there is a heap miss in the heap cache, we may find the data is in the victim buffer and eliminate the no slow fetch from the main memory. The victim buffer may locate one block of heap objects or other number of blocks. The optimal number is application dependent and this effect will be discussed in Section 5.4.

5.3.3.2 Implementation of _g2l Function

When *_g2l* is used for each heap access, several corresponding steps will be taken. We use Figure 5.4 to illustrate these steps as follows.

1. When *_g2l* function takes in *global_addr*, the hash function returns the set index corresponding to the requested global address.

2. After finding the set number i, the function directly goes to entry i in HMT, where tag status for set i is stored. Then the valid tags in the selected set are compared to the tag in the global address. As the figure shows a 2-way associative heap cache, the comparison in HMT will be conducted twice for set i.

3. After comparison, the function can know which block the accessing heap object should be located[1]. Then, it can further know the object offset of the accessing heap object in the cache block from *global_addr*. In this example, we suppose the offset is 1.

4. Finally, *_g2l* checks the entry i in HMT to determine whether it is in the location b.

    - If there is a valid matching entries in HMT, the request is a hit and the local address is determined by adding the object offset to the local address of the heap block that corresponds to the matching entry.

    - Otherwise, it is a miss, and the miss handler is invoked. The miss handler goes through the fully associative victim buffer to find whether the heap object is there.

        - If it is a hit, the local address of the object in the block within the victim buffer will be returned.

        - Otherwise, an old heap block following the predefined replacement policy will be selected and evicted out to victim buffer to make space for the coming one. Before overwriting heap block in the victim buffer, the modified bit of the block will be checked. If this block is dirty, it will be written back to the main memory. Otherwise, we can directly overwrite this location. Then, it fetches the heap block that corresponds to the requested global address from the main memory and places it in the evicted location.

[1] The size of block is the granularity of heap management, which means the smallest unit of data transfers.

Round Robin Replacement Policy

When an old heap block should be chosen to be evicted for N-way ($N = 2, 4, 8$) associative heap cache, *_g2l* chooses the evicted block in a round-robin fashion. Specifically, an array of counters is maintained. One counter is for one set in the heap cache. Whenever a heap block in one set is evicted, the corresponding counter will be incremented by 1 and modulo the number of blocks in one set (for example, 4 for 4-way associative heap cache). Then, when one heap block in the set needs to be selected in the future, the counter will return this candidate.

SIMD Comparison for Associative Heap Cache

Our runtime library provides N-way ($N = 1, 2, 4, 8$) associative heap cache. The tag comparisons for the implementation of 4-way associative heap cache and 8-way associative heap cache can be optimized by using SIMD (Single Instruction Multiple Data) comparison instruction supported by the accelerator core [27]. This SIMD programming operates on vector data types that are 128-bits (16-bytes) long. As one entry in HMT is a word long, and therefore 4 comparisons for a set in 4-way associative heap cache can be finished in one SIMD instruction. Accordingly, 8-way associative heap cache requires 2 SIMD instructions for its 8 comparisons of a set.

5.4 Experimental Results

5.4.1 Experimental Setup

We run several benchmarks on the IBM Cell processor [27] with operating system Fedora 9 and 6 accesses of the 8 SPEs. The benchmarks include applications from Mibench suite [31] and some other possible applications. The benchmarks from Mibench are modified to be multi-threaded, where just input and output happen on the main core, and the rest happens in the execution core. Table 5.1 shows the details of the heap size requirement of each benchmark. MIN is the minimum size needed by our methods, while MAX is the total amount of heap space required by the application with the standard input set (without management). *yes* of Exceed means the application needs to be managed, as it is larger than available space for heap data in the local scratchpad memory. Available space means the subtraction of the size of local memory (i.e., 256KB) and sizes of stack data, code and global data required by the program.

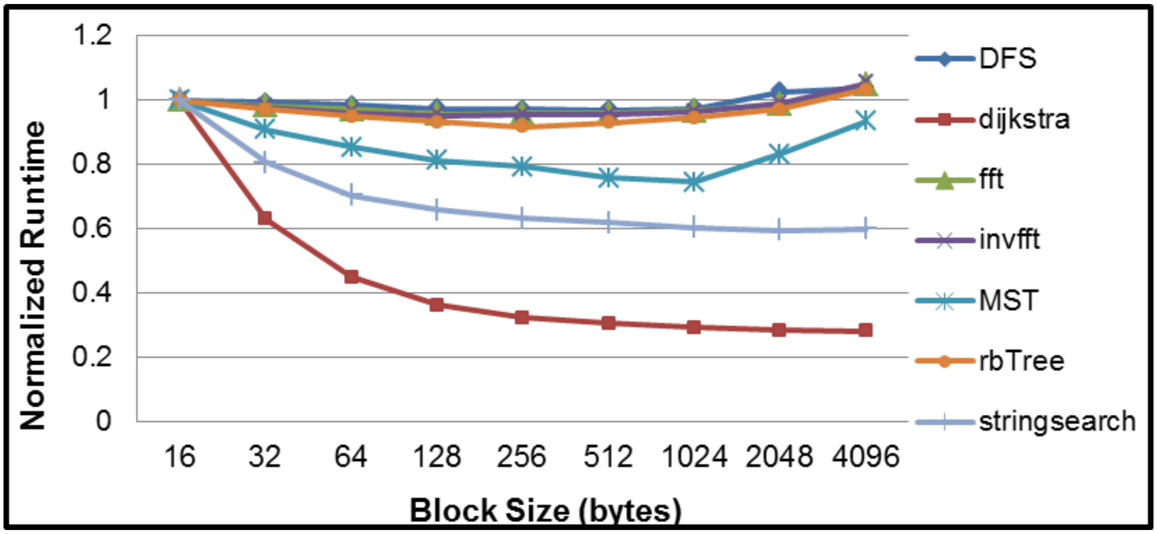


Figure 5.5: Impact of block size (in Bytes)

5.4.2 Impact of Heap Cache Parameters in FHDM

Heap design space is immense. We can choose different block sizes, set associativities, victim buffer sizes and number of sets. In this section, we performed many simulations with different configurations to evaluate the performance of benchmarks.

5.4.2.1 Block Size

The first result we show in Figure 5.5 is the impact of the block size for heap cache. In this experiment, we configure heap cache to use all available space in the local memory. In addition, there is no victim

Table 5.1: Heap requirement of all benchmarks

Benchmarks	Heap Size (bytes)		Exceed
	MIN	MAX	
basicmath	0	0	no
DFS	16	16000	yes
dijkstra	16	24064	yes
fft	16	262208	yes
invfft	16	262208	yes
MST	16	24576	yes
rbTree	32	49152	yes
sha	0	0	no
stringsearch	16	4096	no

buffer in the data structure and heap objects are managed in 4-way associative fashion. We varied the block size from 16 bytes to 4096 bytes[2] A larger block size provides the functionality of prefetching as it brings in a few nearby elements together to the local memory. In Figure 5.5, we normalized the execution time of application with other block sizes to that with block size 16 bytes. We can observe that, the performance of *dijkstra* and *stringsearch* can be improved by simply increasing the block size. Because they access data sequentially in nature, a large block size takes advantage of data prefetching and higher data transfer bandwidth. In contrast, other benchmarks can get the best performance with block size between 256 bytes and 512 bytes, since there is a trade-off between the transfer granularity and the data locality. When the block size is small, increasing the block size can increase the reuse of the data. After the block size reaches a certain extent, increasing the block size will not change the data locality of the application, while the overhead of transferring larger data increases and finally beats the benefit of better data locality.

5.4.2.2 Set Associativity

The second factor we have tried to explore is the set associativity of our heap cache. In this experiment, we changed the set associativity of heap cache from direct mapped to 8-way associative, without a victim buffer. Besides, the heap cache size is varied from the minimum size to all the available space in the local memory, and the block size is tested from 16 bytes to 4096 bytes. We show the average execution time for each associativity in the y-axis of Figure 5.6. The average runtime with N-way(N=2,4,8) associative management is normalized to the average runtime with direct mapped management. We can observe that the 4-way associative heap cache can improve performance of benchmarks *DFS*, *fft*, *invfft* and *MST*. However, the performance are even worse with the higher associativity for benchmarks *dijkstra* and *stringsearch*. A higher associativity for heap cache can decrease heap data miss rates and therefore reduce the number of DMA transfers, while it in turn requires more computations in software implementation, i.e., the computation spent on looking up the management data structure. If the benefit brought by high hit ratio beats the computation overhead, higher associativity is better. Otherwise, higher associativity will just degrade the performance.

[2] The block size is also the minimum data transfer unit between the main memory and the local scratchpad memory (can be termed as granularity).

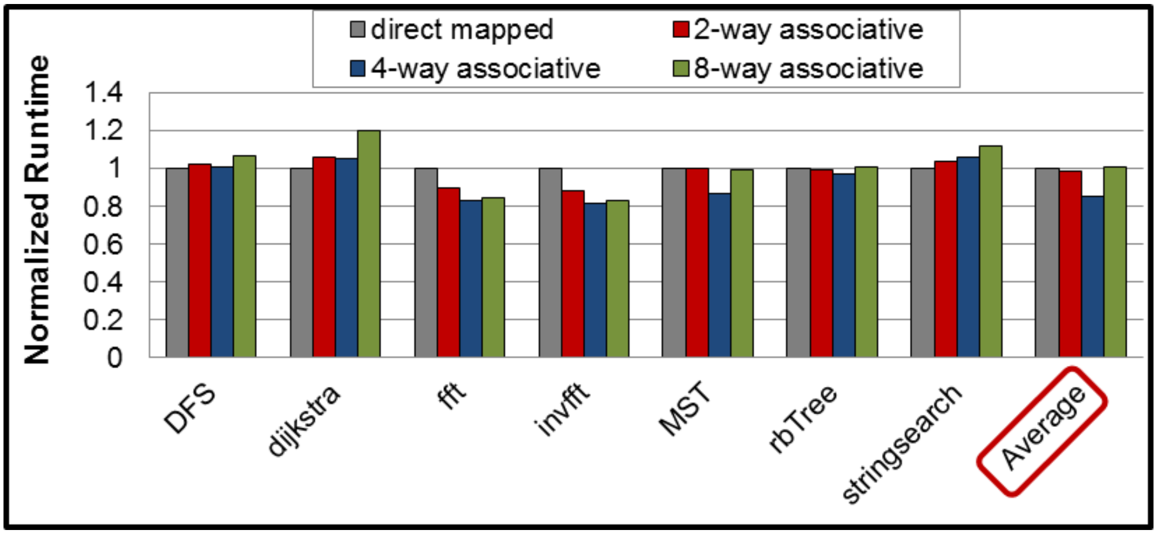


Figure 5.6: Impact of set associativity

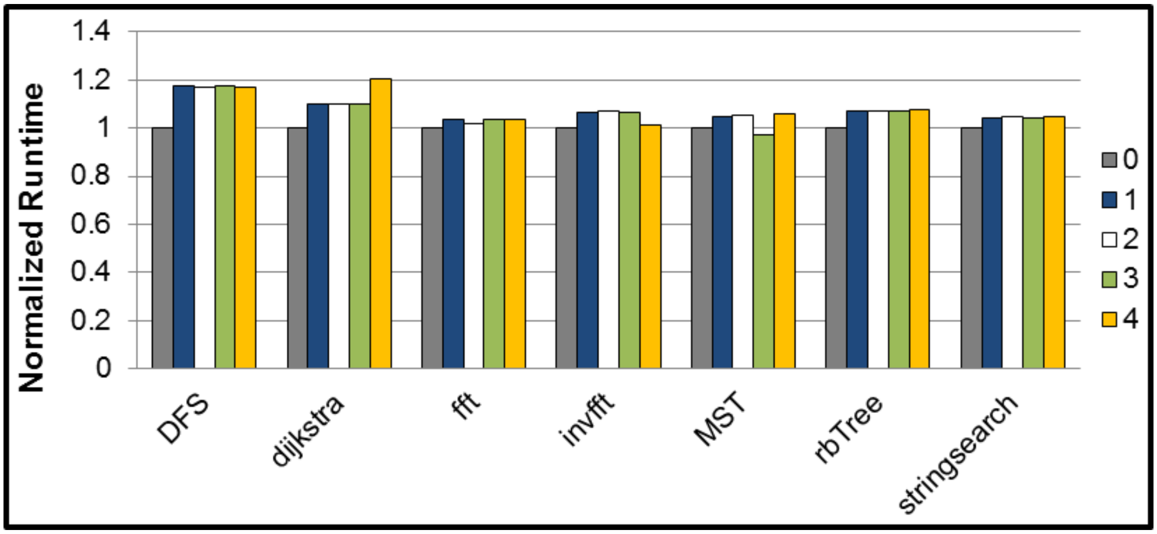


Figure 5.7: Impact of victim buffer (number of blocks)

5.4.2.3 Victim Buffer

Heap data are moved between the main memory and the local scratchpad memory. One problem of this is the effect of trashing, for example, a heap object is just swapped out of the local memory, but is needed in the new future. In order to relieve thrashing of heap objects, a victim buffer is implemented in the heap management buffer. The victim buffer holds the just-evicted heap objects temporarily, and provides the data if accessed.

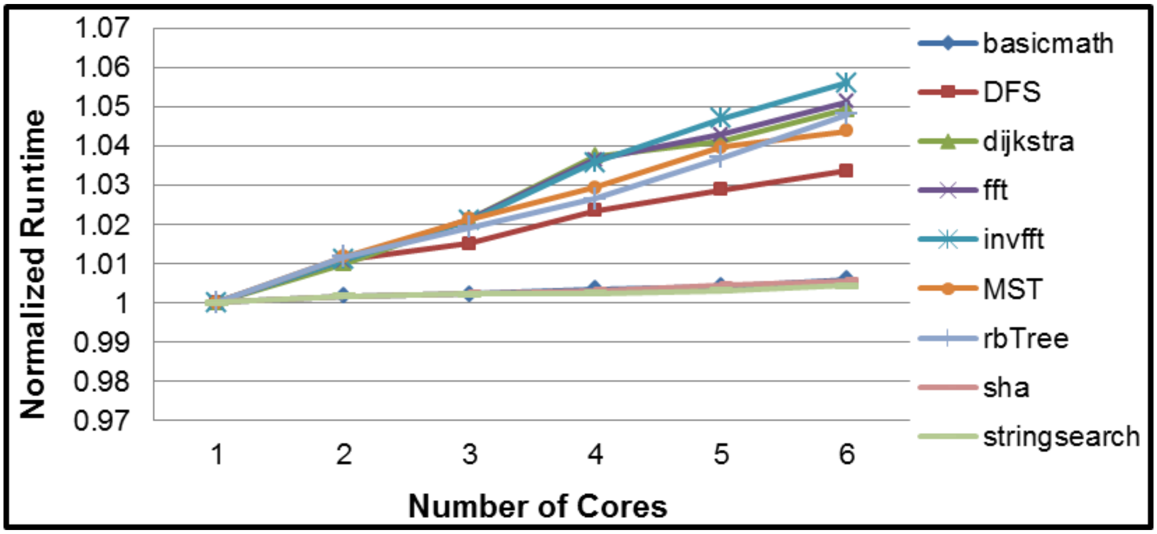


Figure 5.8: Performance - different number of cores.

In this experiment, we assigned all available space in the local memory to heap cache, changed the block size from 16 bytes to 4096 bytes, and set heap cache as 4-way associative. We then varied the number of blocks in the victim buffer from 0 to 4, and show the results in Figure 5.7. All results with N $(N = 1, 2, 3, 4)$ number of blocks in the victim buffer are normalized to the result with 0 block in the victim buffer. Each bar shows a different number of blocks in the victim buffer, for example, 0 means FHDM has no victim buffer and 1 means the victim buffer can hold only one cache block. As shown in Figure 5.7, most of the benchmarks performed with the best performance *without* a victim buffer, except that benchmark *MST* has the best performance when the victim buffer can accommodate three heap cache blocks. Victim buffer is designed for reliving the thrashing of heap objects, which is heap access pattern dependent. When an application rarely has such thrashing access pattern, the implementation complexity will add more overhead to heap data management. The programs without a victim buffer performs best means that, the performance degradation by extra instructions for victim buffer implementation is larger than the benefit obtaining from less number DMAs. One the other hand, when an application has lots of thrashing access pattern, the victim buffer can reduce the number of conflict misses and improve the performance.

5.4.3 Scalability of FHDM

In this section, we conducted another set of experiments to examine the scalability of FHDM. In the experiment, we use the identical program on different number of cores. Besides, we configured the

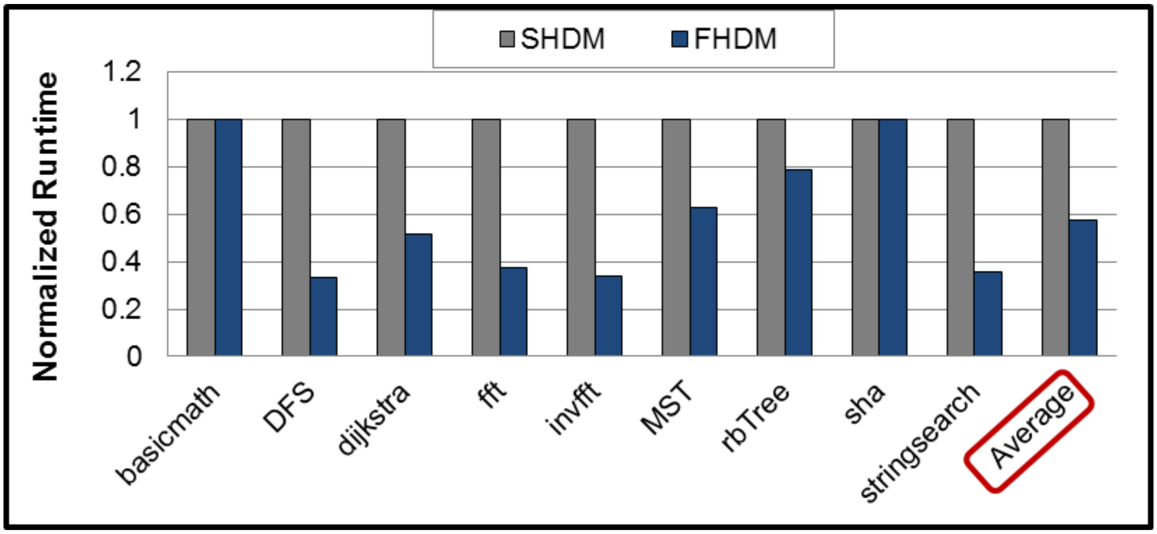


Figure 5.9: Runtime comparison between SHDM and FHDM

heap cache to i) use all available space in the local memory, ii) has no victim buffer, iii) utilize 4-way associate heap cache in library functions. The performance of the application with other number of cores is normalized to it with only one core. The results are presented in Figure 5.8. We can observe that the runtime increases gradually as the number of core increases. It is mostly due to the competition request of shared resource, e.g., DMA engine and mailbox. The increase for benchmark *fft* and *invfft* is more steep, since heap accesses in these two benchmarks are scattered and intensive, which cause frequent accesses to the main memory and therefore increase the memory latencies as the number of cores increases.

5.4.4 Thorough Comparison between SHDM and FHDM

5.4.4.1 Overall Comparison

In this experiment, heap cache is configured to use all the available space in the local memory. Besides, we used 4-way associative heap cache without a victim buffer for FHDM, as this configuration is found to be the best choice for most benchmarks in previous section. The results with both techniques are demonstrated in Figure 5.9. The x-axis in the figure indicates the nine benchmarks and the y-axis provides the execution time of the benchmark with FHDM that normalized to its runtime with SHDM [12, 13]. SHDM is a semi-automatic heap data management scheme with full associativity. We can observe that most of benchmarks have better performance with FHDM, with an average improvement

Table 5.2: Library code size of heap manager (in bytes)

	_malloc	_free	_g2l	_l2g
SHDM	3792	200	3360	968
FHDM	240	80	624	0

by 43%. There are two benchmarks, *basicmath* and *sha*, that have no improvement. The reason is that these two benchmarks have no heap data and therefore there is no heap data management with either scheme. For the other seven benchmarks, we can get improvement to different extent. There are two main reasons. First, SHDM implemented LRU (Least Recently Used) replacement for its heap cache. Performance of fully-associative heap cache is degraded by the sequential table walking to find the valid matching address for the heap data request. On the contrary, FHDM finds the corresponding set by hashing the global address of accessing heap object. In addition, FHDM uses low associativity for heap data management. Therefore, expensive table walking is avoided. Second, in order to maintain high associativity in SHDM, the Heap Management Table (HMT) needs to reserve one entry for each object. The entry stores the local address and global address of each heap object, the time stamp of the heap object, valid bit and modified bit. HMT increases as the number of heap objects increases. In order to support unlimited heap data, HMT must be managed between the main memory and the local memory, since the table itself can occupy all space in the local memory [12]. Due to the transfers of table entries, the performance can be even worse. The library function needs to fetch table entries from the main memory, walk through entries, put back entries, and fetch other entries again until getting the right entry for the least recently used heap object in the local memory. On the contrary, the HMT size in FHDM does not change with the increase of heap objects, as the mapping scheme is between HMT entries and heap blocks, rather than between HMT entries and heap objects in SHDM. In other words, the HMT in FHDM occupies constant space and can be fit into the local memory. Therefore, the overhead incurred by HMT data transfers is avoided.

5.4.4.2 Management Library Size

Since extra statement must be inserted before each heap data access no matter manually or automatically, the code size will be increased. We found that both techniques only increase less than 1% of code size because of this. In addition, as both schemes will link heap data management library with the original code of the application, the additional code overhead should be kept as small as possible. This is

Table 5.3: Dynamic instructions per function

	_malloc	_free	_g2l		_l2g
			hit	miss	
SHDM	948	50	280	373	243
FHDM	60	20	51	117	0

because library code locates in the local memory. If the library occupies more space, then the heap data will have less space in the local memory, because of which the performance will be further degraded. We removed *_l2g* function in FHDM by code transformation and therefore its size is 0 byte. In addition, as shown in Table 5.2, we also decreased the code size for the other three library functions. Overall, the library code size in SHDM is around 8KB, but FHDM has only 1KB static code size overhead.

5.4.4.3 Dynamic Extra Instructions and DMAs

In addition to static code overhead, both schemes incur dynamic runtime overhead. It must be noted that the application even cannot be executed in the local memory without two schemes. The total overhead consists of two components, the number of extra instructions and the number of data transfers (in terms of the number of DMAs) between the main memory and the local memory. To delve into the overhead of both heap data managements, we ran experiments and show results in Table 5.3. In the experiment, we ran all benchmarks with both schemes, and approximately calculated the average extra instructions incurred by each library function call. There are two columns for the function *_g2l*, *hit* and *miss*. *hit* means the accessing heap object is residing in the local memory when the function *_g2l* is called, while *miss* means the accessing heap object is not in the local memory when the function *_g2l* is called. In this circumstance, the function first writes back old data and then fetches the required data to the local memory by initiating a DMA command. Consequently, more instructions are needed. As shown in Table 5.3, FHDM has much less extra instructions per call than SHDM has.

5.5 Summary

In this chapter, we proposed an advanced scheme, fully-automatic heap data management (FHDM), for managing heap data on Software Managed Manycore (SMM) architectures. It scales well with the number of cores. It provides a compilation and runtime system (*_malloc*, *_free*, and *_g2l*) to manage unlimited size of heap data for SMM architectures. Its heap cache has low associativity. Compared to the state of the art SHDM, FHDM is fully transparent to programmers, supports multi-level heap pointers, and has better performance.

Our work foresees further possibilities of research in efficient data management. First, we can reduce the calls to *_g2l* function before each heap access if we can guarantee that the heap data is present in the local memory. Second, heap data can be prefetched if a heap access in the later stage can be predicted. Both directions requires a more advanced program analysis.

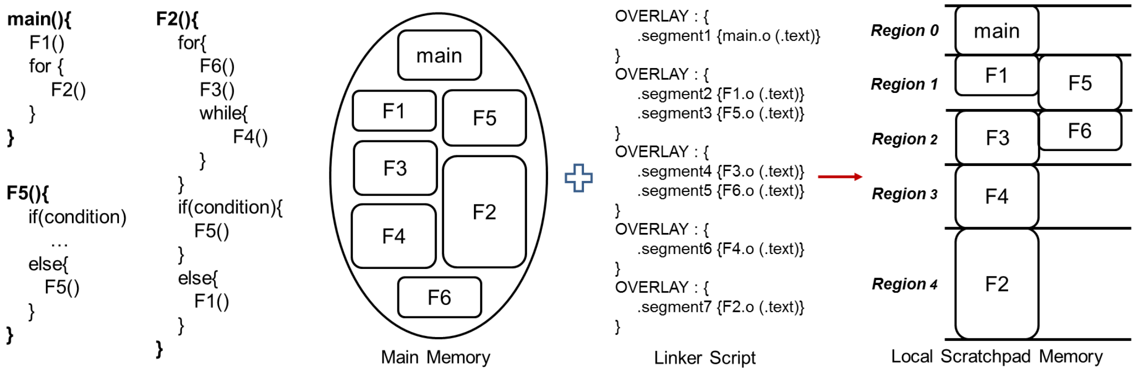
Chapter 6

EFFICIENT AND EFFECTIVE CODE MANAGEMENT

6.1 Introduction

On desktops or clusters with general purpose processing units, the system loads the complete program running on it into the main memory and then execute it. Even if the whole program is not loaded, most of instructions can be put into the memory, and instruction cache could automatically fetch the required ones when needed. This is the most efficient way to perform the operation. However, in case of Software Managed Manycore (SMM) architecture, the memory of the processing unit is limited, for example, each Synergistic Processing Element (SPE) on the IBM Cell processor has its own local memory of size 256KB. In this case, loading the complete program onto the local scratchpad memory before its execution usually does not work due to its memory constraints, unless the program is a small computation task which requires relatively low memory for both code and data. Even worse, SMM architectures lack of virtual memory facilities. To enable the execution of large applications on SMM architecture, it is necessary to use code overlay [36].

Code overlay is a programming technique that allows programs to be larger than the available memory. Besides, overlays may also be used for achieving performance improvement. As the local memory is shared by code and data of the mapped process, the size of data areas can be increased by constraining code into overlay area. Although there is performance loss by performing code overlay, data management may gain more because of larger memory resource. Usually, the overlay organization is generated manually by programmers or automatically by a specialized linker. A good code overlay requires deep understanding of the program structure, with the consideration of maximum memory savings and minimum performance degradation. The overlaid program objects are not loaded onto local scratchpad memory before the main program begins its execution. They actually reside in main memory until that object is required to be executed. Figure 6.1 shows one example of code overlay for SMM architectures. Functions mapped to the same region are located in the same physical address, and must replace each other during run time [36]. The size of a region is the size of the largest function mapped to the region. The total code space required is equal to the sum of the sizes of regions.



```
main(){
   F1()
   for {
      F2()
   }
}

F5(){
   if(condition)
      …
   else{
      F5()
   }
}
```

```
F2(){
   for{
      F6()
      F3()
      while{
         F4()
      }
   }
   if(condition){
     F5()
   }
   else{
     F1()
   }
}
```

```
OVERLAY : {
    .segment1 {main.o (.text)}
}
OVERLAY : {
    .segment2 {F1.o (.text)}
    .segment3 {F5.o (.text)}
}
OVERLAY : {
    .segment4 {F3.o (.text)}
    .segment5 {F6.o (.text)}
}
OVERLAY : {
    .segment6 {F4.o (.text)}
}
OVERLAY : {
    .segment7 {F2.o (.text)}
}
```

(a) Example code

(b) Code overlay

Figure 6.1: Code overlay on scratchpad memory - when task assigned to the execution core has larger footprint than the available space, code needs to be mapped between external shared main memory and the local scratchpad memory of the core.

6.2 Working of Code Overlay

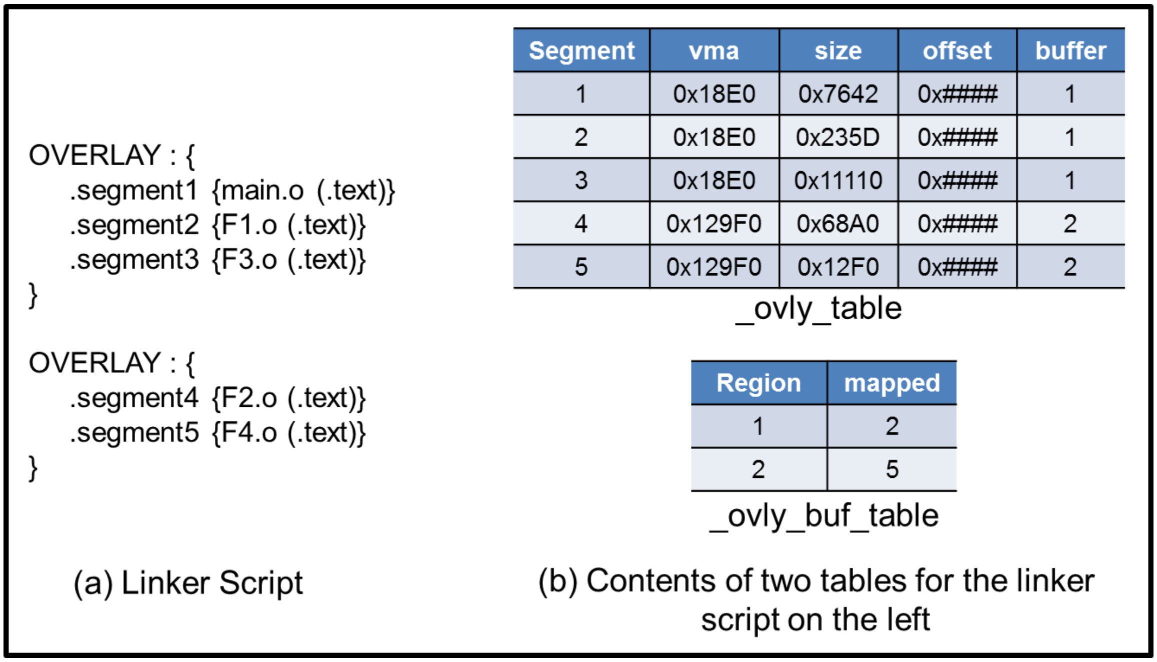
Code overlay is composed of two fundamental parts, overlay manager and the linker or one's own overlay scheme. The linker plays an important role of generating call stubs for all the regions and the associated management table, which has all the tags stored for the reference of the overlay manager. These stubs (one *__ovly_load()* for each function call) and tables (more details about the management table are present in the next paragraph), both are always reside in the local scratchpad memory. Instructions to call functions in the overlay regions are replaced by branches to these call stubs, which load the function code to be invoked, if necessary, and then branch to the function. When a particular function f is called by the currently executing function, overlay manager goes through the management table to check whether the instructions of f are already in the local memory. If they are already present, the program sequence jumps to the starting address of the target function and begins execution from there. On the other hand, if they are not present in the local memory, the instructions of f are loaded into the appropriate memory region, to its specific memory address during run-time, by performing special DMA operation. The DMA command is issued, controlled and executed by the overlay manager. In addition, the granularity of transfer unit is determined by specific code management schemes. They vary from one function object, one instruction word, to several function objects. The code management approach in this dissertation works at the granularity of one function object. The instruction fetching operation overwrites the existing instructions present in that region. Before jumping to the target address once the

Table 6.1: Elements in one entry of _ovly_table

Element	Information Stored
vma	local memory address that the section is loaded to
size	size of the overlay in bytes
offset	offset in executable where the section can be found
buffer	One-origin index into the _ovly_buf_table

Table 6.2: The element in the entry of _ovly_buf_table

Element	Information Stored
mapped	one-origin index into _ovly_table for the currently loaded overlay. 0 if none



```
OVERLAY : {
    .segment1 {main.o (.text)}
    .segment2 {F1.o (.text)}
    .segment3 {F3.o (.text)}
}

OVERLAY : {
    .segment4 {F2.o (.text)}
    .segment5 {F4.o (.text)}
}
```

(a) Linker Script

Segment	vma	size	offset	buffer
1	0x18E0	0x7642	0x####	1
2	0x18E0	0x235D	0x####	1
3	0x18E0	0x11110	0x####	1
4	0x129F0	0x68A0	0x####	2
5	0x129F0	0x12F0	0x####	2

_ovly_table

Region	mapped
1	2
2	5

_ovly_buf_table

(b) Contents of two tables for the linker script on the left

Figure 6.2: One example shows the status of two tables (_ovly_buf_table and _ovly_table) for code overlay.

code segment has been loaded, the overlay manager also checks to ensure successful completion of the DMA process to avoid any unwanted behavior in the program execution.

Two tables are present in the local memory, _ovly_buf_table and _ovly_table. The information for each entry of _ovly_table is shown in Table 6.1, and the same of _ovly_buf_table is present in Table 6.2. _ovly_table has only one entry per overlay segment. The overlay manager has only read permission for this table. This table should never change during execution of the program. _ovly_buf_table has only one entry per overlay region. The overlay manager has read-write permissions for this table. It changes to reflect the current overlay mapping state. Whenever the overlay manager loads a segment into a region, it updates the _ovly_buf_table with the corresponding segment number. Figure 6.2 shows contents of _ovly_table and _ovly_buf_table for a example linker script.[1] Column *vma* in _ovly_table indicates the local address where corresponding segment is loaded. Column *buf* shows to which region corresponding segment is mapped. Column *mapped* in _ovly_buf_table shows which segment is currently located in the region.

6.3 Objective of Code Overlay

For code overlay to work best, there are two fundamentals need to be determined: first is to choose the number of regions, and the second is to map all functions to regions. From the performance perspective, it is best to place each function into a separate region, so that it will not interfere with any other objects, but that may increase the code space too much. In contrast, mapping all functions into one region uses the minimum amount of code space, while incurs too many instruction transfers and therefore runtime overhead. *The task of optimizing code overlay is, to organize the application functions into regions that will obtain a balance between the code space used and the data transfers required.*

6.4 Related Work

Scratchpad memory has been well known for a decade in the embedded area. Since it sheds hardware required for cache management to enable performance and silicon area advantages over the system cache, all code and data management must rely on compiler or programmer's hand inserted code [18]. There are a number of approaches for selecting what to place into the scratchpad memory and when to place them there. Steinke et al. [17] view the instruction placement problem in terms of minimizing memory accesses, and evaluate the structure of the program in the granularity of basic blocks and functions to formulate an ILP problem. Udayakumaran et al. [64] present an algorithm which looks at timestamps in code sections to determine temporal locality, The work [7, 38] present algorithms which require trace data. Egger et al. [25] implements a paged SPM management and prefetching scheme. These schemes require profiling information which is impractical as program execution varies widely on different input data when there are branches. Besides, these techniques cannot be directly applied to

[1] The linker script is used to control the generation of overlays as this allows maximum flexibility in specifying overlay regions and in mapping functions to overlay regions.

Software Managed Multicore (SMM) architecture. This is because of the difference in the way scratchpad memory has been traditionally used, and the way it is used in SMM architectures. In embedded systems, e.g., ARM processors, the scratchpad memory is present in addition to the regular cache hierarchy of the processor. Programs can be executed correctly without using scratchpad memory, but scratchpad memory can be used to optimize performance and power efficiency. On the contrary, the scratchpad memory is the only memory hierarchy in SMM architecture and is therefore essential, rather than optional. All code and data must go through it. As a result, while the problem of using scratchpad memory in embedded systems is that of optimization, the problem of using scratchpad memory in SMM architectures is to enable the execution of applications.

To the best of our knowledge, work [16, 39, 40, 54] are similar to our effort for code and [40] is the most similar one. Two mapping algorithms were proposed in [40]. One is function mapping by updating and merging (FMUM) and the other one is function mapping by updating and partitioning (FMUP). FMUM begins with a mapping in which each function is placed in a separate region. It repeatedly selects and merges a pair of regions with the minimal merge cost among all pairs of regions until all functions can fit in the given scratchpad memory size. In contrast, FMUP starts with a mapping where all functions are placed in only one memory region. It repeatedly selects the function which maximally decreases the cost and places it to another region until the size of the total amount of instruction space is less than the given memory size.

6.5 Cost Calculation

As mentioned in Section 6.1 and Section 6.2, when two functions are mapped into one region, they would swap each other during the execution time, which therefore lead to performance degradation. To develop any code overlay mapping, there is a need to estimate this swap cost. Cost is an estimation for mapping algorithm to determine the funtions-to-regions relationship. In this dissertation, the number of bytes that will be transferred between main memory and the local scratchpad memory is used as a metric to measure this cost. Proposing a correct and comprehensive cost calculation is of utmost importance, as it is the foundation upon which any mapping algorithm can be developed.

6.5.1 Motivation

The work [16, 39, 54] provide several different heuristics for code overlay mapping on SMM architectures. However, they are all not efficient enough, which is mainly because of inaccurate or incorrect cost calculation. They statically calculate the mapping cost and generate a mapping. They never dynamically update the cost during the course of mapping algorithm. This is unsatisfactory and can lead to inferior mapping. Figure 6.3 (a) shows an example where function *main* calls F1, F1 calls F2, and F2 calls F3, and then they all return. The function nodes also indicate the sizes of functions. Let us consider a case which requires us to map all functions into a scratchpad memory of 5 KB. It is slightly tricky to calculate the cost between indirect function calls. For example, when computing the cost between *main* and F2, if *main* and F2 are mapped to the same region, the interference[2] between them depends on where F1 is mapped. If F1 is mapped to another different region, then the interference between *main* and F2 is just sum of their sizes, namely, 3 KB + 1 KB = 4 KB. The calculation is as follows. When F2 is called, 1 KB of function F2 will need to be brought into the memory. When the calling state returns to *main*, 3 KB of the code of *main* needs to be brought into the scratchpad. However, if all *main*, F1 and F2 are mapped to the same region, then the interference cost between *main* and F2 is 0. This is because, when F2 is called, *main* is already replaced with F1, and when the program returns to *main*, F2 is already replaced. In a sense, there is interference between *main* and F1, and between F1 and F2, but there is no interference between *main* and F2.

Previous approaches [16, 39, 54] computed the worst case interference cost, i.e., 4 KB for *main* - F2, and never updated it, and therefore obtained inferior mapping. To explain this, Figure 6.3 (b) shows a state in mapping when *main*, F1 and F2 have already been mapped. *main* is alone in region 0, F1 and F2 are together in the region 1. Now is the time to map function F3. Size of F3 is 0.5 KB, therefore it can be mapped to either region, without violating the size constraint. The interference cost between region 0 and F3, i.e., between *main* and F3 is 3.5 KB. The interference cost between region 1 and F3 is traditionally computed as the sum of interferences between the functions in region 1 and F3, i.e., 2.5 KB between F1 and F3, and 1.5 between F2 and F3, totalling to 4 KB. Consequently traditional techniques will map F3 to region 0 with *main* (shown in Figure 6.3 (c)). Clearly there is a discrepancy in computing the interference cost between region 1 and function F3. If F2 is also mapped to the same region, the

[2] The interference here means the two functions mapped to the same region will replace each other during execution time. We use the amount of data transfer to estimate this interference cost.

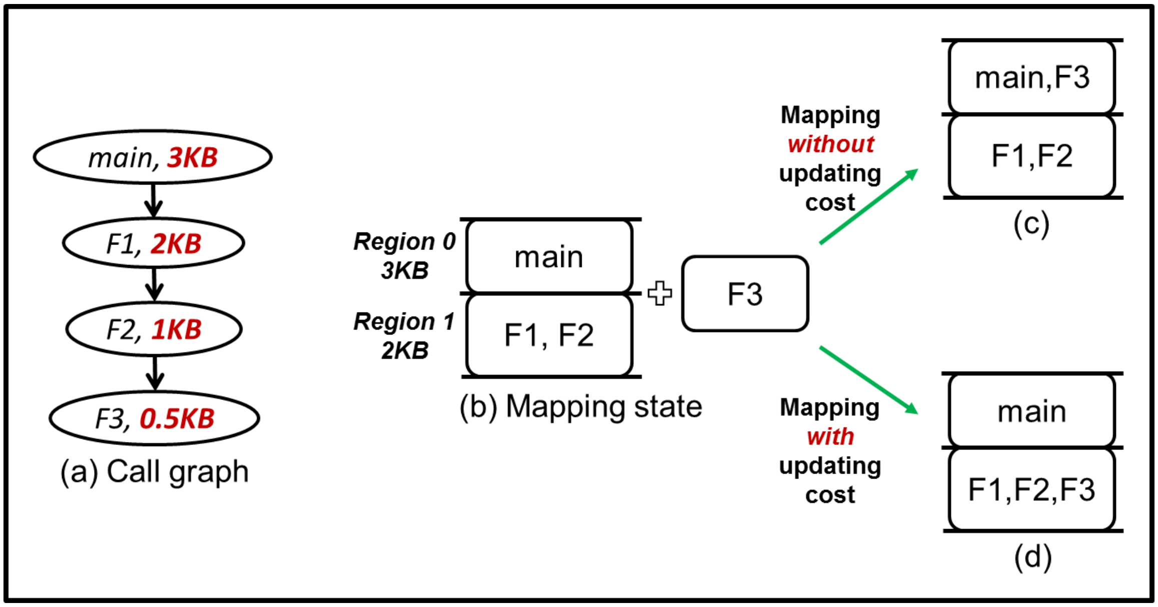


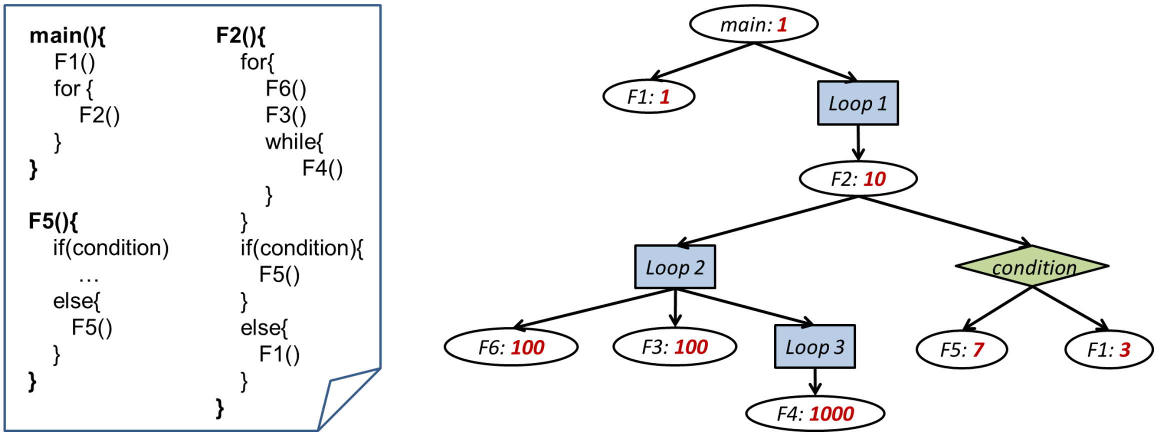
Figure 6.3: Cost between functions depends on where other functions are mapped, and updating the costs as we map the functions can lead to a better mapping.

interference cost between F1 and F3 should be estimated as 0. Otherwise, the interference cost between region 1 and function F3 are incorrectly (over)estimated. With this fixed, the interference between region 1 and F3 is just the interference between F2 and F3, which is just 1.5 KB. As per this correct interference calculation, F3 should be mapped to region 1 (shown in Figure 6.3 (d)). The required total data transfer between main memory and the local memory, in this case $9.5 = 3+(2+1+0.5+1+2)$ KB, as compared to $11.5 = (3 + 0.5 + 3) + (2 + 1 + 2)$ KB with the previous mapping, resulting in a 18% savings in data transfers.

In next sections, the limitation aforementioned is addressed by deploying a graphical code representation (Section 6.5.2), and proposing a cost calculation algorithm (6.5.3).

6.5.2 Graphical Code Representation

Calculating the management cost correctly and mapping code efficiently requires 1) the deep understanding the structure of the input application, and 2) representing the flow information and control information in an effective format. This information can be built into an enhanced Control Flow Graph (CFG) known as Global Call Control Flow Graph (GCCFG) presented by Pabalkar et al. [54].



```
main(){               F2(){
  F1()                  for{
  for {                   F6()
    F2()                  F3()
  }                       while{
}                              F4()
                          }
F5(){                   }
  if(condition)         if(condition){
    …                     F5()
  else{                 }
    F5()                else{
  }                       F1()
}                       }
                      }
```

(a) Example code

(b) GCCFG

Figure 6.4: The GCCFG for the code in Figure 6.1.

6.5.2.1 Definition of GCCFG

Definition 3 *(Global Call Control Flow Graph). A global call control flow graph (V, E) is an ordered acyclic directed graph, where $V = V_F \bigcup V_L \bigcup V_C$. Each node $v_f \in V_F$ with a weight w_f on it represents a function or F-node, $v_l \in V_L$ denotes a loop or L-node, $v_c \in V_C$ represents a conditional or C-node. w_f is the number of times function f is invoked in the program. An edge e_{ij} ($e_{ij} \in E$) shows a directed edge between F-nodes, L-nodes and C-nodes.*

If v_i and v_j are functions, then the edge represents a function call. If v_j is an L-node or a C-node then it represents control flow. If v_i is a C-node, then the edge represents one possible path of execution. If v_i is a loop, then the edge represents what is being executed in the body of the loop. If v_j is a loop and its ancestor is a loop then the edge represents a nested loop execution. The edges are ordered, edges to the left execute before edges to the right, except in the case of condition nodes. Edges leaving condition nodes can execute their *true* or *false* children, where all true children are ordered and all false children are ordered.

As an example, Figure 6.4 (b) shows the GCCFG of the program present in Figure 6.4 (a). We ignore direct recursive function calls F_5 in the graph. Since we are concerned with cost between different functions, the effect of a direct recursive call is that the code necessary to run the called function is already in memory, resulting in no instruction transfers.

6.5.2.2 Construction of GCCFG

In this section, the complete algorithm to construct the GCCFG of an application is presented. The input of the algorithm is all CFGs in the program. Then all the CFGs are integrated into a GCCFG in two steps.

Step one – basic blocks are scanned for the presence of loops (back edges in a dominator tree), conditional statements (fork and join points) and function calls (branch and link instructions). The basic blocks containing a loop header are labeled as *loop nodes*, those containing a fork point are labeled as *conditional nodes* and the ones containing a function call are labeled as *function nodes*. If a function is called inside a loop, the corresponding *function node* is joined to the loop header *loop node* with an edge. If any *loop node* representing nested loops exist, they are also joined. *Function nodes* not inside any loop are joined to the first node of the CFG. The first node, *function nodes*, *loop nodes* and corresponding edges are retained, while all other nodes and edges are removed. Essentially this step trims the CFG, while retaining the control flow and call flow information.

Step two – all CFGs are merged by combining each *function node* with the first node of the corresponding CFG. The merge ensures that strict ordering is maintained between the CFGs, i.e., if two functions are called one after another, the first function is a left child and the other function is a right child of the caller function. One thing needs to be mentioned herein is that we conservatively expand indirect function calls invoked through function pointers in much the same way as they were called with equal probability outside of any conditional node.

The weight assignments for *function nodes* usually have two ways, profiling or static estimation. Profiling method is straightforward, as the exact number of times the loop to be executed can be determined by executing the program with its input. For example, we could get the number of iterations of a *while* loop with an input dependent condition. The static compile-time weight assignment scheme is tricky but important, as it removes the expensive and prohibitive task of profiling. Our results show the estimation of weight will not degrade too much performance. Here, we present our methodology for estimating the number of function calls on each function node. The basic blocks of the managed application are first scanned for the presence of loops (back edges in a dominator tree), conditional statements (fork and join points) and function calls (branch and link instructions). After capturing these information, we assign the weights on the functions by traversing GCCFG in a top-down fashion. Initially, they are assigned to 1. When a *loop node* is encountered, the weight on all its descendant function nodes equals the weight of *loop node*'s nearest ascendant *function node* in the path multiplying a fixed constant, *loop factor* Q. This ensures that a function which is called inside a deeply nested loop will receive a greater weight than other functions. When a *conditional node* is encountered, the weight on each descendant function node equals to the weight of *conditional node*'s nearest parent *function node* multiplying the branch probability of each edge diverging from the *conditional node*. We adopted a traditional scheme described by Smith [59] to predict the branch probability. We found the impact of Q is negligible as long as it is larger than 1 (details is shown in Section 6.7.3). As a result, Q is chosen to be 10 in this dissertation. The previous Figure 6.4 is the resulted GCCFG of the example code with our static weight assignment scheme.

6.5.3 Cost Calculation Algorithm

Since the cost calculation is very frequently required by CMSM, making interference cost calculation as fast as possible is important. Given a GCCFG, and a mapping M, a naive way to compute interference cost can be done by traversing the GCCFG, (much like simulation) and adding the function sizes, as we visit function nodes. However, the complexity of this will be very high. Therefore, we develop an algorithm to compute the interference cost using just two Depth First Search (DFS) traversals of the GCCFG. If two functions are mapped into the same region, and one function is called after another during the execution, two functions have to swap each other on the SPM, and it is said that two functions are interfered by each other [54, 40]. However, the interference between such functions depends upon mappings of other functions in-between during the execution. Therefore, it is essential to capture the interferences changes between such functions and compare the cost of interference to create a better code placement which reduces interferences between functions in regions.

Algorithm 4 shows the procedure to capture the *interference cost* between two functions. As outlined in Algorithm 4, we calculate the interference cost between functions as we traverse the GCCFG in Depth-First Search order including function return. First, it starts from the initial node of GCCFG (line 1) and search for v_1 as the GCCFG is traversed. After finding v_1, we assign the first edge weight (line 13) between v_1 and the next node. If the next node is a loop node, it keeps traversing the GCCFG until it meets a function node, and then it assigns the first edge weight (lines 15-17). However, if there exists a function which is mapped into the same region as v_1 and v_2 after v_1 is found and before v_2 is found, the edge weight becomes 0 since there is no interference between v_1 and v_2 (lines 4-5). When there is LCA (or least-common-ancestor) of v_1 and v_2 after v_1 is found, the first edge weight is re-assigned (lines 7-8).

Algorithm 4 Algorithm cost (GCCFG, v_1,v_2)

```
$v_{current} = v_{initial}$
while $v_{current} \neq v_{final}$ do
    if $v_1$ is found and $v_2$ is not found then
        if $M(v_{current})$==$v_1$ or $M(v_{current})$==$v_2$ then
            reset all weights
        else
            if $v_{current}$ is $LCA(v_1, v_2)$ then
                assign $weight_1$
            end if
        end if
    end if
    if $v_{current} == v_1$ then
        assign $weight_1$
    end if
    if $v_{current}.nextNode == loopNode$ then
        find next function node, then assign weight
    end if
    if $v_1$ found && $v_2$ found then
        assign $weight_2$
        $totalWeight$+=$min(weight_1, weight_2)$
    end if
    $v_{current} = v_{current}.nextNode()$
end while
return $totalWeight$
```

When v_2 is found after v_1 is found while it is traversing the GCCFG, we assign the second edge weight and add the minimum of edge *weight1* and *weight2* to consider the case where there exists a function mapped in the same region or an LCA between v_1 and v_2 in the execution sequence. As the final interference counts between those two functions, we calculate interference count again with switched order of two functions and take maximum value of two calculations. This is because it is unknown which function comes first during the execution. For the final interference cost, the cost calculation function is given by the sum of two functions multiplied by the final interference count. This procedure visits each node in the GCCFG only once, therefore, the runtime complexity of interference cost calculation is $O(V_f)$.

6.6 Heuristic Approach for Code Overlay

Finding the number of regions and mapping the functions to regions that will minimize the total amount of instruction transfer, both have been proven to be intractable [54, 65]. Consequently, we propose a greedy algorithm for code overlay. Algorithm 5 outlines our CMSM heuristic. It starts with a mapping, in

Algorithm 5 Algorithm CMSM (GCCFG, $\mathcal{S}$)

```
 1: SPMregions {set of $N$ regions in the scratchpad memory}   ▷ $N$ is the number of functions in the program
 2: $R_{dest} \leftarrow 0$, $R_{src} \leftarrow 0$;
 3: while SPMSize() $> \mathcal{S}$ do
 4:     FindMinBalancedMerge($R_{dest}$, $R_{src}$, GCCFG);
 5:     MergeRegions($R_{dest}$, $R_{src}$);
 6:     SPMregions.erase($R_{src}$);
 7: end while
 8:
 9: procedure FindMinBalancedMerge (&$R_{dest}$, &$R_{src}$, GCCFG)
10: begin procedure
11: minMergeCost $\leftarrow$ DBL_MAX, tmpCost $\leftarrow$ 0;
12: for all combination of regions $R_1, R_2 \in$ SPMregions  do
13:     size1 $\leftarrow$ RegionSize($R_1$), size2 $\leftarrow$ RegionSize($R_2$);
14:     max $\leftarrow$ max(size1, size2), min $\leftarrow$ min(size1, size2);
15:     tmpCost = cost($R_1, R_2$, GCCFG) * $\frac{max-min}{(max+min)^2}$;
16:     if tmpCost $<$ minMergeCost then
17:         minMergeCost = tmpCost;
18:         $R_{dest}$ = $R_1$;
19:         $R_{src}$ = $R_2$;
20:     end if
21: end for
22: end procedure
```

which each function is mapped to a separate region (line 1). Now all combinations of two regions are tried to be merged until the total space meets memory constraints (while loop, lines 3-7). To do this, we firstly find two "balanced" regions with minimal merge cost through function FindMinBalancedMerge() in line 4. We then merge two regions and update the region information in the set SPMregions (line 5-6). Function FindMinBalancedMerge() is described in Algorithm 5. To do this, we choose a region pair (R_1, R_2) (Algorithm 5, line 12-21), and calculate its merge cost in line 15. Here, we utilize the cost function from Algorithm 4. Besides, there is a balance factor $\frac{max-min}{(max+min)^2}$. It is inclined to place the functions having close object sizes into the same region. It is important, since we can compress the total code space in the local scratchpad memory and use less memory. This remaining space could result in more number of regions as long as there are functions that could be accommodated to it. Even if no more regions would be generated, it is still beneficial to use less space to achieve competitive performance. As stated before, the local scratchpad memory is shared among global data, stack data, heap data and instructions of the managed program, less space consumed by instructions indicates more space for other data that could eventually results in better performance.

Table 6.3: Benchmarks, their minimum sizes of code space, and maximum sizes of code space.

Benchmark	functions	min code (B)	max code (B)
Adpcm_decoding	13	1552	6864
Adpcm_encoding	13	1568	6880
BasicMath	20	4272	12128
Dijkstra	26	2496	9216
FFT	27	2496	12776
FFT_inverse	27	2496	12776
String_Search	17	632	4708
Susan_Edges	24	19356	37428
Susan_Smoothing	24	19356	37428

Complexity. The *while loop* in line 3 in Algorithm 5 merges two regions at a time. Since in the worst case, all regions might have to be merged into one, this loop can execute $|V_f|$ times. Inside this, the for loop (lines 12-21 in Algorithm 5) runs for each pair of regions. This adds $O(|V_f|^2)$ complexity to the time. Inside the loop, there is a cost calculation which has complexity $O(|V|)$. Thus the worst case timing complexity of CMSM is $O(|V_f|^4)$.

6.7 Experimental Results

6.7.1 Experimental Setup

We use IBM Cell processor [27] as our hardware platform. It is a multicore processor, and gives us accesses to 6 of the 8 Synergistic Processing Elements (SPEs). In addition, this architecture has a shared main memory on main core, and only a local scratchpad memory on each execution core or SPE. Scratchpad memory is limited, and therefore the program needs to be managed in software when its footprint is larger than memory available.

The benchmarks used for experimentation are from Mibench suite [31] and presented in Table 6.3. All those information is profiled by compiling programs for SPE. *functions* is the total number of functions in the program, including library functions tailored for SPE. *min code* is the smallest possible mapping size of code space, defined by the size of the largest function in the application. *max code* is the total size of the program. We utilize main core and only 1 SPE available in the IBM Cell BE in most of our experiments, except the one designed for demonstrating scalability of our heuristics in Section 6.7.4.

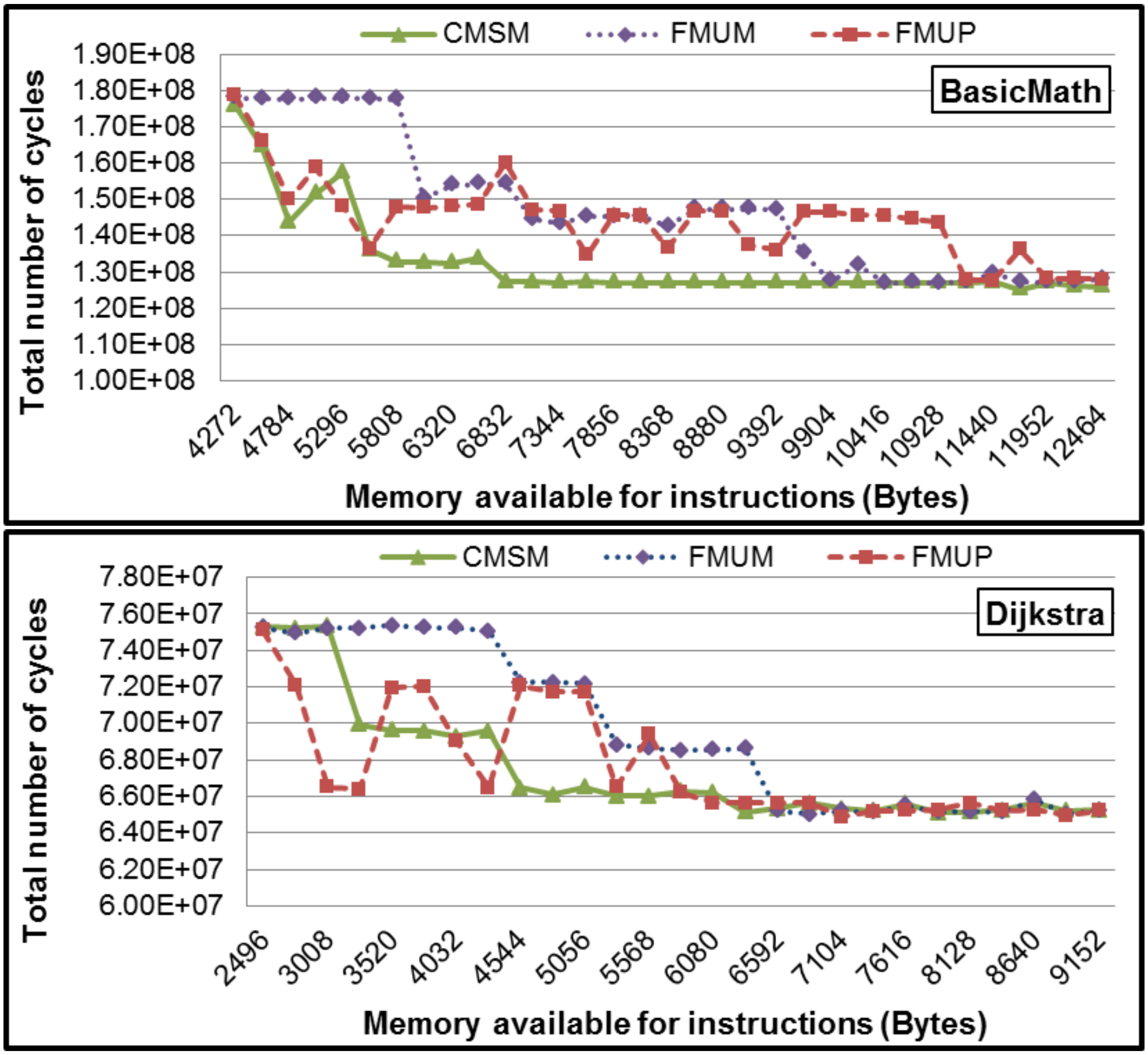


Figure 6.5: Performance comparison against FMUM and FMUP

6.7.2 Overall Performance Comparison

While the conclusion scale for all benchmarks, Figure 6.5 shows the execution time of the binary compiled using each heuristic for only two representative applications. The X-axis shows a wide range from *min code* to *max code* of each program, with the step size 256 bytes. As observed from the figure, when the code space is very tight, all heuristics achieve the same mapping, i.e., mapping all the functions in one region. However, as we start relaxing the code size constraint, CMSM typically performs better than FMUM and FMUP. Our CMSM is inclined to place two functions with small merge cost and similar code size in one region at each step of merging. It is achieved by using a “balance" factor described in our algorithm. The benefit of doing so is to increase the number of regions in the code space. We expect mapping solutions with more regions to give lower overhead costs, as only functions mapped to the same region will swap each other during run time. The reverse effect is also visible. When the code size constraint is extremely relaxed, for example, larger than 70% of *max code* present in Table 6.3, all three algorithms again achieve very similar code mapping. This is because there are quite few functions mapped to one region when the code space is sufficient enough. The small differences in code mapping generate negligible effect on performance.

Note that code mappings created by the CMSM do not always outperform the other two heuristics. For example, when memory available for instructions of benchmark “dijkstra" is 3520 bytes in Figure 6.5, CMSM is worse than FMUP. This is because FMUP has to do very few steps, while CMSM needs to do a lot of merges. The more steps a heuristic has to take, the errors in each step accumulate, and eventually a worse mapping might be generated. Although our heuristic does not consistently gives good results, it gives better results most of the times. We tested the three heuristics for all code size constraints from minimum to maximum. On average over all benchmarks, CMSM gives a better result than other two algorithms 89% of time. Another important observation from Figure 6.5 is that, applications are tend to have less execution time when their code space become larger. A large code space usually leads to more number of regions in it, and therefore less functions overlap each other in regions. This explains the trade-off between the performance and the memory available for instructions.

6.7.3 Accuracy of Weight Assignment

We examined the goodness of our static weight assignment on function nodes of GCCFGs of nine applications. We compared the execution time of each benchmark using static assignment with its execution time using profile-based assignment. Averagely, we found both schemes achieve similar performance for the set of benchmarks. This implies that we can eliminate the compile time overhead to obtain profiling information through the loop based function weight assignment. It also makes the code management technique more comprehensive, since profiling large applications is time-consuming and intimidating.

6.7.4 Scalability

Figure 6.6 shows the results we examined the scalability of our CMSM heuristic. We normalized the execution time of each benchmark with number of SPEs to its execution time with only one SPE, and

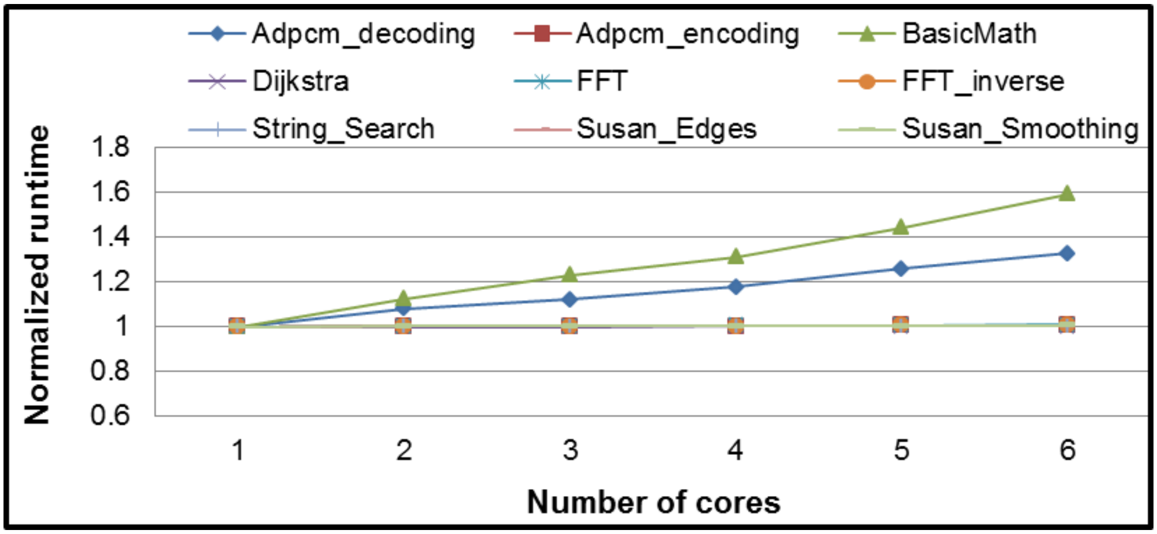


Figure 6.6: Scalability of CMSM on multicore processors

show them on y-axis. In this experiment, we executed the identical application on different number of cores. According to the graph, the runtime difference with the increased number of SPEs is negligible even in such aggressive configuration. In this configuration, DMA transfer occur almost at the same time when instructions need to be moved between the global memory and the local memory. This will make the Elemental Interconnect Bus (EIB) saturated. Benchmark *BasicMath* increases most steeply, as there are many instruction transfers in the program, which makes each SPE have more execution time.

6.8 Summary

Software Managed Multicore (SMM) architectures are one of promising solutions to the problem of scaling the memory hierarchy. However, since scratchpad memory cannot always accommodate the whole task mapped to it, certain schemes are required to mange code, global data, stack data and heap data of the program to enable its execution. This chapter present a framework to manage code between main memory and the local memory, at the granularity of function object. We addressed the cost estimation problems in previous work by devising a correct cost calculation algorithm for code mapping. Since code mapping problem has been proved to be NP-complete, a heuristic approach named CMSM is proposed for the same problem. Our experimental results show that CMSM generates code mapping which leads to significant performance improvement compared to previous work.

Chapter 7

MAIN MEMORY MANAGEMENT

There is a potential of memory overflow in the main memory of data managements. The data management, namely SSDM and FHDM, allocate a large space at the start of the program in the main memory to accommodate all stack/heap data of the execution cores. If enough space is allocated, then this management can be performed very efficiently, by just maintaining a pointer to the start of free space in the main memory. The execution core can then just perform a DMA of function frames or heap data to the main memory. Further, since the execution core knows the size of function frames and heap data, it can update the pointer to free space by itself. Again, this scheme will work in extremely embedded contexts, where the maximum stack/heap space required by applications can be known, but is impossible in general due to recursive functions and dynamic characteristic of heap data. For recursive functions, the stack space required may be unbounded. Heap data is also the same. In other words, no amount of initial memory allocation in the main memory may be enough. Consequently, when the pre-allocated main memory is filled up, any further DMAs can write into the address space of other execution cores, causing an access fault in the best case and wrong results in the worst case. In such a circumstance, we need to develop a scheme to support unbounded stack data and heap data.

In general case, stack data and heap data in the main memory must be managed dynamically. This implies that at some time, the execution core must request the main core to allocate more memory. Since this cannot be done by a DMA call, and therefore some other communication mechanism between the execution core and the main core must be used. For example, in the IBM Cell processor, we can use the mailbox facility for this purpose. Additionally, we implement a new thread on the main core that will continuously listen to requests from the execution core, and allocate memory when requested. Then it sends the start addresses of the allocated space to the execution core. This is done so that in most cases, the address translation can be done in the execution core, and only a direct DMA will be needed.

On the execution core, this functionality is implemented in *_sstore* function for stack management, and *_malloc* function for heap management. *_sstore* and *_malloc* first check if there is space for the incoming function stack or heap data on the local memory. If not, the oldest function frames/heap objects should be evicted to the main memory. Figure 7.1 shows the whole process that are needed to manage a memory request. Before eviction, *_sstore* or *_malloc* check whether more memory is needed

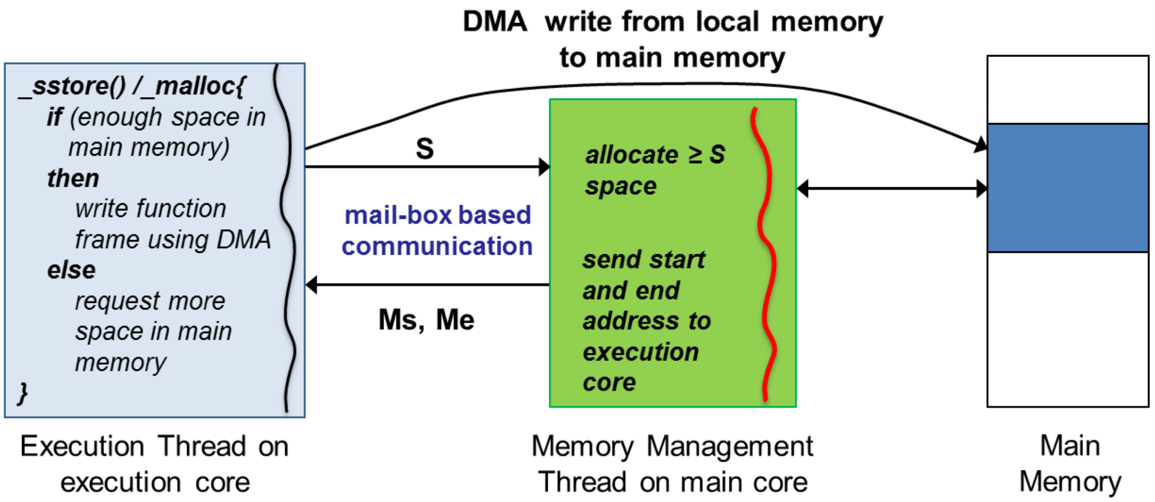

DMA write from local memory to main memory
_sstore() /_malloc{
if (enough space in main memory)
then
write function frame using DMA
else
request more space in main memory
}
S
mail-box based communication
Ms, Me
allocate ≥ S space
send start and end address to execution core
Execution Thread on execution core
Memory Management Thread on main core
Main Memory

Figure 7.1: Typically the execution core can write in the main memory using only a DMA call. However, when there is no more space in the main memory, a request is made to the main core to allocate more memory. After allocation, the main memory management thread sends the memory start and end addresses Ms, Me to the execution core.

in the main memory. We track the remaining space in the main memory by variables *Ms* and *Me* in the execution core. If not, it sends a request via the mailbox to the main core. The memory management thread on the main core accepts this request, allocates more memory (e.g., two times) than the request, and finally sends the start and end address of the newly allocated memory to the execution core, which can then be used for future data management. The reason we allocate more memory is that we can reduce the number of communication calls by not allocating memory each time function stack frame/heap data is evicted. The functionality of _*sload* and _*free* are very similar, except that if all function frames or heap data from a memory region have been brought back to the local memory, the memory is free-ed.

Instead of adding the main memory management functionality in the existing thread of the main core, keeping this as a separate thread has several advantages. First is that the code of the main thread does not need to be modified, and the extra threads can be supplied as a part of the library, and the user just needs to compile their applications with it. Second, this separate thread solution scales with the number of cores, as just one thread will be able to manage memory requirements of all execution threads on the processor. Since memory allocation is managed by the operating system on the main core, the dynamically allocated buffers never infringe each other's space.

Chapter 8

MEMORY PARTITION

Till now, we have developed techniques to manage and optimize the management of each kind of data (and code) in a constant amount of space in the local scratchpad memory, thereby enabling execution of even those tasks whose memory footprint is much larger than the local memory available [10, 11, 12, 13, 14, 15, 40, 49].. Even though execution is possible, but the overhead of the management is strongly correlated to the space provided to the data (shown in Figure 8.1). For example, more space for stack data will result in lesser stack frames being evicted to the global memory, and therefore the stack management overhead will be less. Therefore an important question is, how to partition the local scratchpad memory among the different data-kinds so as to maximize performance. While in general this is a difficult question, the data requirements of some embedded systems are predictable, and the compilation time is not as important, since the application is compiled once, and it is executed forever. In such systems, the memory partitioning problem can be tackled.

Memory partitioning is very important, as it has a very significant impact on performance, but even in embedded systems, in order to find the optimal memory partition, we must execute the application for all partitions. This simulation process is time-consuming. Let c_{min} and c_{max} be the minimum and maximum size of code region, where c_{min} is the largest function object size. s_{min} and s_{max} be the minimum and maximum size of stack region, where s_{min} is the largest function stack size. In addition, h_{min} and h_{max} can be used to denote the minimum and maximum size of heap region. If the scale unit for code, stack and heap data are l_c, l_s and l_h respectively, the total number of simulations required is:

$$\frac{c_{max} - c_{min}}{l_c} \times \frac{s_{max} - s_{min}}{l_s} \times \frac{h_{max} - h_{min}}{l_h}$$

The question is how to obtain a good memory partition without running so many simulations. To do this, we assume that the overhead of managing a data-kind is not correlated to the other space

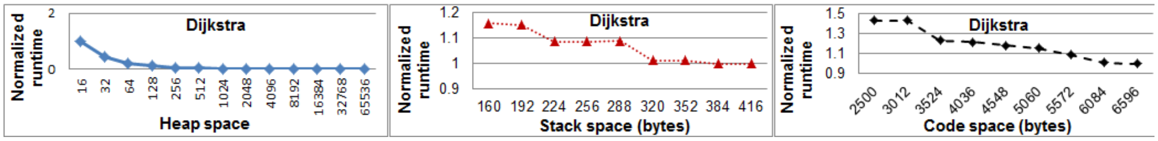


Figure 8.1: There is a space and performance trade-off. The more space we allow for a data, the lower is the management overhead. We intend to estimate the slopes of these curves to decide the partitioning of the local memory space for different data.

allocation of the other data-kinds, and is only dependent on the space allocated to that data-kind. By doing this, we only need to find out how performance depends on the space allocated for that data. We keep the size of the allocation for the other two data-kinds at minimum, and vary the allocation of the data-kind in question, and observe the performance. Exploration results (in Figure 8.1) show that the space-performance dependency can be relatively accurately approximated as 2-degree polynomials. Let C, H and S be code size, heap size and stack size respectively. Three 2-degree polynomials can be represented as:

$$R(C) = m_1C^2 + m_2C + m_3$$
$$R(H) = n_1H^2 + n_2H + n_3$$
$$R(S) = q_1S^2 + q_2S + q_3$$

where m_i, n_i and q_i ($i = 1, 2, 3$) are coefficients and R means the runtime.

Using these three quadratic dependencies, we formulate the memory partitioning problem as follows:

$$\begin{aligned} min \quad & m_1C^2 + m_2C + m_3 + n_1H^2 + n_2H + n_3 + q_1S^2 + q_2S + q_3 \\ s.t. \quad & c_{min} \le C \le c_{max} \\ & h_{min} \le H \le h_{max} \\ & s_{min} \le S \le s_{max} \\ & C + H + S = TS \end{aligned}$$

where TS is the available memory resource that can be used by three data.

Now, we only need to change the size for each data from its minimum to its maximum to get coefficients in each polynomial curve. Consequently, the number of simulation time is reduced to:

$$\frac{c_{max} - c_{min}}{l_c} + \frac{s_{max} - s_{min}}{l_s} + \frac{h_{max} - h_{min}}{l_h}$$

To demonstrate our proposed approach, we run all benchmarks from previous chapters on the IBM Cell processor [27]. Each application uses the whole space in the local scratchpad memory. Besides, stack data is managed with smart stack data management (SSDM) scheme; Code is handled with CMSM heuristic; Heap data is managed through fully-automatic heap data management (FHDM). The size for each data varies from its minimum to its maximum, in steps of 128 bytes.

The best performance of the managed program can be achieved with exhaustive simulations, yet this may take intolerant time. On the other hand, our scheme reduces the simulation time with a slight

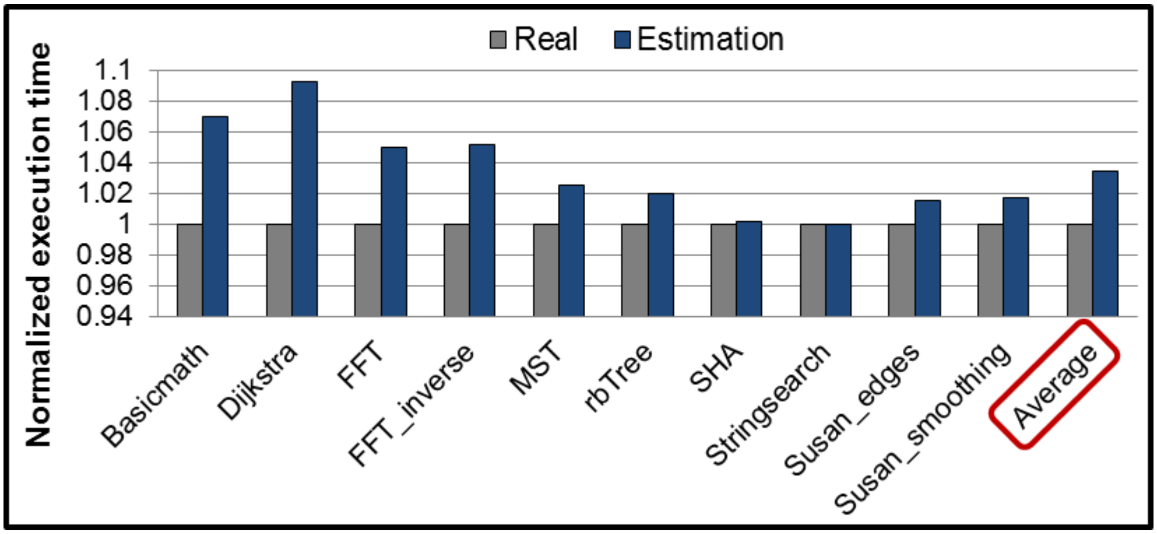


Figure 8.2: Using our heuristic, the simulation time is reduced to 19% of the exhaustive scheme, but the runtime is only increased by an average of 3%.

performance degradation. As shown in Figure 8.2, the simulation time is reduced to 19% of the time on exhaustive simulations. With the memory partition generated from our scheme, the application performs only 3% worse than the best performing partition obtained from exhaustive simulations. This result demonstrates that our scheme greatly reduces the simulation time, yet not losing too much performance.

Chapter 9

PUTTING IT ALL TOGETHER

9.1 Supporting Limitless Stack Data, Heap Data, and Code

In this section, we demonstrate the applicability of our stack data, heap data and code management [10, 11, 12, 13, 14, 15, 40, 49]. Though the result scales for all applications in previous sections, we only show the application *rbTree*. *rbTree* dynamically allocates space for each node and creates a red black tree. Finally, it prints out all nodes in the tree. To demonstrate our management techniques, we change the number of nodes in the tree from 1 to 131072. Since the size of each node is 32 bytes, the total space requirement of 131072 nodes is 12 times larger than 256KB space of the local memory.

We can observe from Figure 9.1 that, only 6800 number of nodes can be created in the tree without any management, larger than which the program will crash. The reason is that stack data and heap data eventually consume up the total available space when more number of nodes are created in the tree. When managed, the application is assigned with all available space in the local memory. By doing this, we can fairly compare the performance of application *with* and *without* data management. We can see that, our schemes enable the execution of the application with very large number of nodes.

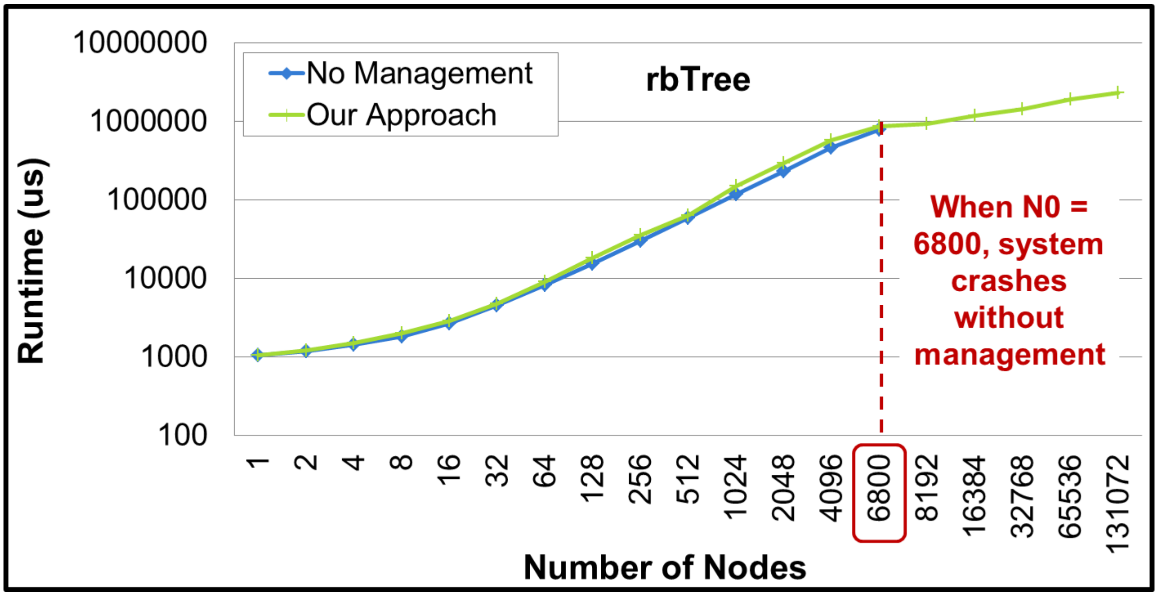


Figure 9.1: Supporting limitless stack data, heap data, and code

Table 9.1: Comparison against cache

	cache		Our approach
	number of misses	miss penalty (us)	miss overhead (us)
BasicMath	93561677	8514112	19
Dijkstra	6060791	551531	723
FFT	9318774	848008	74
FFT_inverse	7996980	727725	86
SHA	27402	2493	23
String_Search	134834	12269	174
Susan_Edges	57983	5276	11
Susan_Smoothing	314874	28653	19

Another observation is that there is a leap after the number of nodes is larger than 6800. This is because that data transfer between the main memory and the local scratchpad memory will happen after the space is full.

9.2 Comparison against Cache-based Architecture

We discussed that data and code management overhead comprises of DMA time for data transfer and the execution of additional instructions in the management library functions. However, the DMA time should not be fully counted as overhead. It is because, when there is a data miss in the hardware cache, there is also penalty for applications.

In Table 9.1, we show the differences between cache miss penalty and the overhead of our management. In this experiment, we use SimpleScalar [8] to collect cache misses of data and instructions. In SimpleScalar, the total cache size is configured to be equal to the size of local scratchpad memory. This size is set to be $4/5$ of the total memory requirement (namely, the space required for stack data, heap data, and code) of each application. The cache line size is 32 bytes and the cache is 4-way associative. When managed with our approach, the application uses the memory partition generated from Chapter 8. The penalty per miss used for calculating miss penalty is 91 nano seconds, which is the time of DMA latency on the IBM Cell processor [45]. As shown in Table 9.1, our memory management scheme has less miss overhead than cache miss penalty of cache-based processors. There are two reasons for less overhead: i) Data and instructions are initiated in the local memory and DMA occurs only when each region in scratchpad memory is full. ii) The management granularity is coarser in our data management, but the cache line size in cache based architecture cannot be too large.

Chapter 10

SUMMARY AND FUTURE WORK

As the number of cores on chip increases, the design complexity and power consumption of hardware cache coherence logic increases exponentially. Thus a more scalable memory architecture is expected in manycore processors. Software Managed Manycore (SMM) architectures emerge as one of promising solutions to the problem of scaling the memory hierarchy. Such architectures lack hardware cache and only contain a scratchpad memory on each core. As scratchpad memory cannot always accommodate the whole task mapped to it, certain schemes are required to mange code, global data, stack data and heap data of the program to enable its execution.

In this dissertation, we propose an infrastructure to manage unlimited size of stack data, heap data and code for SMM architectures: 1) Stack data: we propose SSDM to manage stack data for SMM architectures. It manages stack data within a constant space in the local scratchpad memory. It also addresses the stack pointer problem for any stack management approaches. Compared to CCSM, SSDM judiciously places management stubs in the program. In addition, SSDM also presents a systematic scheme to further optimize the stack pointer management. 2) Heap data: we propose a compilation and runtime system to manage unlimited size of heap data for SMM architectures. Moreover, our system supports multi-level heap pointers in the program. 3) Code: we formally define the problem of mapping code for SMM architectures at the granularity of function objects. Besides, we propose a correct cost calculation for code mapping, as well as a heuristic approach named CMSM for the same problem. 4) Memory partition: we propose an estimation scheme to partition memory space among stack data, heap data and code. It greatly reduces the time to get the best partitioning, at the expense of a slight loss of runtime performance. Experimental results on several benchmarks demonstrate that our scheme scales with different number of cores.

The work can be extended from five major directions: 1) integrating with general purpose parallel programming languages (for example, Open Computing Language (OpenCL) [51]); 2) support of cache/SPM mixed multicore architectures such as TI TMS320C6678 [62]; 3) support of Dynamically Linked Libraries (DLL); 4) support of object oriented languages; 5) automatic mapping of tasks to execution cores. In the rest of this chapter, we will discuss them in details.

10.1 Future Work on Programming Model

We adopt Open Computing Language (OpenCL) as the representative to discuss the work that can be done in this section. OpenCL is a framework for writing programs that execute across heterogeneous platforms. Typically, such platforms are composed of central processing units (CPUs), graphics processing units (GPUs), and several other processors. An OpenCL program requires programmers to define kernels executing on OpenCL devices, plus employing the provided application programming interfaces (APIs) to define and control the platforms. Academic researchers have investigated automatically compiling OpenCL programs into application-specific processors running on FPGAs [37], yet OpenCL doesn't have good support for memory management. It requires users to estimate the space requirement of kernels on platforms. Our work is orthogonal to these high-level work, and thus can be integrated into those OpenCL APIs.

10.2 Future Work on Mixed Cache/SPM Processors

In this section, we adopt TMS320C6678 as the representative to discuss the work that can be done. TI TMS320C6678 evaluation board has a single C6678 processor and a 2GB DDR3 memory. It is based on TI's KeyStone multicore architecture, and has eight cores on chip, each of which can run at up to 1.25 GHz. Each core has its own local memory, and all the cores share the off-chip DDR3 memory as the main memory. The local memory for each core can be either configured as scratchpad memory (SPM) or cache. Our work assumes each core only has a SPM without a hardware cache. In an architecture mixed with cache/SPM, we can evict old data to cache instead of the main memory to further optimize our management. In addition, applications can be analyzed to determine the partition of local memory to cache and SPM.

10.3 Supporting Dynamically Linked Libraries (DLL)

Dynamic Link Library (DLL) is an implementation of the shared library concept in operating systems (commonly seen in MS Windows operating systems). DLLs can contain code, program data, and OS-based resources, in any combination. Owing to the dynamic nature of its interaction with the underlying software, the amount and nature of data required for the program execution, is not known at compile time. In the case of SPM based local memory architectures, a runtime solution has to be provided that monitors for data accesses to DLLs, and triggers data re-allocation on the local memory, such that the data loaded does not disrupt the data needs of other program components. Since the DLL contains code, program data, and OS-based resources, some of which are shared among different threads of applications, we need to decide whether we should make a copy of shared data or code. In addition, we need to change our management scheme to dynamically manage the dynamic library.

10.4 Support for Object Oriented Languages

Our compiler supports C language on software managed manycores. When solutions for stack data, heap data and code are proposed, many features in object oriented language are not taken into consideration, e.g., virtual function, polymorphism, each of which require specific data management schemes to be implemented. As part of our approach, the compiler cannot fix which exact method a virtual function will call at compilation time. It then creates a virtual table for functions that from now on will always be consulted on each function call. In this case, we need to change all the methods whose addresses are stored in the virtual table. A table will be formed to store stack sizes and code sizes for all methods in the virtual table. During execution, our management function can look up the information and use it for efficient management.

10.5 Automatic Mapping of Tasks to Execution Cores

Our current solution assumes that programmers are responsible for manually distributing data and tasks to execution cores. Our compiler solution gives the mapped application low-level support, which automatically transfers stack/heap data and code between the main memory and the local scratchpad memory on each execution core. Since the mapping of tasks is not intuitive and is a cumbersome process (affecting productivity and time-to-market), we could globally analyze the existing applications that will be transformed to manycore applications, and provide a tool set to transparently perform this task. This task requires us to 1) decide which part of code should be mapped to the execution core; 2) estimate the communication overhead across different execution cores. 3) provide an efficient runtime environment and incorporate it to the managed applications.

REFERENCES

[1] *"ARM Architecture version 5 (ARMv5TE)"*. http://www.arm.com/.

[2] *"GCC Internals"*. http://gcc.gnu.org/onlinedocs/gccint/.

[3] *Intel Core i7-3770 Processor*. http://ark.intel.com/products/65719/.

[4] Intel Core i7 Processor Extreme Edition and Intel Core i7 Processor Datasheet, Volume 1. In *White paper*. Intel.

[5] *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[6] Dennis Abts, Steve Scott, and David J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proc. of IPDPS*, pages 11.2–, 2003.

[7] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A Post-compiler Approach to Scratchpad Mapping of Code. In *Proc. of CASES*, pages 259–267, 2004.

[8] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, February 2002.

[9] Oren Avissar, Rajeev Barua, and Dave Stewart. An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.

[10] Ke Bai, Di Lu, and Aviral Shrivastava. Vector Class on Limited Local Memory (LLM) Multi-core Processors. In *Proc. of CASES*, pages 215–224, 2011.

[11] Ke Bai, Jing Lu, Aviral Shrivastava, and Bryce Holton. CMSM: An Efficient and Effective Code Management for Software Managed Multicores. In *Proc. of CODES+ISSS*, 2013.

[12] Ke Bai and Aviral Shrivastava. Heap Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proc. of CODES+ISSS*, pages 317–326, 2010.

[13] Ke Bai and Aviral Shrivastava. A Software-Only Scheme for Managing Heap Data on Limited Local Memory (LLM) Multicore Processors. *Trans. on Embedded Computing Sys.*, 13(5), 2013.

[14] Ke Bai and Aviral Shrivastava. Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures. In *Proc. of DATE*, pages 593–598, 2013.

[15] Ke Bai, Aviral Shrivastava, and Saleel Kudchadker. Stack Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proc. of ASAP*, pages 231–234, 2011.

[16] Michael A. Baker, Amrit Panda, Nikhil Ghadge, Aniruddha Kadne, and Karam S. Chatha. A Performance Model and Code Overlay Generator for Scratchpad Enhanced Embedded Processors. In *Proc. of CODES+ISSS*, pages 287–296, 2010.

[17] M. Balakrishnan, Peter Marwedel, Lars Wehmeyer, Nils Grunwald, Rajeshwari Banakar, and Stefan Steinke. Reducing Energy Consumption by Dynamic Copying of Instructions onto On-chip Memory. In *Proc. of ISSS*, pages 213–218, 2002.

[18] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache on-chip Memory in Embedded Systems. In *Proc. of CODES+ISSS*, pages 73–78, 2002.

[19] S. Borkar. Major Challenges to Achieve Exascale Performance. In *Salishan Conference on High-Speed Computing*, 2009.

[20] Garo Bournoutian and Alex Orailoglu. Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive Mobile Processors. In *Proc. of CODES+ISSS*, pages 89–98, 2011.

[21] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proc. of ASPLOS*, pages 224–234, 1991.

[22] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proc. of PACT*, pages 155–166, 2011.

[23] A. Dominguez, S. Udayakumaran, and R. Barua. Heap Data Allocation to Scratch-pad memory in Embedded Systems. *J. Embedded Comput.*, 1(4):521–540, 2005.

[24] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization. In *Proc. of CASES*, pages 223–233, 2006.

[25] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad Memory Management for Portable Systems with A Memory Management Unit. In *Proc. of EMSOFT*, pages 321–330, 2006.

[26] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine^{TM} Architecture. *IBM Syst. J.*, 45(1):59–84, January 2006.

[27] B. Flachs, S. Asano, Sang H.Dhong, H.P. Hofstee, G. Gervais, Roy Kim, T. Le, Peichun Liu, J. Leenstra, J. Liberty, B. Michael, Hwa-Joon Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D.A. Brokenshire, M. Peyravian, Vandung To, and E. Iwata. The Microarchitecture of the Synergistic Processor for A Cell Processor. *IEEE Solid-state circuits*, 41(1):63–70, 2006.

[28] A. Garcia-Guirado, R. Fernandez-Pascual, A. Ros, and J.M. Garcia. Energy-Efficient Cache Coherence Protocols in Chip-Multiprocessors for Server Consolidation. In *Proc. of ICPP*, pages 51–62, 2011.

[29] James R. Goodman. Using Cache Memory to Reduce Processor-memory Traffic. In *Proc. of ISCA*, pages 255–262, 1998.

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A Free, Commercially Representative Embedded Benchmark Suite. *Proc. of the Workload Characterization*, pages 3–14, 2001.

[31] M. R. Guthaus et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of Workload Characterization*, pages 3–14, 2001.

[32] Tom's Hardware. Raw Performance: SiSoftware Sandra 2010 Pro (GFLOPS).

[33] Mark Heinrich, Vijayaraghavan Soundararajan, John Hennessy, and Anoop Gupta. A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. *IEEE Trans. Comput.*, 48(2):205–217, February 1999.

[34] J. Howard, S. Dighe, S.R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V.K. De, and R. Van Der Wijngaart. 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*, 46(1):173–183, 2011.

[35] Shih-Hao Hung, Chia-Heng Tu, and Wen-Long Yang. A Portable, Efficient Inter-core Communication Scheme for Embedded Multicore Platforms. *J. Syst. Archit.*, 57(2):193–205, February 2011.

[36] IBM. Programmer's Guide: Software Development Kit for Multicore Acceleration Version 3.1. Technical report, 2008.

[37] Pekka O. Jääskeläinen, Carlos S. de La Lama, Pablo Huerta, and Jarmo H Takala. OpenCL-based Design Methodology for Application-specific Processors. In *Proc. of SAMOS*, pages 223–230, 2010.

[38] Andhi Janapsatya, Aleksandar Ignjatović, and Sri Parameswaran. A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric. In *Proc. of ASP-DAC*, pages 612–617, 2006.

[39] Choonki Jang, Jaejin Lee, Bernhard Egger, and Soojung Ryu. Automatic Code Overlay Generation and Partially Redundant Code Fetch Elimination. *ACM Trans. Archit. Code Optim.*, 9(2):10:1–10:32, June 2012.

[40] Seung Chul Jung, Aviral Shrivastava, and Ke Bai. Dynamic Code Mapping for Limited Local Memory Systems. In *Proc. of ASAP*, pages 13–20, 2010.

[41] M. Kandemir and A. Choudhary. Compiler-directed Scratch Pad Memory Hierarchy Design and Management. In *Proc. of DAC*, pages 628–633, 2002.

[42] Mahmut T. Kandemir, J. Ramanujam, and Alok N. Choudhary. Exploiting Shared Scratch Pad Memory Space in Embedded Multiprocessor Systems. In *Proc. of DAC*, pages 219–224, 2002.

[43] Mahmut T. Kandemir, J. Ramanujam, Mary Jane Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and Amisha Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Proc. of DAC*, pages 690–695, 2001.

[44] Arun Kannan, Aviral Shrivastava, Amit Pabalkar, and Jong-eun Lee. A Software Solution for Dynamic Stack Management on Scratch Pad Memory. In *Proc. of ASP-DAC*, pages 612–617, 2009.

[45] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, May 2006.

[46] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of ISCA*, pages 148–159, 1990.

[47] Lian Li, Hui Feng, and Jingling Xue. Compiler-directed Scratchpad Memory Management via Graph Coloring. *ACM Trans. Archit. Code Optim.*, 6(3):1–17, 2009.

[48] Lian Li, Lin Gao, and Jingling Xue. Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In *Proc. of PACT*, pages 329–338, 2005.

[49] Jing Lu, Ke Bai, and Aviral Shrivastava. SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs). In *Proc. of DAC*, pages 149–156, 2013.

[50] R. McIlroy, P. Dickman, and J. Sventek. Efficient Dynamic Heap Allocation of Scratch-pad Memory. In *Proc. of ISMM*, pages 31–40, 2008.

[51] A. Munshi. The OpenCL Specification, version 1.0. In *Khronos OpenCL Working Group*, 2009.

[52] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory Allocation for Embedded Systems with A Compile-time-unknown Scratch-pad Size. In *Proc. of CASES*, pages 115–125, 2005.

[53] Kathryn O'Brien. Issues and Challenges in Compiling for the CBEA. In *Proc. of LCTES*, pages 134–134, 2007.

[54] Amit Pabalkar, Aviral Shrivastava, Arun Kannan, and Jongeun Lee. SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories. In *Proc. of HPC*, pages 569–582, 2008.

[55] Francesco Poletti, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose Manuel Mendias. An Integrated Hardware/Software Approach for Run-time Scratchpad Management. In *Proc. of DAC*, pages 238–243, 2004.

[56] Randolf Rotta, Thomas Prescher, Jana Traue, and Jorg Nolte. In-Memory Communication Mechanisms for Many-Cores – Experiences with the Intel SCC.

[57] Aviral Shrivastava, Arun Kannan, and Jongeun Lee. A Software-only Solution to Use Scratch Pads for Stack Data. *IEEE TCAD*, 28(11):1719–1728, 2009.

[58] Richard Simoni and Mark Horowitz. Dynamic Pointer Allocation for Scalable Cache Co- herence Directories. In *Proc. of ISSMM*, pages 72–81, 1991.

[59] James E. Smith. A Study of Branch Prediction Strategies. In *Proc. of ISCA*, pages 135–148, 1981.

[60] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proc. of DATE*, page 409, 2002.

[61] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, June 1990.

[62] Texas Instruments. TMS320C6678.

[63] Ehsan Totoni, Babak Behzad, Swapnil Ghike, and Josep Torrellas. Comparing the Power and Performance of Intel's SCC to state-of-the-art CPUs and GPUs. In *Proc. of ISPASS*, pages 78–87, 2012.

[64] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic Allocation for Scratchpad memory Using Compile-time Decisions. *Trans. on Embedded Computing Sys.*, 5(2):472–511, 2006.

[65] M. Verma and P. Marwedel. Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. *IEEE VLSI*, 14(8):802–815, 2006.

[66] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *Proc. of DATE*, page 21264, 2004.

[67] Manish Verma, Klaus Petzold, Lars Wehmeyer, Heiko Falk, and Peter Marwedel. Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach. In *ESTImedia*, pages 115–120, 2005.

[68] Yi Xu, Yu Du, Youtao Zhang, and Jun Yang. A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs. In *Proc. of ICS*, pages 285–294, 2011.