Application-aware Performance Optimization for Software Managed Manycore Architectures

by

Jing Lu

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

Approved March 2019 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Hessam Sarjoughian
Carole-Jean Wu
Adam Doupe

ARIZONA STATE UNIVERSITY

May 2019

ABSTRACT

One of the main goals of computer architecture design is to improve performance without much increase in the power consumption. It cannot be achieved by adding increasingly complex intelligent schemes in the hardware, since they will become increasingly less power-efficient. Therefore, parallelism comes up as the solution. In fact, the irrevocable trend of computer design in near future is still to keep increasing the number of cores while reducing the operating frequency. However, it is not easy to scale number of cores. One important challenge is that existing cores consume too much power. Another challenge is that cache-based memory hierarchy poses a serious limitation due to the rapidly increasing demand of area and power for coherence maintenance.

In this dissertation, opportunities to resolve the aforementioned issues were explored in two aspects.

Firstly, the possibility of removing hardware cache all together, and replacing it with scratchpad memory with software management was explored. Scratchpad memory consumes much less power than caches. However, as data management logic is completely shifted to Software, how to reduce software overhead is challenging. This thesis presents techniques to manage scratchpad memory judiciously by exploiting application semantics and knowledge of data access patterns, thereby enabling optimization of data movement across the memory hierarchy. Experimental results show that the optimization was able to reduce stack data management overhead by 13X, produce better code mapping in more than 80% of the case, and improve performance by 83% in heap management.

Secondly, the possibility of using software branch hinting to replace hardware branch prediction to completely eliminate power consumption on corresponding hardware components was explored. As branch predictor is removed from hardware, soft-

ware logic is responsible for reducing branch penalty. Techniques to minimize the branch penalty by optimizing branch hint placement were proposed, which can reduce branch penalty by 35.4% over the state-of-the-art.

ACKNOWLEDGMENTS

I would like to express my special appreciation to my advisor Dr. Aviral Shrivastava for your constant support, continued advice and inspiration. I'm grateful that you brought me into this area and offered me the great opportunity to study in Compiler Microarchitecture Lab. You've been a mentor to me both academically and personally. Academically, I appreciate your time, idea, discussion, and encouragement to make my Ph.D. experience productive and stimulating. Personally, you inspired me by your hardworking and passionate attitude.

I would like to thank my committee members, Dr. Hessam Sarjoughian, Dr. Carole-Jean Wu, Dr. Adam Doupe for serving at my committee. Thank you for your brilliant comments and suggestions. Without your valuable inspireation and advice I won't be able to finish this thesis.

I am also grateful to my team members and collaborators, Reiley Jayapaul, Yooseong Kim, Chuan Huang, Jian Cai, Bryce Holton, Di Lu, Fei Hong, Jinn-Pean Lin and the many others, for your support and encouragement. I enjoyed technical discussions, team lunches, weekend hikings, and happy hours with you. I'm proud to be surrounded by so many brilliant and enthusiastic researchers of you.

I would like to show my deepest gratitude to my husband, my best friend, and also my collaborator Ke Bai for your constant love, support and encouragement. I would not be able to finish my Ph.D. study without your faithful support.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 Manycore Architecture Design 1

2 SMM: A Promising Approach 5

3 Challenges in Shifting the Intelligence from Hardware to Software 10

3.1 Challenge of Data Management on SMM 10

3.2 Challenge of Software Branch Hinting on SMM 13

4 Contributions of This Dissertation 14

4.1 Publications and My Contributions in the Publications 15

5 Software Branch Hinting for SMM 17

5.1 Overview 17

5.2 Branch Hinting Mechanism 19

5.3 Branch Penalty Model 21

5.4 Problem Formulation 25

5.5 Branch Hint Management 26

5.5.1 NOP Padding 26

5.5.2 Hint Pipelining 28

5.5.3 Nested Loop Restructuring 31

5.5.4 Branch Penalty Reduction Heuristic 34

5.6 Related Work 36

5.7 Experimental Results 38

5.7.1 Experimental Setup 38

5.7.2 Branch Penalty Reduction 39

CHAPTER Page

5.7.3 Effectiveness of NOP Padding 42
5.7.4 Performance Improvement 43
5.8 Summary 44
6 Data Management for SMM 46
6.1 Stack Data Management 46
6.1.1 Motivation 46
6.1.2 Challenges 48
6.1.3 Smart Stack Data Management 49
6.1.4 Related Work 61
6.1.5 Experimental Results 65
6.1.6 Summary 73
6.2 Effective Code Management 74
6.2.1 Motivation 74
6.2.2 Code Overlay Mechanism 74
6.2.3 Objective of Code Overlay 76
6.2.4 Cost Calculation of Code Overlay 77
6.2.5 CMSM Heuristic 82
6.2.6 Related Work 83
6.2.7 Experimental Results 87
6.2.8 Summary 91
6.3 Heap Data Management 92
6.3.1 Motivation and State of the Art 92
6.3.2 Efficient Heap Data Management 95
6.3.3 Experimental Results 105

CHAPTER Page

6.4 Compiler and Runtime Infrastructure 110
7 Summary 113
REFERENCES 115

LIST OF TABLES

Table Page

5.1 Branch Penalty Can Be Crippling in the Cell SPU in Absence of Any Branch Hints. 18

6.1 Benchmarks, the Number of Nodes and Edges in Their WCG, Their Stack Sizes, and the Scratchpad Space We Manage Them on. 66

6.2 Library Code Size of Stack Manager (In Bytes) 70

6.3 Comparison of Number of DMAs 71

6.4 Number of *_sstore* and *_sload* calls 72

6.5 Dynamic Instructions Per Function 73

6.6 Benchmarks, Their Minimum Sizes of Code Space, and Maximum Sizes of Code Space. 88

6.7 Maximum Heap Usage of Benchmarks 106

6.8 Number of *g2l* Calls with and without Heap Access Detection Technique 108

6.9 Instructions Executed per *g2l* with and without Different Optimization Techniques 109

6.10 Runtime Library for Data and Code Management 112

LIST OF FIGURES

Figure Page

2.1 A Software Managed Manycore (SMM) Architecture Has Multiple Cores: 1) Each Core with a ScratchPad Memory (SPM) But No Hardware Cache. The Data Transfers Between the SPMs and the Main Memory of the System Take Place via Direct Memory Access (DMA) Instructions Which Must be Explicitly Specified in the Application. 2) Each Core is Made Simpler, Namely, Hardware Branch Predictor is Removed Completely. Branch Prediction Should be Accomplished by Software Branch Hinting. 6

2.2 Hardware View of Difference Between Cache and SPM: SPM is Just a Raw Memory Without the Hardware Mechanism to Manage It (as is Present in Caches). 7

3.1 The Data Movement from and to Caches is Performed Automatically in Hardware, in SPM-based Manycore Systems, it Has to be Present Explicitly in the Software in the Form of Data Movement Instructions, e.g., DMA Instruction. 11

3.2 The Virtual Anatomy of the Compiled Program in Local Scratchpad Memory 12

5.1 Software Branch Hinting is Characterized by Hint Instructions Setting the BTB Entries. It Has 3 Key Parameters, 1) d, the Number of Pipeline Stages Where BTB is Set, 2) f, The Time To Fetch Target Instructions, and 3) s, the Number of Entries in BTB. 20

5.2 Branch Penalty is Plotted as We Increase the Separation When Hint is Correct. We Need at least 8 Instructions for a Hint to Become Effective, and the Penalty Decreases as Separation Increases. 23

Figure Page

5.3 Misprediction Penalty is Plotted as We Increase the Separation. On Top of Branch Penalty, the Time to Flush Pipeline is Added Resulting Larger Branch Penalty Than When Hint is Correct. 24

5.4 (a) Before NOP Padding, the Branch Cannot be Hinted. (b) NOP Padding Enables Hinting the Branch. 27

5.5 (a) Before Hint Pipelining, Branch b_1 Cannot Be Hinted Due to the Hoisted Hint for b_2 (b) After Hint Pipelining, Both b_1 and b_2 are Hinted. 29

5.6 (a) Before Nested Loop Restructuring, the Separation for b_4 is Limited to l_4. (b) After Nested Loop Restructuring, the Outer Loop Branch is Changed to Unconditional Branch b_2, and the Separation is Increased to $l_2 + l_4$. 31

5.7 (a) Hint for b_3 Can be Hoisted into $L2$. Branch b_4 Cannot be Hinted Even After Loop Restructuring. (b) After Restructuring, The Hint for b_3 Cancels the Hint for b_4. (c) Pipelining is Applied and Both Branches Can be Hinted. 33

5.8 The Percentage of Branch Penalty in The Total Execution Cycles After GCC Inserts Hints Into the Program. Benchmarks are Grouped Into Two Groups 'high' and 'low' According to the Percentage. 39

5.9 *Reduction of Branch Penalty is* 35.4% *at Maximum and* 19.2% *on Average.* 40

5.10 (a) GCC Can Only Hint the Innermost Loop Branch. (b) The Proposed Technique Can Hint All of The Four Loop Branches. 41

Figure Page

5.11 Execution Time Comparison Between Our NOP Padding Technique and GCC's "-mhint-max-nops" Option. In All Benchmarks, Our Technique Outperforms GCC. GCC Even Results in Performance Degradation for Several Benchmarks. 42
5.12 Performance Improvement Obtained with the Proposed Heuristic is 18% at Maximum. 43
6.1 Suppose We Want to Execute the Program Shown in (a) On the Execution Core with Local Scratchpad Memory. (b) Shows that We Can Easily Manage the Stack Data of This Program in 100 Bytes, However, Trying to Manage it in Only 70 Bytes Local Memory Requires Data Management. 47
6.2 An Overview of SSDM Infrastructure 50
6.3 WCG with Cuts of Benchmark SHA: The Edge with Dashed Yellow Color Represents an Artificial Edge for Root Node and Leaf Node. 52
6.4 Illustration of SSDM Heuristic: the Values on Edges Are the Numbers of Function Calls. 58
6.5 An Example Shows Static Edge Weight Assignment. 60
6.6 In the ARM Processor, SPM is in Addition to the Regular Memory Hierarchy, While in SMM System, the Local Memory is an Essential Part of the Memory Hierarchy on the Execution Core. 62

Figure Page

6.7 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. 63

6.8 **Pointer Management** - Function F2 Accesses the Pointer p, Which Points to a Local Variable 'a' of Function F1. Since 'a' is a Local Variable on the Stack of F1, It Has a Local Address. When F2 is Called, if F1 is Evicted From the Local Memory, Then the Pointer p Will Point to a Wrong Value. This is Fixed by Assigning a Global Address to the Pointer When It is Created (Through **_l2g**), and Then When Needed, It is Accessed Through **_g2l**. Finally It is Written Back Using **_wb**. 64

6.9 Performance Improves When Stack Region Size Increases. 67

6.10 SSDM is Scalable, Since Performance Regression is Negligible When the Number of Cores Increases. 68

6.11 Performance Comparison Between SSDM and CSM. 69

6.12 Overhead Comparison Between SSDM and CSM. 70

6.13 Code Overlay on Scratchpad Memory: When Task Assigned to the Execution Core Requires Larger Memory Than the Available Space, Code Needs to Be Mapped Between External Shared Main Memory and the Local Scratchpad Memory of the Core. 75

6.14 The GCCFG for the Example Code 78

Figure Page

6.15 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. 85
6.16 Performance Comparison Against FMUM and FMUP 89
6.17 Scalability of CMSM on Multicore Processors 91
6.18 Performance Overhead with the State-of-the-art Heap Management. . . . 93
6.19 The Previous Approach Inserts *g2l* Before Every Memory Access, While Ours Tries to Identify Heap Accesses Statically and Skip Unnecessary *g2l*s. 97
6.20 When It Cannot Be Determined at Compile-time Whether There is a Heap Access, We Check it at Run-time. 100
6.21 Comparison of Heap Management Workflow. 101
6.22 De-dupe Management Calls and Move Common Operations to the Beginning of the Caller Function. 103
6.23 The Execution Time of Our Approach Normalized to the Previous Work with Optimizations Incrementally Added. 107
6.24 A Direct-mapped Cache other than a 4-way Set-associative Cache Reduces More Execution Time Thanks to Simplified Management Functions, Compared to the Extra Time Introduced due to Increased Cache Misses. 110
6.25 General Compilation Flow for Data Management on SMM Architectures 111

Chapter 1

MANYCORE ARCHITECTURE DESIGN

Higher performance is undeniably expected over all computing platforms, from sensor, handheld devices (such as watches, cell-phones, and tablets), to laptops, desktops, servers, data centers. However, it can no longer be simply obtained by increasing the operating frequency. This is because power consumption increases cubically with frequency of operation, while most computing systems are limited by power, energy and thermal constraints. High performance computing centers and data centers are designed with the constraint of total power draw, embedded platforms are often designed around battery capacity, and the rest systems in the middle are designed with thermal constraints.

One of the main goals of computing architecture design in this decade is to improve performance without much increase in the power consumption. It cannot be achieved by adding increasingly intelligent schemes for caches and branch prediction in the hardware, since, by the law of diminishing returns, they will become increasingly less power-efficient. Therefore, parallelism comes up as the solution. As a result, the irrevocable trend of computer design in near future is still to keep increasing the number of cores while reducing the operating frequency. The new interpretation of Moore's law states that the number of cores will double every two years. Soon, we will have architectures that have hundreds of cores. Industry experts project over a thousand cores per chip in about a decade Borkar (2009).

Hardware intelligence logics, such as caches and branch predictors, have been a part of processor design since the earliest of processors including IBM 360 in early 70s, and the capabilities of these intelligence logics have increased over time. Caches

store frequently accessed data in a memory close to the processor, and make the memory accesses faster and consume lower power. Branch predictors improve the flow of instructions in the instruction pipeline and therefore play a critical role in achieving high effective performance. However, as we scale to manycore systems, it becomes increasingly challenging to scale the corresponding cache-based memory hierarchies and branch predictor.

One important reason is that existing cores consume too much power. For example, the Intel Core i7-7700 (Kaby Lake-S, 14nm) with 4-core consumes 91W at 4.2GHz Intel (2017). If we use these cores to design a 100-core processor, then the total power consumption of the processor will be $91 * 100/4 = 2275$W. As the current thermal cap of packaging technologies is about 250W, this is definitely unsustainable. We have to trade off performance of a single core in order 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*.

Another important reason is that the current coherent-cache architecture designs are not scalable for hundreds and thousands of cores Bournoutian and Orailoglu (2011); Choi *et al.* (2011); Garcia-Guirado *et al.* (2011); Xu *et al.* (2011). Coherence is mainly implemented with two mechanisms in hardware, directory based and snooping. Snooping scheme Goodman (1998) is a process where the individual caches monitor address lines for accesses to memory locations that they have cached. It is called a write invalidate protocol when a write operation is observed to a location that a cache has a copy of and the cache controller invalidates its own copy of the snooped memory location. However, when we try to scale the design to hundreds of cores, this monitoring and invalidation become the bottleneck. Directory based protocols Lenoski *et al.* (1990); Chaiken *et al.* (1991); Heinrich *et al.* (1999); Simoni and Horowitz (1991) scale better with the number of cores. In a directory-based system,

the data being shared is placed in a common directory that maintains the coherence between caches. The directory keeps an entry for every cache block to identify the cache that contains the most up to date copy of the block. For a 1024-core processor, each entry of the directory will be 128 bytes (1024 bits divides 8 bits per byte) in full map implementation. As it is also the typical cache block size, the 100% area cost is overwhelming. Worse than that, this extra transistor requirement adds significant power and performance overheads. Although there are some schemes that attempt to mitigate the space overhead, they do so by making the directory structure distributed and hierarchical. This not only increases latency, but also makes the coherence protocol distributed which are notoriously challenging to design and verify Stenström (1990); Abts *et al.* (2003).

Consequently, designing manycore processors is not a simple extension of the processor design today. In order to make hardware more scalable to hundreds and thousands of cores, each core has to be made simpler, and therefore more power-efficient. We need to think anew in higher layers of system design. Due to this requirement, a lot of scaling solutions have been employed in modern manycore architectures in recent years.

Some manycore architectures maintain cache coherece in hardware when power efficiency is not a critical requirement, but still have software-managed scratchpad as an open option when reducing coherence management overhead becomes necessary. The 9-, 16-, 32- or 64-core TilePro processors manufactured by Tilera use the Dynamic Distributed Cache (DDC) to provide a hardware-managed, cache-coherent approach to shared memory Tilera (2013). This technique allows the shared memory pages to be hosted on a specific core, and cached remotely in other cores. The coherence was realized by message passing among different cores. Needless to say, the communication overhead of maintaining such a hardware-managed, cache-coherent memroy system

is very high. As a result, the Quanta S2Q Server, which contains 8 TILEPro64 chips (512 cores) runs at more than 400w, which is approaching the thermal cap. For power efficient optimization, TilePro has a software-programmable hardware direct memroy access engine (DMA) implemented and allows users to use part of the chache as a scratchpad memory to improve the power efficiency.

Some other architectures scale well on memory hierarchy, but with the price of cache size and core size. The Intel Many Integrated Core (MIC) processors fall into this category. In 2015, Intel unveiled a 72-core x86 Knights Landing processor Sodani *et al.* (2016), which features high bandwidth, integrated memory. The Knights Landing processor has two types of memory: multichannel DRAM (MCDRAM) and double data rate (DDR) memory. MCDRAM is the 16 GB high bandwidth memory, which could be configured as a hardware-managed coherent, memory-side cache. Though power efficient, it's large cache size and core size makes it only suitable for exascale supercomputing applications.

GPUs (Graphics Processing Units) could be considered as another form of many-core architectures, which could have cores numbered in 100s or 1000s. The computational power of modern GPUs can easily reach teraFLOP scale. Though originally designed for graphics processing, modern GPUs are extensively used for general-purpose programming. However, GPUs are more like vector processors, whose instruction operate on vectors. Generally, GPUs are only suitable for applications that are highly parallel.

Chapter 2

SMM: A PROMISING APPROACH

Many efforts from both research space and industrial spheres have been spent in search of designs that could scale to manycore processors. Software Managed Manycore (SMM) architecture is a design philosophy that has emerged as a solution to this problem. In SMM architectures, each of cores are simplified as much as possible, and intelligence are shifted from hardware to software. For example, Intel SCC Howard *et al.* (2011) and Kalray MPPA-256 Dinechin *et al.* (2013) removed the cache coherence logic from hardware, and implement the coherence in software. Since coherence is not supported, applications written in the multithreading paradigm will not work directly. Message passing paradigm, and scatter gather (where no data dependency exists among tasks) are a natural fit for such an architecture. 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 Hung *et al.* (2011); Rotta *et al.* (2012). However, merely moving cache coherence logic to software does not resolve the power cap challenge that is faced in scaling to manycore architectures. For example, the Intel SCC Howard *et al.* (2011) consumes 125W at 1.14V, out of which 87.7W is spent on cores Totoni *et al.* (2012). If we scale the number of cores to 1000, the total power consumption of the processor will be $87.7/48 * 1000 = 1827$W, which is clearly prohibitive. To enable scaling to thousands of cores, we have to reduce the power consumption of the cores by 10X.

Another more aggressive option is to remove the hardware caches all together, and to employ software cacheing mechanisms for smart data management, using the raw

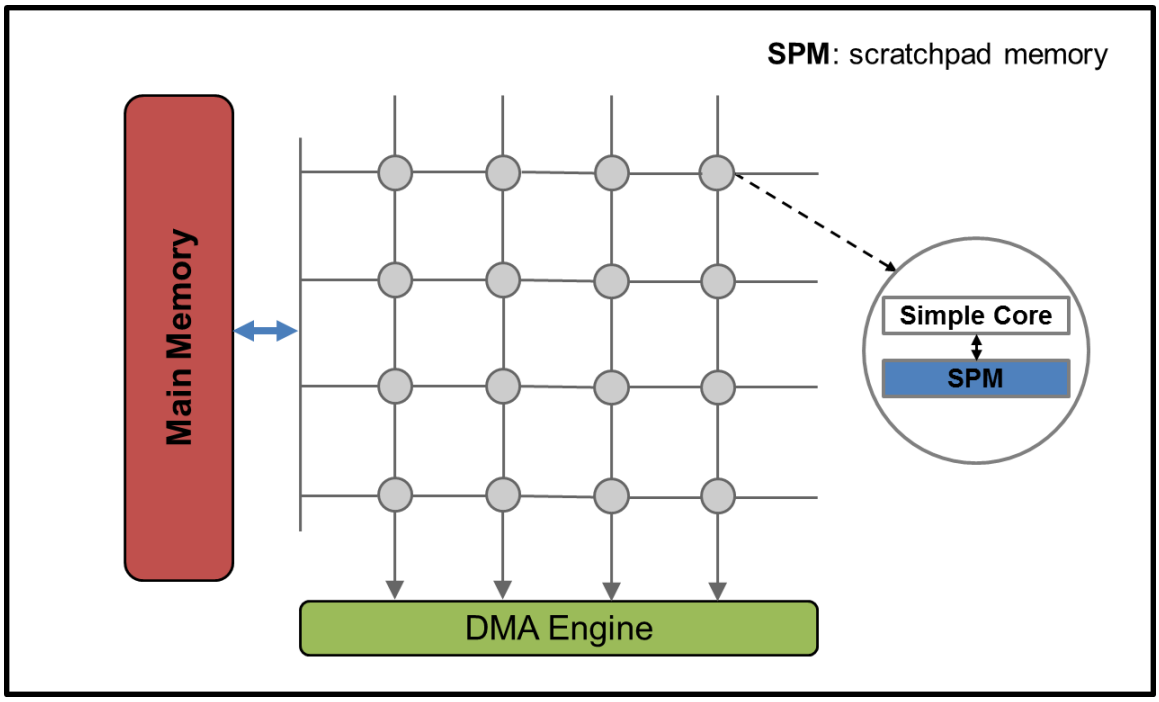

Figure 2.1: A Software Managed Manycore (SMM) Architecture Has Multiple Cores: 1) Each Core with a ScratchPad Memory (SPM) But No Hardware Cache. The Data Transfers Between the SPMs and the Main Memory of the System Take Place via Direct Memory Access (DMA) Instructions Which Must be Explicitly Specified in the Application. 2) Each Core is Made Simpler, Namely, Hardware Branch Predictor is Removed Completely. Branch Prediction Should be Accomplished by Software Branch Hinting.

memories in the processor. Here the data movement between the close-to-processor memory and the main memory has to be done explicitly in software, typically done through the use of Direct Memory Access (DMA) instructions. Figure 2.1 shows an example of a typical SMM architecture. SMM architecture replaces hardware cache with only ScratchPad Memory (SPM) Banakar *et al.* (2002) in the cores and all cores on the processor share a larger main memory. SPM is attached to the processor in much the same way as the L1 cache. However, SPM is raw memory, in the sense that it only contains decoding and column access logic, without the complex circuitry required to achieve hardware control of replacement policies, and managing coherence (tag directory, tag look-up circuitry, etc.). As Figure 2.2 shows, while a cache stores

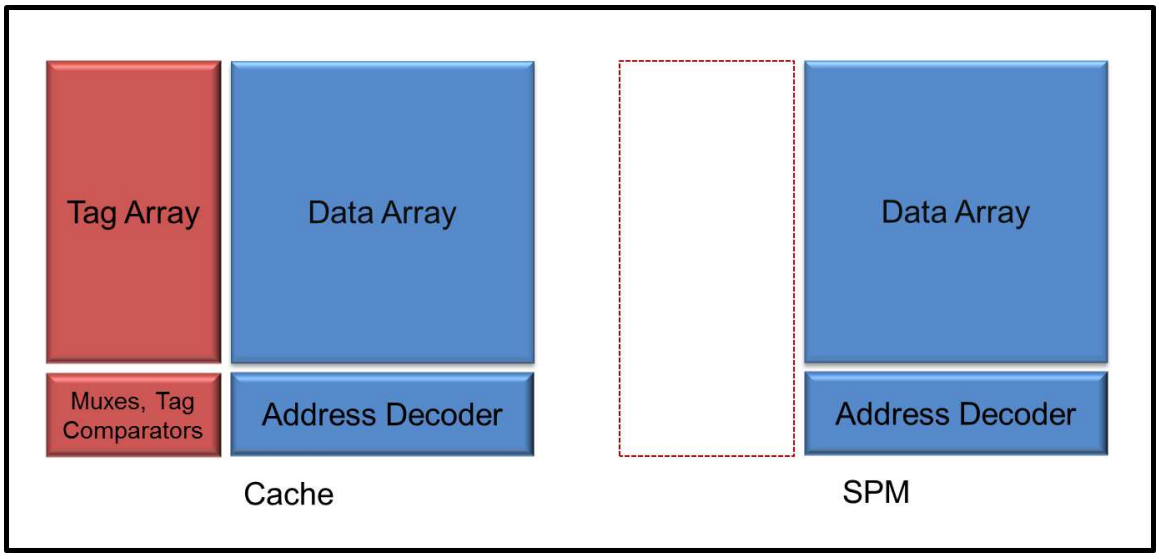

Figure 2.2: Hardware View of Difference Between Cache and SPM: SPM is Just a Raw Memory Without the Hardware Mechanism to Manage It (as is Present in Caches).

both the data and its address, an SPM only stores data, avoiding the extra lookup circuitry. Therefore, SPMs are about 30% smaller in area, slightly faster, yet consume about 30% less power than direct mapped caches (for the same data capacity) Banakar *et al.* (2002).

Execution on SMM systems can be more efficient than on cache based processors. Caches are a one-size-fits-all approach, which have only one general style of data management, regardless of how the data is actually accessed. Either data is accessed randomly or it is accessed in a first-in-first-out manner on a cache-based system, it will always be accessed in the manner implemented in hardware. In contrast, SPM-based systems allow more efficient management of data by exploiting application semantics and knowledge of data access patterns, thereby enabling customization of data movement across the memory hierarchy. For example, SPMs can manage stack data at stack-frame granularity, which is much more natural than managing them by cache blocks. By deploying a coarser granularity of data management, we may be able to reduce the number of times thus the overhead for checking if the requested data

is in the fast memory. More importantly, by analyzing access patterns of application data, we can achieve further efficiency. If we know several stack frames will always be in the SPM at the same time by analyzing the call graph of the program, we can bring these stack frames from the main memory into the SPM at once, instead of fetching each of them separately. As a result, we can reduce number of data transfers and alleviate the overall DMA startup cost. Furthermore, this eliminates status checking of stack frames whenever either one calls the other.

As power-efficiency becomes paramount concern in processor design, another promising option is to get rid of hardware branch prediction, and relies solely on software branch hinting.

Branch predictor has long been an important study area because of its powerful ability in performance leverage. Firstly, it reduce branch penalty, especially considering the fact that piplelines are becoming longer. On the other hand, branch predictor can serve to improve instruction level parallalism in out of order exection.Without branchprediction, reorder can only be done inside basic block. With branch predictor, more instructions can be prefeched, which increase the opportunity of reordering. As a result, branch predictor is becoming more and more complex. However, with the total power budget capped, more cores could only be added by reducing the power and the complexity of each core Gschwind *et al.* (2006); Hofstee (2005). Consequently, reducing branch predictor energy consumption becomes import, as branch predictors already account for a large fraction of on-chip power dissipation, which is as much as 10% Parikh *et al.* (2002). Software branch hinted processors target on getting rid of the complex hardware branch predictor.

In software branch hinted processors, applications may contain branch hint instructions which indicate that the branch instructions at specified PC addresses will jump to specified target addresses. After executing a hint instruction, the hardware

will start to speculatively execute target instructions when the specified branch instruction is executed.

Since both data management and branch prediction are passed to software, SMM architecture proves to be extremely power-efficient, if software intelligence is high enough. For example, the IBM Cell processor belonging to such architecture can compute at a power-efficiency of roughly 5 GFlops per watt Flachs *et al.* (2006). In contrast, Intel i7 4-core Bloomfield 965 XE can only achieve a power-efficiency of 0.5 GFlops per watt Intel (2010); Hardware (2010).

Chapter 3

CHALLENGES IN SHIFTING THE INTELLIGENCE FROM HARDWARE TO SOFTWARE

SMM architecture has scalable memory design and much less complicated branch predictor, and therefore is potentially more power-efficient.

3.1 Challenge of Data Management on SMM

The memory hierarchy, ScratchPad Memories (SPMs), on SMM architectures are functionally similar to caches, from the aspect that they allow for fast access to frequently used data but with lower power and latency. However, the main challenge in using SPM-based memories is that the data of the program must be explicitly managed in the software, as shown in Figure 3.1. Using caches is seamlessly integrated into the whole follow of program execution. If required data is not present in the cache, hardware mechanisms are built to bring in the requested data to the cache, potentially preventing the necessity of repeated operations if the data is reused. However, SPM contains no such automatic hardware mechanism to bring the desired data to the SPM. It must be brought in explicitly through memory transfer instructions that trigger Direct Memory Access (DMA) transfers. Furthermore, once data is brought in, it must be accessed using its new local SPM address, and not the original main memory address. Figure 3.1 shows that the left pseudo code can execute on any cache based architectures, even if the variable *global* is not in the cache before its execution. This automatic data movement is not provided in SMM architectures. For such architectures, the software must be modified as shown in the right hand side.

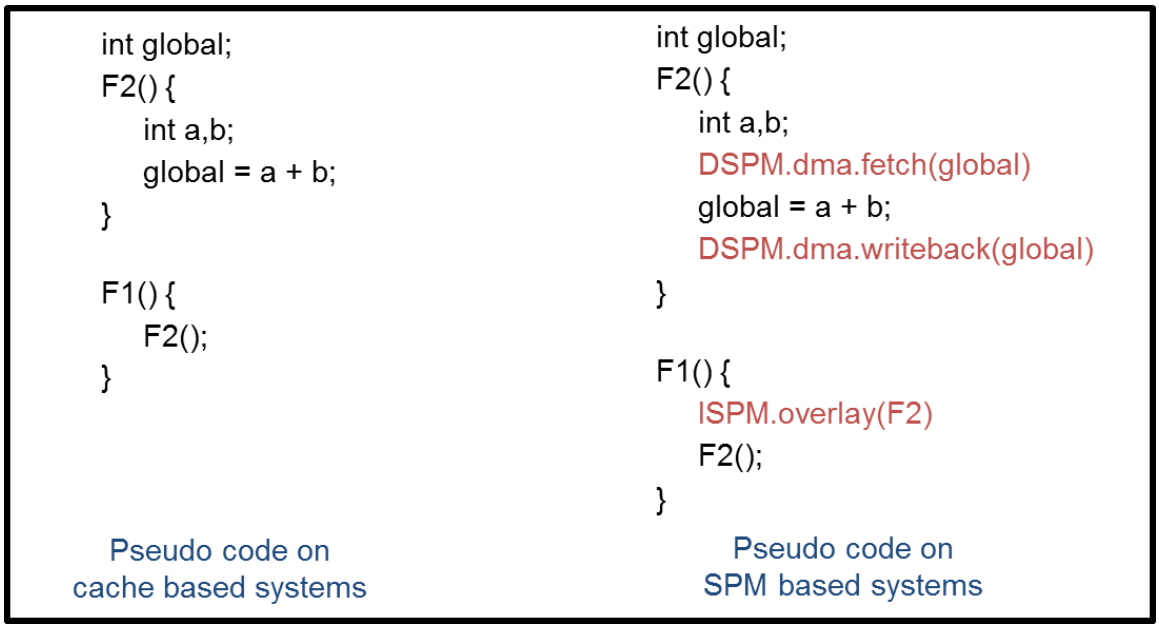

Figure 3.1: The Data Movement from and to Caches is Performed Automatically in Hardware, in SPM-based Manycore Systems, it Has to be Present Explicitly in the Software in the Form of Data Movement Instructions, e.g., DMA Instruction.

DMA instruction insertion for applications on SPM-based architectures can be done extremely efficiently by programmers Eichenberger *et al.* (2006); O'Brien (2007), it's not trivial though. With the increasing software complexity, as well as the diversity of the underlying architectures, the burden on developers becomes heavier. Now they must explicitly manage data in the program, on top of worrying about the functional correctness of the program - which is already quite complicated. Developers could have a very good understanding of when a data is needed, yet the local scratchpad memory is limited and the required space of the program can be input dependent. Shown in Figure 3.2, the compiled assembly code and all data of the application share the whole local scratchpad memory, in which the area below _end is the code and global data sections of the program, and the top is dynamic storage. The dynamic storage is occupied by stack data and heap data created with *malloc*. Stack data grows downward (from high addressed memory to low addressed memory), and heap data grows upward towards stack. Their sizes increase and de-

crease in the whole execution time. Because the local memory is limited and lacks hardware-enabled protection, it is possible that stack data and heap data could overflow the space and therefore corrupt the program's code or data or both. This often leads to hard-to-debug problems as the effects of the overflow are not likely to be observed immediately. Now, developers 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 not trivial for C/C++ programs, since stack size and heap size may be variable and input data dependent. The difficulty of programming these SMM architectures requires automated techniques to understand the application and insert data management instructions automatically.

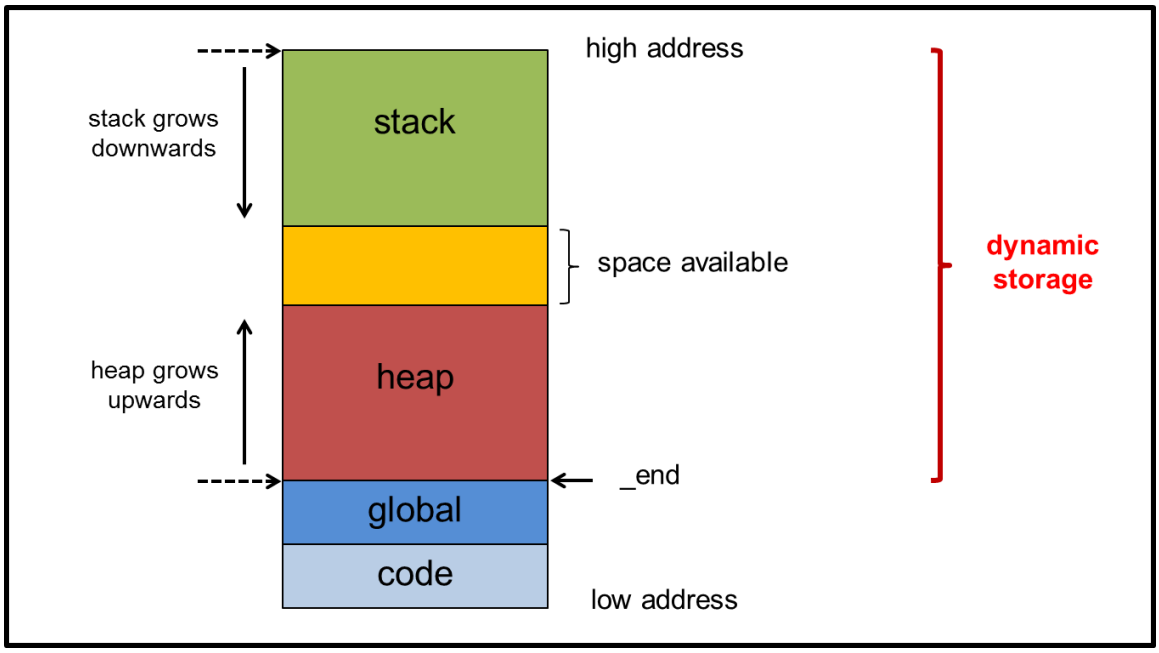

Figure 3.2: The Virtual Anatomy of the Compiled Program in Local Scratchpad Memory

3.2 Challenge of Software Branch Hinting on SMM

In software branch hinting processors, the application makes use of branch hint instructions to indicate that the branch instructions at specified PC addresses will jump to specified target addresses. After executing a hint instruction, the hardware starts to speculatively execute target instructions when the specified branch instruction is executed. It will harm the running time if the branch hinting instructions are not efficiently inserted. For software branch hinting to work best, there are two fundamental considerations. One is to estimate the taken probabilities of branches, and the other one is to find the locations in the code for branch hint instructions to minimize branch penalty. The first problem is significantly important because branch hint instructions should be inserted for only heavily taken branches. Though this problem has been extensively studied Ball and Larus (1993); Wu and Larus (1994); Wagner *et al.* (1994), the second problem, to insert branch hint instructions to minimize branch penalty, is rather unexplored. Even if we know the taken probabilities of all the branches, minimizing branch penalty by means of branch hint instructions is not trivial. The reasons origin from two constraints of the software branch hinting architectures. Firstly, for a branch hint to be effective, there must be some separation between a branch and its hint. The hint instruction must be executed several instructions earlier than the branch. Secondly, only a limited number (one for the IBM Cell processor) of branch hints can be active at any given time. For example, if two branches are too closely located in the control flow, the second branch cannot have enough separation. To hint the second branch, its hint needs to be placed above the first branch, and this will overwrite the hint for the first branch. Thus, hints may conflict with each other, and reduce the achievable benefits. When developing applications for SMM architectures, we must take these into considerations.

Chapter 4

CONTRIBUTIONS OF THIS DISSERTATION

The contribution of this dissertation is in developing some processor management policies in software in order to enable SMM architectures. In specific, this dissertation:

- Develops a technique to create and insert the branch hinting instructions to reduce the branch penalty (Chapter 5). It i) constructs a branch penalty model for compiler, ii) formulates the problem of minimizing branch penalty using branch hinting and iii) proposes a heuristic to solve this problem. The heuristic is based on three basic techniques: NOP padding, hint pipelining, and nested loop restructuring. Experimental results on several benchmarks show that our solution can reduce the branch penalty as much as 35.4% over the previous approach.

- Manages stack data of the application efficiently and seamlessly (Chapter 6.1). It first formulates the problem of stack data management optimization on an SMM core, and then develops both an ILP and a heuristic - SSDM (Smart Stack Data Management) to find out where to insert stack data management calls in the program. Experimental results demonstrate SSDM can reduce the overhead by 13X over the state-of-the-art stack data management technique Bai *et al.* (2011).

- Formulates formal code management problem for SMM architectures for the first time (Chapter 6.2). Then, two polynomial time heuristics for Code Mapping on Software Managed multicore systems (named as CMSM and CMSM_advanced) are proposed Lu *et al.* (2015). Experimental results demonstrate that the heuris-

tics can reduce runtime in more than 80% of the cases, and by up to 20% on our set of benchmarks, compared to the state-of-the-art code assignment approach Jung *et al.* (2010).

- Implements a framework to manage heap data for SMM architectures (Chapter 6.3). It consists of a modified compiler and a runtime library Lin *et al.* (2019). The runtime library consists of three generic optimizations (compile-time heap access detection, simplified management framework, and combined management calls). The experimental results show that the execution time is reduced by 80% on average.

4.1 Publications and My Contributions in the Publications

- (CODES+ISSS 2011) Branch Penalty Reduction on IBM Cell SPUs via Software Branch Hinting. I defined the problem, designed the algorithm, did all the experiments and wrote most part of the paper.

- (DAC 2013) SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs). I defined the problem, designed and implemented the algorithm, conducted some of the experiments, and participated in the paper writting.

- (CODES+ISSS 2013) CMSM: An Efficient and Effective Code Management for Software Managed Multicores. I defined the problem, designed the algorithm, participated in the experiments and paper writing.

- (TECS 2015) Efficient Code Assignment Techniques for Local Memory on Software Managed Multicores. I defined the problem, designed the algorithm, participated in the experiments and paper writing.

- (VLSID 2019) Efficient Heap Data Management on Software Managed Many-core Architectures. I Contributed the idea, participated the discussions and wrote the paper.

Chapter 5

SOFTWARE BRANCH HINTING FOR SMM

5.1 Overview

One of the critical limitations of pipelined modern computer architectures is the branch penalty, which grows larger as the pipeline depths increase. To minimize the effect of branch penalties, target of the branch is predicted and instructions from there are speculatively fetched. This prediction is typically history based and hardware implemented. Because performance of pipelined processors is critically dependent on the accuracy of branch predictions, many processors use large Branch Target Buffers (BTBs) to store the results of previous branches, and use complex and often proprietary algorithms to predict the branch target Stephen *et al.* (2002); Jiménez and Lin (2001).

While branch prediction became the de-facto standard in processor architectures, power-efficiency became an increasingly important consideration in processor design. With the total power budget capped, more cores could only be added by reducing the power and the complexity of each core Gschwind *et al.* (2006); Hofstee (2005). Therefore, architects started looking at processor components that could be removed to simplify the cores, yet not lose too much on performance Agarwal and Levy (2007). For instance, in the power-efficient IBM Cell Synergistic Processing Units (SPUs) Kahle *et al.* (2005) (which is one of SMM architectures), architects decided to remove hardware branch predictor and used software branch hinting in the hope to recover lost performance Sinharoy and White (2005). This is significantly different from earlier architectures that supported software branch hinting, e.g., the Sun Niagara

Table 5.1: Branch Penalty Can Be Crippling in the Cell SPU in Absence of Any Branch Hints.

Benchmark	**Branch penalty**
cnt	58.5%
insert_sort	31.4%
janne_complex	62.7%
ns	50.9%
select	36.2%

Kongetira *et al.* (2005) and Intel Itanium Itanium (2007), in the sense that the Cell SPUs do not have any hardware branch predictor but solely rely on software branch hints. Table 5.1 shows the branch penalty (in terms of percentage of execution time spent in branch penalty) on some of our benchmark applications, which shows that branch penalty in the absence of any branch hints can be very significant.

In software branch hinted systems, the application may contain branch hint instructions which indicate that the branch instructions at specified PC addresses will jump to specified target addresses. After executing a hint instruction, the hardware starts to speculatively execute target instructions when the specified branch instruction is executed. There are usually two constraints of the given architecture. First, the system requires some separation between a branch and its hint to make a branch hint to be effective, and the hint instruction must be executed several instructions earlier than the branch. Second, only a limited number (one for the IBM SPU) of branch hints can be active at any given time. For example, if two branches are too closely located in the control flow, the second branch cannot have enough separation. To hint the second branch, its hint needs to be placed above the first branch, and this will overwrite the hint for the first branch. Thus, hints may conflict with each other, and reduce the achievable benefits.

This dissertation present work on minimizing branch penalty in processors with software branch hinting. It firstly constructs a branch penalty model for the compiler (Section 5.3), in which branch penalty is expressed as a function of number of instructions between hint and branch instruction, branch probability, and the number of times a branch is executed. Secondly, three fundamental approaches is proposed to hint branch instructions (Section 5.5): 1) NOP padding scheme finds out the number of NOP instructions needed between a branch and its hint to maximize profit. 2) the hint pipelining technique enables hinting branches that are very close to each other, and 3) the nested loop restructuring technique allows us to change the loop structure to increase the effectiveness of branch hinting. Eventually, a heuristic that applies the above three methods to the code prudently to minimize overall branch penalty is present (Section 5.5.4).

5.2 Branch Hinting Mechanism

Figure 5.1 elaborates the software branch hinting mechanism. The description in this section can perfectly explain the behavior of branch hint instructions. Just like hardware branch predictors, software branch hinting mechanism also requires a Branch Target Buffer (BTB). When a hint instruction is executed, the BTB entry is first updated, and then target instructions are loaded to the Hint Target Buffer from the specified target address. The hardware usually fetches instructions from Inline Prefetch Buffer which is constantly loaded with the sequential instructions according to PC address. When a branch instruction is fetched, the PC address is compared with the branch address in the BTB entry. If it matches, the instructions are fetched from Hint Target Buffer instead of Inline Prefetch Buffer. As compared to BTBs in hardware branch prediction, the BTBs to support software branch hinting should be typically much smaller and simpler, without having to store the history of branches.

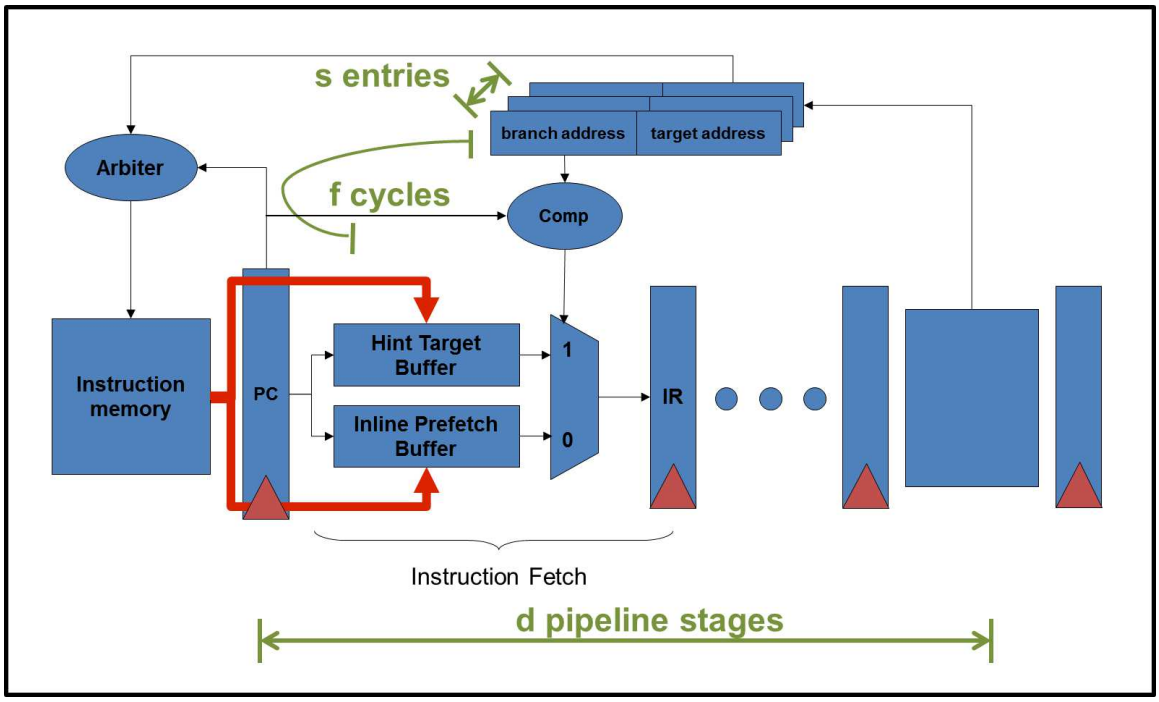

Figure 5.1: Software Branch Hinting is Characterized by Hint Instructions Setting the BTB Entries. It Has 3 Key Parameters, 1) d, the Number of Pipeline Stages Where BTB is Set, 2) f, The Time To Fetch Target Instructions, and 3) s, the Number of Entries in BTB.

While the actual design for SMM architectures may vary, there are three fundamental parameters of any software branch hint implementation. The first parameter is d, which is the number of pipeline stages, where the branch hint is executed and BTB entry is set. This implies that, if the separation between the branch hint and branch instruction is less than d cycles, then the fetch stage will not even recognize that there is a hint to this branch, and the default prediction (typically, "not taken") will happen. For instance, d of SPUs is 8. The second parameter is f, which is cycles to fetch target instructions. After a BTB entry is set, a request is made to the arbiter to fetch the target instructions from memory into the Hint Target Buffer. Note that f may not be statically known, since the delay to get target instructions from the memory depends on the availability of the memory bus. Consequently, in order to completely avoid branch penalty, the separation between a branch hint and branch

should be at least $d+f$. This is termed as *separation constraint*, and it is 11 in SPUs. If the branch and branch hint are separated by more than separation constraint, then there is no penalty for a correctly hinted branch. However, if the separation between the branch and branch hint is less than $d+f$, but greater than d, say d', then a correctly hinted branch will incur a branch penalty of $d+f-d'$. During the hint stall, the branch instruction is stalled before going into execution pipeline. Therefore, even if the hint is incorrect, the comparison between hinted target address and the actually calculated target address, namely branch resolution, can only take place after actually executing the branch instruction. Thus, on top of the branch penalty, the time to wait for the target instructions to be loaded is added to misprediction penalty. This should be the same as $d+f-d'$ from the above. The third parameter is s, which is the number of entries in the BTB. A N-entry BTB would imply that N branches can be hinted at the same time. Note that along with the size of BTB, s also impacts the size of Hint Target Buffer, which must be large enough to hold target instructions for all the BTB entries. s is expected to be a small number in order to keep the software branch hinting mechanism power-efficient. For example, SPUs on the IBM processor have one-entry BTB, making $s = 1$.

5.3 Branch Penalty Model

In order to implement any software branch hinting algorithm, the penalty of a branch has to be modeled as a function of separation in terms of the number of instructions, the branch probability, and the number of times the branch is executed, which are all the information a compiler can have.

As the IBM Cell processor is the only SMM architecture that has fully software branching hint feature, branch penalty model is proposed upon it. However, similar branch penalty model may apply to other SMM architectures with software branch

hint. To do this, several experiments are conducted in which we run a synthetic benchmark composed of a branch, and branch hint, separated by a varying number of `lnop` instructions. In each case, some more `lnop` instructions above the hint are inserted to keep the total number of `lnop` instructions as 18. We plot the execution time (in cycles) of the benchmark as we change the "separation" between the branch and the hint. The execution time is measured using spu decrementer IBM (2007). Since the granularity of timing measured by spu decrementer is hundreds of cycles, the branch and hint are put in a loop and the loop is executed hundreds times to enlarge the granularity of time measurement.

`lnop` is inserted so that the execution time is not affected by the dual-issue nature of the SPU. SPU is a dual-issue core, and has two unbalanced execution pipelines, named *even* and *odd*, and each of them can execute a disjoint set of instructions. Even pipeline can only execute floating point or fixed point arithmetic operations while odd pipeline can only execute memory, logic, flow-control instructions, including branch and branch hint instruction. Instructions are dual-issued only when i) two instructions are issuable and aligned at an even word address, ii) the first instruction can be executed on even pipeline, and iii) the second instructions can be executed on odd pipeline. There are two NOP instructions, `nop` and `lnop`, in SPUs. `nop` is executed in even pipeline, and `lnop` in odd pipeline. By having only control flow instructions (branch and branch hint) and `lnop`, the SPU is effectively made single-issue.

Figure 5.2 shows branch penalty plot when the hint is correct (namely, the branch is taken). When separation is less than 8 instructions, the hint is not recognized and we have branch penalty of 18 cycles. After that, the branch waits for the target instructions to be loaded. The penalty decreases with the increase of separation, because executing NOP instructions is now hiding the latency of fetching target instructions. When the separation becomes larger than 19 instructions, the branch

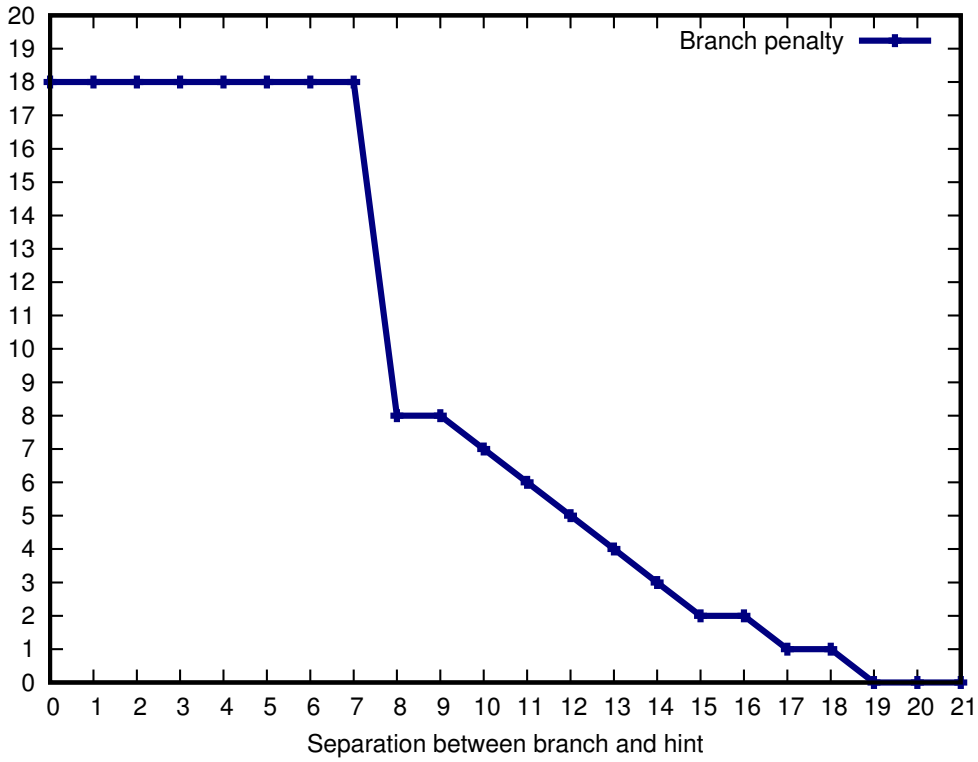

Figure 5.2: Branch Penalty is Plotted as We Increase the Separation When Hint is Correct. We Need at least 8 Instructions for a Hint to Become Effective, and the Penalty Decreases as Separation Increases.

penalty can be fully eliminated. The following is the empirical branch penalty model when hint is correct.

$$Penalty_{correct}(l) \approx \begin{cases} 18, & if \;\; l < 8 \\ 18 - l, & if \;\; 8 \leq l < 19 \\ 0, & if \;\; l \geq 19 \end{cases} \tag{5.1}$$

where l is the separation in the number of instructions.

Figure 5.3 shows the same experiment result except that hint was incorrect (namely, misprediction penalty when the branch is not taken). As expected, when the separation is less than 8, there is no penalty because the architecture assumes branches to be not taken by default. When the separation is greater than 18 instructions, misprediction leads to 18 cycles of branch penalty. Interestingly, when the separation

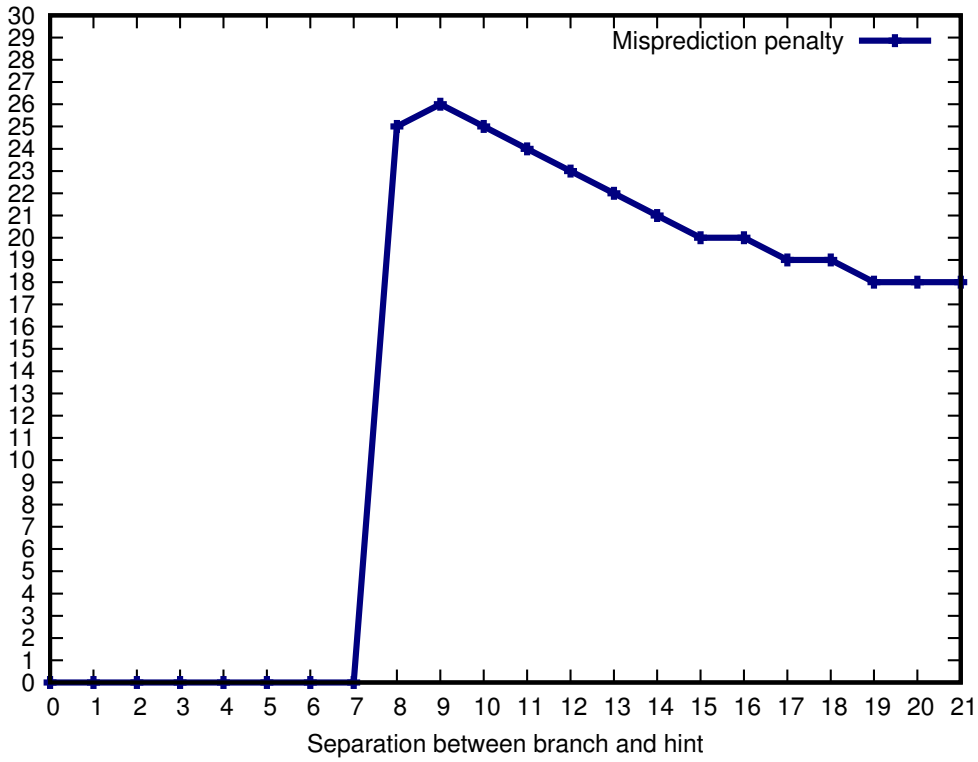

Figure 5.3: Misprediction Penalty is Plotted as We Increase the Separation. On Top of Branch Penalty, the Time to Flush Pipeline is Added Resulting Larger Branch Penalty Than When Hint is Correct.

is between 8 to 18 instructions, the misprediction penalty is greater than 18 cycles and decreases as the separation increases. This means that the branch still waits for target instructions to be fetched, even though the branch is not taken. Thus, branch resolution occurs after target instruction arrives, and this makes incorrect hints more detrimental to performance. Our empirical branch penalty model when hint is incorrect is as follows.

$$Penalty_{incorrect}(l) \approx \begin{cases} 0, & if \;\; l < 8 \\ 36 - l, & if \;\; 8 \leq l < 19 \\ 18, & if \;\; l \geq 19 \end{cases} \tag{5.2}$$

where l denotes the separation in the number of instructions.

Overall, the penalty of a hinted branch is the sum of Equation 5.1 and 5.2. Consid-

ering branch probability and the number of times the branch is executed, the branch penalty can be calculated as follows.

$$Penalty(l, n, p) = Penalty_{correct}(l) \times np + Penalty_{incorrect}(l) \times n(1 - p) \tag{5.3}$$

where n and p are the number of times the branch is executed, and the branch probability respectively.

5.4 Problem Formulation

Two fundamental problems of minimizing overall branch penalty using software branch hinting are to find i) a set of branches to be hinted, and ii) a set of assembly program locations where the hints for those branches should be placed. First of all, branch probabilities and frequencies should be obtained. This is because, as discovered in Section 5.3, when a branch is not taken, hinting the branch will not only increase the instruction count, but also lead to a larger branch penalty causing a significant performance degradation. The problem of finding branch probabilities has been well-studied for decades, and improving the state of the art is not the intent of this dissertation. This dissertation adopt the static estimation technique Ball and Larus (1993); Wu and Larus (1994) embedded in GCC compiler, but any branch probability estimation technique can be used.

Even if the probabilities are known, and branches that benefit by hinting are identified, it is rarely possible to hint all of them. It is primarily because of the separation constraint that is architecture dependent. In an architecture with s size BTB, only s branch hints can be active at any point of time. In SPUs, only one hint can be effective at any point of execution, which means when two branches are located too close to each other, only one branch can be hinted. To overcome this problem, three methods are present to enable hinting more branches later in this

dissertation. Since our technique involves restructuring of basic blocks, the control flow of the program may change after applying the proposed technique. However, the program semantic will stay the same. Now, the problem can be formulated as follows.

- **Input:** A program which can be represented in Control Flow Graph, and branch probabilities and frequencies of the branches.
- **Output:** A new program with branch hint instructions. The program may have a different control flow, but the semantic should remain the same.
- **Objective:** Minimize branch penalty.
- **Constraint:** For every pair of a hint and branch, separation must be at least d cycles. Also, only s hints can be effective at any point of time, so the lifetime of more than s hints should not overlap. d and s are architecture dependent.

5.5 Branch Hint Management

In this section, three basic techniques are present to enable hinting more branches: NOP padding, branch pipelining, and basic block restructuring. In addition, we analyze the conditions when the application of each method can be beneficial to performance. Lastly, we will present a heuristic that combines three schemes and applies each method prudently.

5.5.1 NOP Padding

When there is not enough separation to be accommodated, NOP instructions are inserted to artificially increase separation as shown in Figure 5.4. In the figure, NOP padding increases the separation to 8 instructions making the hint to be effective. Let us assume this branch is taken always. Using the branch penalty model aforementioned, the penalty drops from 18 to 10 cycles. With the help of dual-issue, inserted

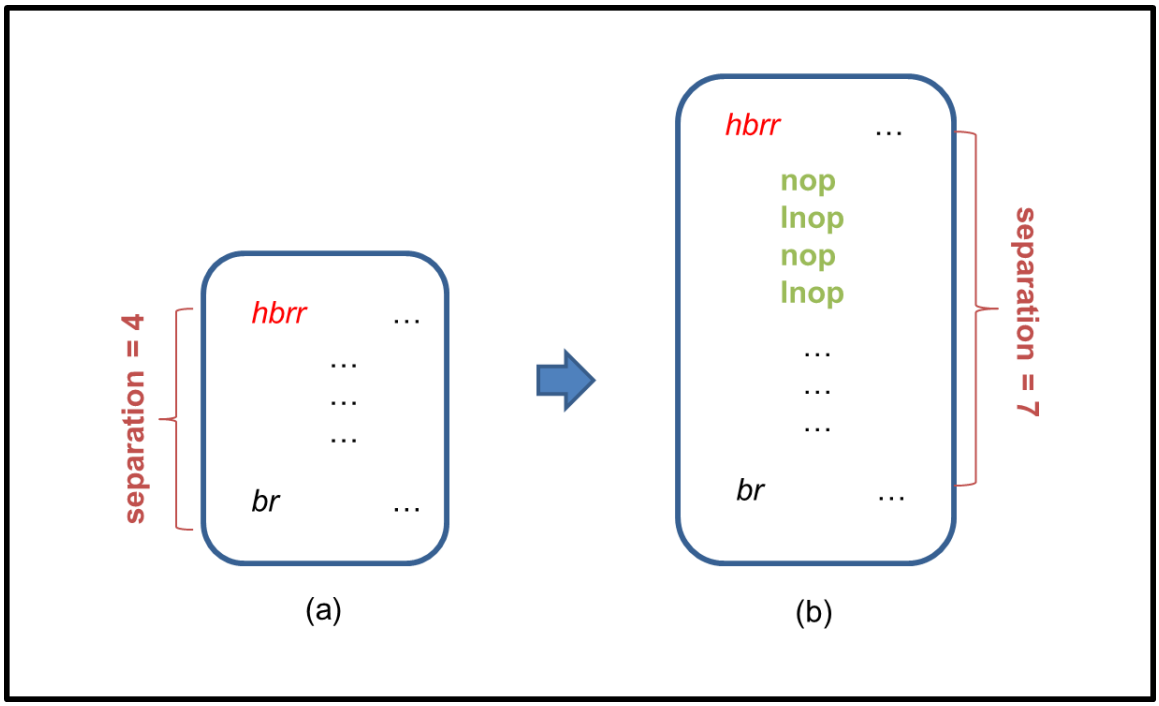

Figure 5.4: (a) Before NOP Padding, the Branch Cannot be Hinted. (b) NOP Padding Enables Hinting the Branch.

2 `nop`-`lnop` pairs can be executed in 2 cycles, and therefore the total performance improvement is 6 cycles.

GCC compiler included in IBM Cell SDK also inserts `nop` instructions only when user explicitly specifies the maximum number of `nop` instructions to be inserted gcc (2005). We insert both `nop` and `lnop` to minimize the overhead of executing additional instructions. GCC can also inserts `nop` instructions when a user-specified option is given. It inserts whenever the branch cannot have enough separation, but in reality, NOP padding may not be always profitable. This will be shown in our experimental results.

On the other hand, in this dissertation, the performance gain of NOP padding is analyzed using the proposed branch penalty model. NOP padding is used not only to enable a branch to be hinted but also to increase the performance gain, so called profit of hinting the branch.

Let l, n, and p denote the original separation before padding, number of times the branch is executed, and the branch probability respectively. The branch penalty before applying padding can be calculated as follows.

$$Penalty_{no\_pad} = Penalty(l, n, p)$$

Since l is less than the constraint, hint does not work and the penalty is $18np$.

The branch penalty after applying padding is modeled as follows with the separation increased by the number of NOP instructions.

$$Penalty_{pad} = Penalty(l + n_{NOP}, n, p)$$

where n_{NOP} represents the number of inserted NOP instructions.

Because the branch is taken n times, the hint instruction and the inserted NOP instructions are also taken n times. A pair of `nop` and lnop instructions can be executed in one cycle with a help of dual issue. Then, the overhead of NOP padding can be modeled as the following.

$$Overhead_{pad} = n(n_{NOP} + 1)/2$$

Combining all of the above, the performance improvement by NOP padding can be modeled as follows. NOP padding is applied only when it is profitable.

$$Profit_{pad} = Penalty_{no\_pad} - Penalty_{pad} - Overhead_{pad} \tag{5.4}$$

5.5.2 Hint Pipelining

As shown in Figure 5.5 (a), a compiler may try to hoist the hint for branch b_2 above branch b_1 to increase separation. This will lose any opportunity to hint branch b_2, and this is another common performance limiting factor in GCC. In this case, GCC will simply give up hinting b_1 since b_2 has more priority (Otherwise, GCC would not have tried to host the hint in the first place.)

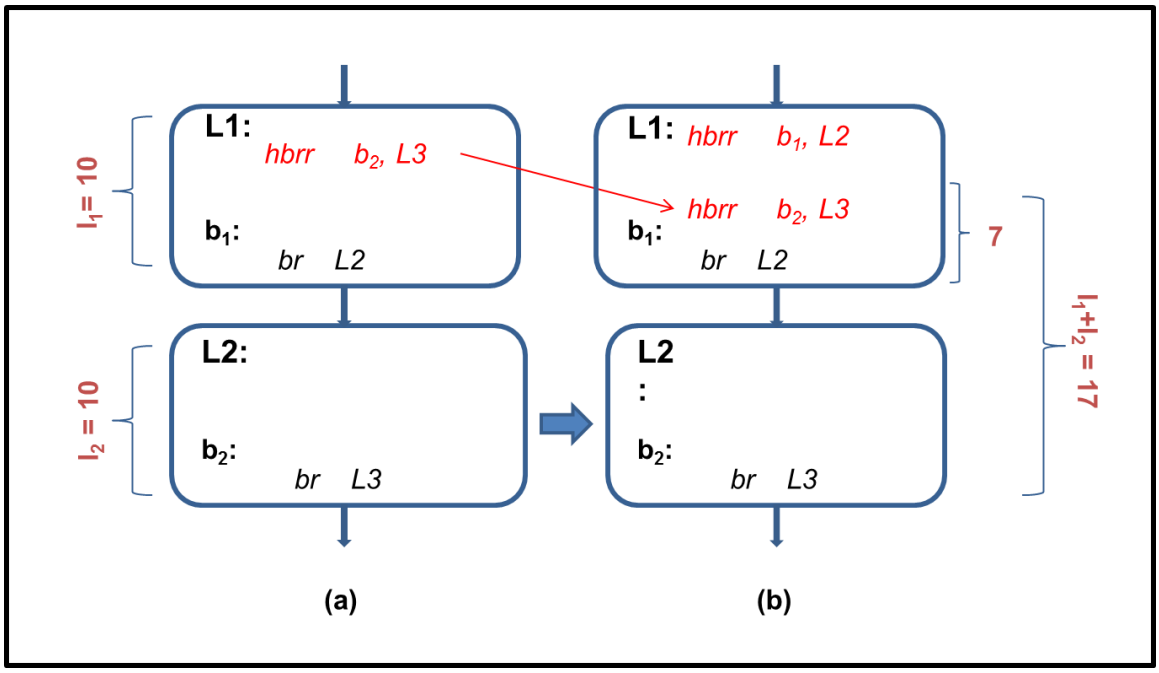

Figure 5.5: (a) Before Hint Pipelining, Branch b_1 Cannot Be Hinted Due to the Hoisted Hint for b_2 (b) After Hint Pipelining, Both b_1 and b_2 are Hinted.

To overcome this problem, let us consider the fact that a hint instruction is not recognized when the separation is less than 8 instructions. This gives us an intuition that we can insert multiple hints for multiple branches in a pipelined fashion. Figure 5.5 (b) shows how hint pipelining works. If the hint for branch b_2 is placed less than 7 instructions ahead of branch b_1, the hint will have not yet recognized when branch b_1 is executed. As a result, both b_1 and b_2 can be hinted since the later execution of the hint for b_2 will not affect the previous executed hint for b_1.

The above example is also used to show how to analyze the profit of hint pipelining. In this case, the profit can be modeled as the decrease of branch penalty of newly hinted branch b_1 minus the possible increase of branch penalty of b_2. Let l_x represent the number of instructions in basic block Lx. The path from $L1$ to $L2$ is only taken when the branch b_1 is not taken. Thus, when the branch b_1 is taken, the branch b_2 is not hinted. The branch penalty before applying hint pipelining can be modeled as

sum of the penalty of two branches as follows, and the penalty of not hinted branch is modeled as the case when separation is zero.

$$\begin{aligned} Penalty_{no\_pipeline} =& Penalty(0, n_1, p_1) + (1 - p_1) \cdot Penalty(l_1 + l_2, n_2, p_2) \ + \\ & p_1 \cdot Penalty(0, n_2, p_2) \end{aligned}$$

where p_x and n_x denote the branch probability of branch b_x and the number of times b_x is executed.

After hint pipelining, both b_1 and b_2 can be hinted. The maximum possible separation for the hint for b_2 is decreased from $l_1 + l_2$ to l_2, which possibly increases branch penalty of b_2, but another branch b_1 can be hinted instead. Since our heuristic starts inserting hint instructions from bottom basic blocks, when this analysis is being done, the hint for branch b_1 is not yet inserted. We always assume that b_1 will be hinted at the top of $L1$, even though it can be hinted farther above, possibly reducing more branch penalty. The penalty after applying hint pipelining is modeled as follows.

$$\begin{aligned} Penalty_{pipeline} =& Penalty(l_1, n_1, p_1) + (1 - p_1) \cdot Penalty(7 + l_2, n_2, p_2) + \\ & p_1 \cdot Penalty(0, n_2, p_2) \end{aligned}$$

Note that is only applied when $l_1 \leq 8$. The above calculation is an example when a hint is hoisted to the immediate predecessor. A similar analysis can be done to any other cases.

The overhead of hint pipelining is the number of times the hint instruction is executed. When the hint is in basic block Lx, it is executed n_x times. Then, the overall overhead is the difference of execution counts as shown below.

$$Overhead_{pipeline} = n_2 \ - \ n_1$$

Hint pipelining is applied only when the overall profit of it is greater than zero,

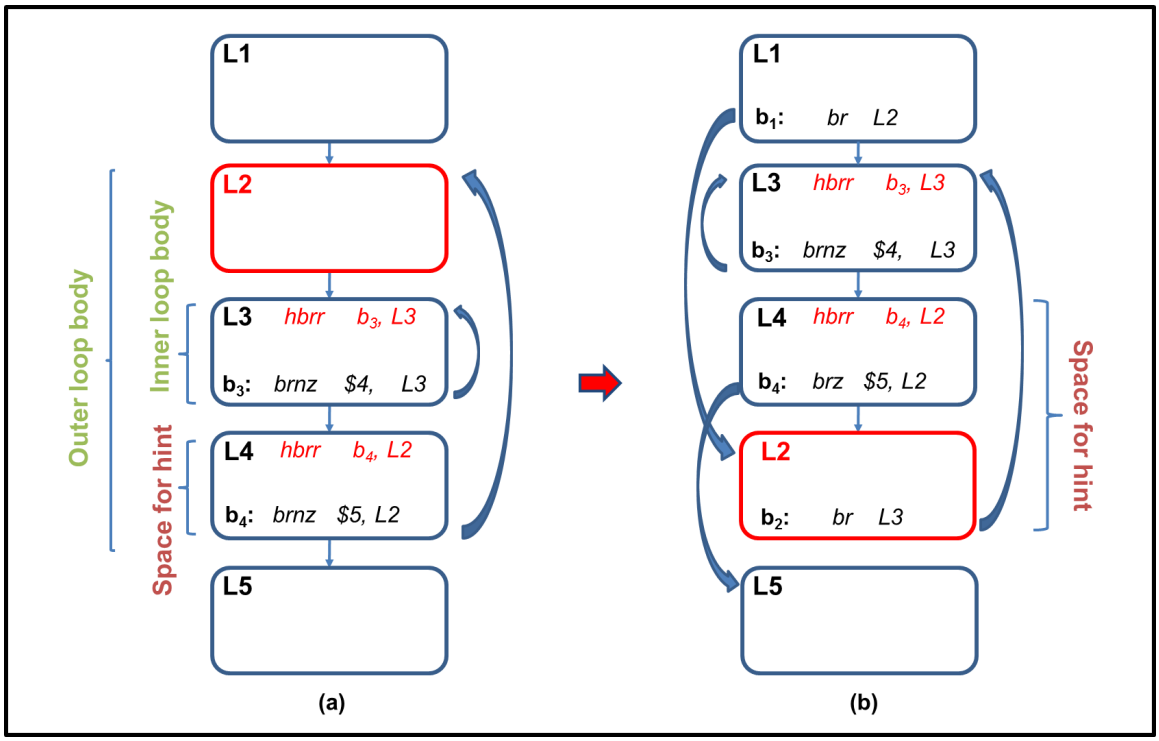

Figure 5.6: (a) Before Nested Loop Restructuring, the Separation for b_4 is Limited to l_4. (b) After Nested Loop Restructuring, the Outer Loop Branch is Changed to Unconditional Branch b_2, and the Separation is Increased to $l_2 + l_4$.

which can be modeled as the following.

$$Profit_{pipeline} = Penalty_{no\_pipeline} - Penalty_{pipeline} - Overhead_{pipeline} \tag{5.5}$$

5.5.3 Nested Loop Restructuring

The branch penalty from loops is of paramount importance, since even a small penalty can be accumulated for the whole iteration and significantly impact performance. In this section, a method specially developed for nested loops is present. This scheme is motivated by our observation that usually in nested loops, only innermost loop branch can be hinted, and the outer loop branch cannot be hinted due to separation constraint.

As summarized in Figure 5.6, the structure of nested loop can be changed so that the space to insert a hint for the outer loop branch is enlarged. Throughout the

dissertation, loop branches are assumed always be at the bottom of the loop body. In Figure 5.6 (a), let us suppose the size of basic block $L4$ is too small to hint the branch b_4. Figure 5.6 (b) presents our solution in which basic block $L2$ is moved after $L4$, and two unconditional branch b_1 and b_2 are introduced. In addition, the target address of branch b_4 is changed to $L5$, and the branch condition is flipped. This technique is applied before any hints are inserted into the code, and here the hint for b_3 is assumed to be placed in $L3$.

The same example is used to illustrate the calculation of the profit of nested loop restructuring. Before applying restructuring, the overall branch penalty is the sum of branch penalties of b_3 and b_4. In this example, l_4 is smaller than 8 instructions, so the branch b_4 will not be hinted.

$$Penalty_{no\_reorder} = Penalty(l_3, n_3, p_3) + Penalty(l_4, n_4, p_4)$$

After applying restructuring, the outer loop branch is changed to unconditional branch b_2 and it has separation of $l_2 + l_4$. We may get more profit from this, but this introduces branch b_1 which will be taken only once when entering the loop. Also, the branch condition of b_4 is changed, so it is taken only once when exiting the loop. We assume that b_1 and b_4 are not hinted incurring 18 cycles of penalty for each. The penalty becomes the sum of branch penalties of b_1, b_2, b_3, and b_4. Note that the path probability for $L4$ to $L2$ is one since the branch will always fall through except when the loop terminates.

$$Penalty_{reorder} = 18 \ + \ Penalty(l_2 + l_4, n_2, p_2) \ + \ Penalty(l_3, n_3, p_3) \ + \ 18$$

The overhead of this technique is the difference of the numbers of times hint instructions are executed. In this particular example, the hint for b_4 could not be inserted at first due to separation constraint, but now it is inserted into $L4$. However,

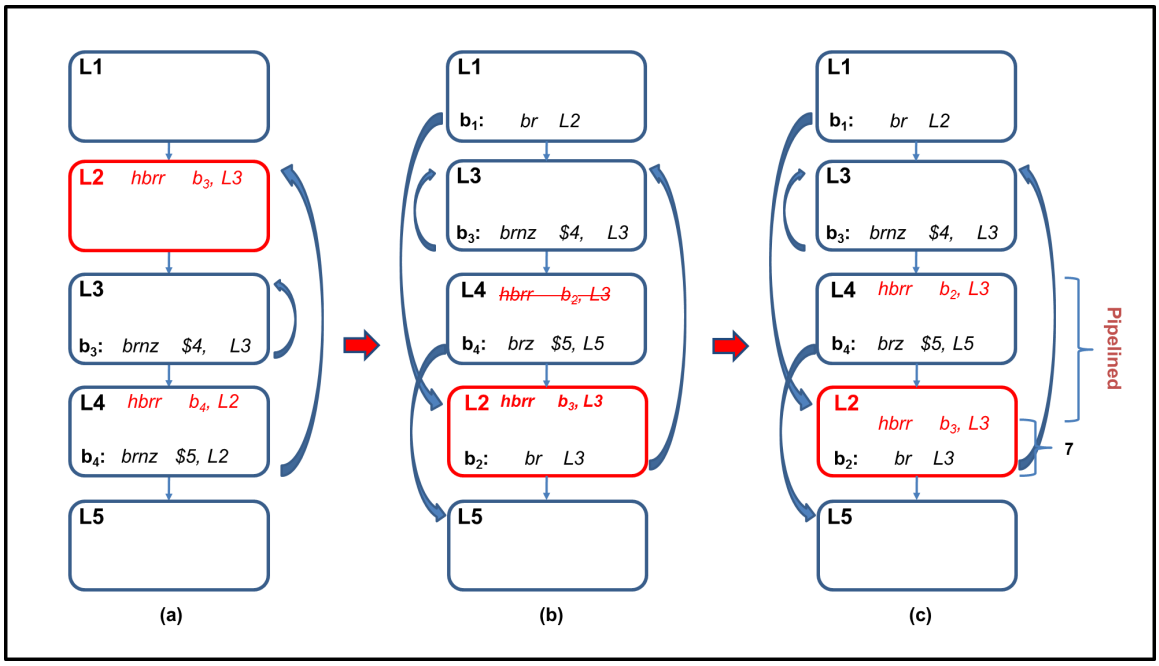

Figure 5.7: (a) Hint for b_3 Can be Hoisted into $L2$. Branch b_4 Cannot be Hinted Even After Loop Restructuring. (b) After Restructuring, The Hint for b_3 Cancels the Hint for b_4. (c) Pipelining is Applied and Both Branches Can be Hinted.

in general, the nested loop restructuring can be used to improve the profit of b_4 even if l_4 is greater than eight instructions. In this case, the overhead is considered as zero because the hint instructions are not moved to other basic blocks.

$$Overhead_{reorder} = \begin{cases} n_4, & if \ \ l_4 < 8 \\ 0, & otherwise \end{cases}$$

Nested loop restructuring is only applied when the overall profit of it is greater than zero, which can be modeled as the following.

$$Profit_{reorder} = Penalty_{no\_reorder} - Penalty_{reorder} - Overhead_{reorder} \tag{5.6}$$

Note that in the above example, without loss of generality $L3$ can denote a loop body containing multiple basic blocks. This is because the intention of nested loop restructuring is to give more separation to outer loop branch, and the inner loop is not affected. For the loop body which does not have any likely-taken branches (for

example, function call or if-then-else), we hoist the hint for the loop branch to the loop-initialization block, which is executed only once. This is to reduce the overhead of repetitive execution of the hint instruction. Figure 5.7 (a) shows an example where the hint for inner loop can be hoisted to outer loop body. After applying restructuring, the hint for b_3 is hoisted to $L2$ and cancels previously inserted the hint for b_2, as shown in Figure 5.7 (b). Instead of canceling the hint for b_2, we can apply pipelining to hint both branches. Figure 5.7 (c) shows the final solution. We check the structure of the inner loop body, and if the hint can be hoisted, we assume it pipelining will be applied later. To determine its profitability, a similar analysis to the above can be done assuming the hint for b_3 will be placed seven instructions above b_2.

This abstraction of loop body enables us to apply this technique to all kinds of loop nests. For loop nests whose depth is more than two, this technique is recursively applied from the innermost loop to the outermost loop. For example, let us suppose we have three loops $L1$, $L2$, and $L3$. $L1$ is the innermost loop, and $L3$ is the outermost loop. $L1$ and $L2$ are first considered as restructuring candidates, and we check the profit of restructuring two loops. If those two loops are reordered, they can be considered as one loop body. Then, either the reordered loop body or $L2$ can be considered as restructuring candidate with the next outer loop $L3$. Also, if there is more than one loop a loop, all of the inner loops can be considered as one loop body. This restructuring may result in severe instruction cache misses in conventional machines, but it is not the case in software branch hinting because instructions are explicitly prefetched by branch hint instructions.

5.5.4 Branch Penalty Reduction Heuristic

Given all schemes and their applicable conditions we discussed above, this section presents a heuristic that combines all of them. It requires the information of all

Algorithm 1 Branch Penalty Reduction Heuristic

```
1: Apply nested loop restructuring for all loop nests;
2: b = last basic block in the program;
3: while basic block b is not the first basic block do
4:     h = b;
5:     while Basic block h is not the first basic block do
6:         if Branch in h->predecessor is likey-taken then
7:             break;
8:         end if
9:         h = h->predecessor;
10:    end while
11:    insertHint(b, h);
12:    if b != h then
13:        b = h;
14:    else
15:        b = b->predecessor;
16:    end if
17: end while
```

nested loops, the branch probabilities, and the number of times each branch is taken as input.

Algorithm 1 shows the complete heuristic. The algorithm starts with applying nested loop restructuring to all loop nests to increase the possible separation for loop branches. Then, it starts inserting hint instructions from the bottom basic block. For a branch, the procedure tries to hoist its hint to the predecessor basic blocks, scanning predecessor basic blocks recursively. If there is a branch in the predecessor and it is likely-taken, it stops going into predecessors and returns the current basic block. Then, the procedure *insertHint()* (shown in Algorithm 2) inserts a hint instruction in the current basic block. It checks the separation and applies NOP padding and hint pipelining when applicable.

Algorithm 2 insertHint(b, h)

```
1: // Insert a hint instruction for the branch in basic block b into basic block h.
2: if h contains a branch then
3:     if Pipelining is profitable then
4:         Insert a hint instruction for the branch in b in a pipelined mode;
5:     else
6:         if NOP padding is profitable then
7:             Insert as many NOP instructions as it is profitable;
8:         end if
9:         Insert a hint instruction for the branch in b;
10:    end if
11: else
12:    if NOP padding is profitable then
13:        Insert as many NOP instructions as it is profitable;
14:    end if
15:    Insert a hint instruction for the branch in b;
16: end if
```

5.6 Related Work

Software branch hinting has been present in processors for decades, it has not been an active area of research, however. This is mainly because it has always been in addition to the hardware branch prediction, and in this situation, branch hinting can only improve upon the performance of hardware branch prediction, and the scope of improvement was minimal. The Cell processor, which does not have any hardware branch prediction and relies solely on software branch hinting to avoid branch penalty, changes all that. Severe performance degradation is observed if the system is without any branch hints. In such SMM like architectures, software branch hinting is no longer optional, but has become mandatory!

In processors with only software branch hinting, branch penalty can be reduced by predication Kalamatianos and Kaeli (1999) (if supported), i.e., executing both possible execution paths. Loop unrolling IBM (2007) can also reduce branch penalty by reducing the number of times branches are executed. Our focus is orthogonal,

in the sense that we intend to reduce branch penalty by hinting the likely-taken branches, by prudent placement of branch hints.

Recently, Briejer et al. studied the energy efficient branch prediction on Cell SPUs by modifying hardware Briejer *et al.* (2010). In their work, the performance and power trade-off of different hardware setups is studied where hardware branch predictor is present in conjunction with software branch hinting. Our techniques, on the other hand are completely in software, and do not require any hardware changes.

Two main problems exist in branch hinting to minimize branch penalty. One is to accurately estimate the taken probability of branches, and the other is to find prudent placement of branch hints to minimize the penalty. Researchers has been done on estimating taken probabilities of branches. A set of program-based heuristics, especially focused on non-loop branches, was proposed in Ball and Larus (1993). Another approach Wu and Larus (1994) estimates not only branch probabilities but also the execution frequencies of blocks and edges, including function calls, in Control Flow Graphs (CFGs). These techniques are already embedded in GCC compiler. The focus of this dissertation is the second problem.

GCC compiler in IBM Cell BE SDK gcc (2005); IBM (2009) has a heuristic to insert branch hint instructions to the code. We consider this as the closest related work. It works with a set of principles such as moving hint instructions outside the loops to reduce the overhead of executing hints repeatedly Eichenberger *et al.* (2005), and giving priority to hinting innermost loop branches, however, it suffers from several problems in effectively hinting branches. For instance, if two branches are close to each other, then only one of them is hinted, and in nested loops, typically only the innermost loop branch is hinted.

Our proposed technique alleviates some of the problems of GCC by carefully analyzing conflicting branches. It can hint them better through accurate cost functions,

and increase the opportunity of hinting low priority branches while keeping all the high priority branches hinted.

5.7 Experimental Results

5.7.1 Experimental Setup

The effectiveness of the proposed heuristic is validated using various benchmarks from Multimedia loops Kolson *et al.* (1996) and WCET benchmarks Gustafsson *et al.* (2010). The Software Managed Manycore (SMM) architecture we choose for demonstration is the IBM Cell processor Flachs *et al.* (2006). Our baseline is GCC compiler gcc (2005), which is included in IBM Cell SDK IBM (2009). It has a heuristic that inserts branch hint instructions to the code, which is designed and implemented by the manufacturer. All benchmarks are compiled with O3 optimization level. To measure the performance and the branch penalty of the program, the cycle accurate IBM SystemSim Simulator for Cell BE sys (2006) is deployed. As library functions (e.g., printf()) are not changed, all the measurements are done only on user codes. Branch probabilities and the cyclic frequencies of branches are obtained by a static analysis Ball and Larus (1993); Wu and Larus (1994), which is also implemented in GCC.

Figure 5.8 shows the percentage of branch penalties in the total program execution cycles after GCC inserts hints. The benchmarks are divided into two groups 'high' and 'low' according to the percentage of branch penalty in the total execution time. The benchmarks which have more than 20% of branch penalty are grouped as 'high', while the others fall under the group 'low'.

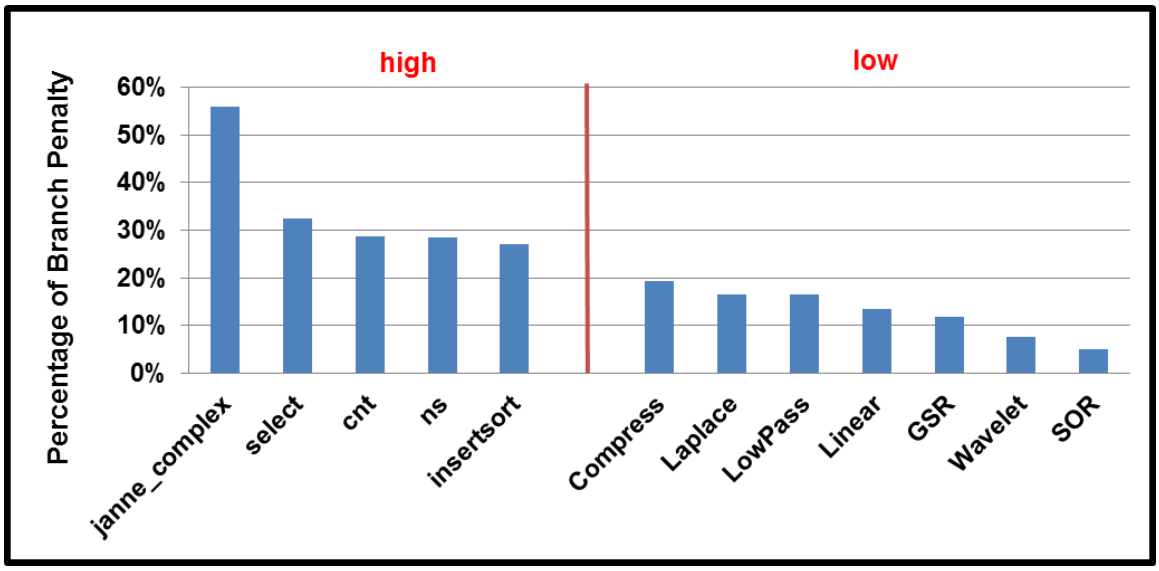

Figure 5.8: The Percentage of Branch Penalty in The Total Execution Cycles After GCC Inserts Hints Into the Program. Benchmarks are Grouped Into Two Groups 'high' and 'low' According to the Percentage.

5.7.2 Branch Penalty Reduction

The effectiveness of our heuristic can be shown as the reduction of branch penalty after applying our heuristic. Figure 5.9 shows the reduction in branch penalty cycles after applying our heuristic, compared to the GCC-inserted hints. Overall, we can reduce average 19.2% of the branch penalty more than GCC. Since we insert NOP instructions through our NOP padding technique, we consider the increased NOP cycles as part of branch penalty. SystemSim simulator can output NOP cycles separately as well as branch penalty cycles, and branch penalty in our results is the summation of branch penalty cycles and increased NOP cycles.

The proposed heuristic works more effectively for the benchmarks with deeply nested loops, such as **janne_complex**, **cnt**, **insertsort** and **ns**. As shown in Figure 5.8, GCC cannot reduce the branch penalty effectively in those benchmarks, and all of them fall under 'low' group. Figure 5.10 compares the code change in a deeply nested loop in benchmark **ns** after GCC and our heuristic. Loop branches are shown

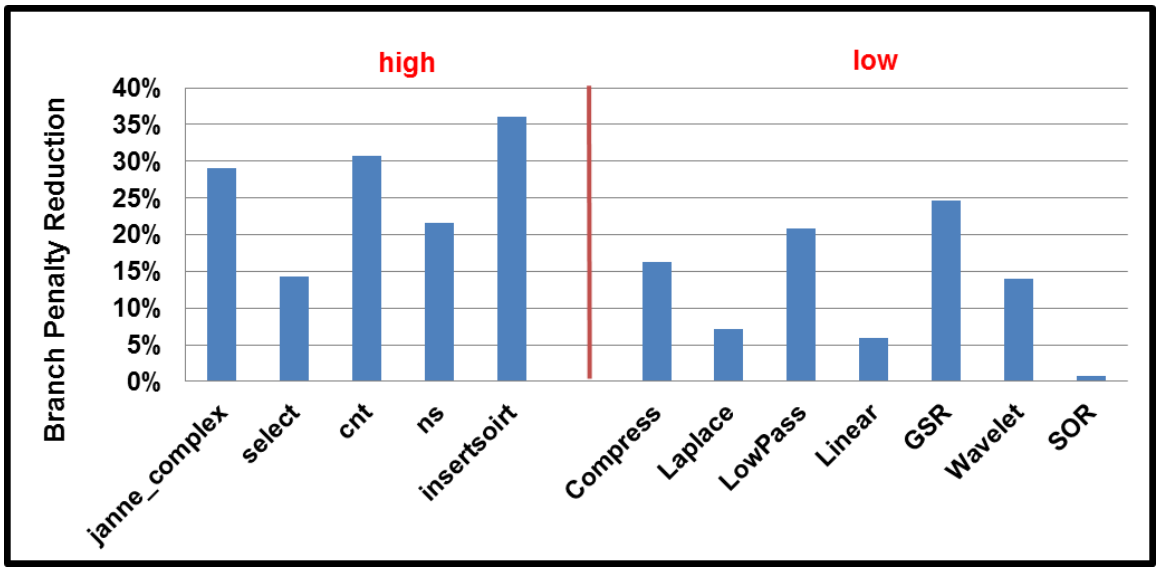

Figure 5.9: *Reduction of Branch Penalty is* 35.4% *at Maximum and* 19.2% *on Average.*

in RED arrows, while others are shown in GREEN arrows. GCC can only hint the loop closing branch for the innermost loop, and because of limited basic block sizes, all other loop branches cannot be hinted. In contrast, our technique can hint all of the loop branches.

Even with our scheme, the highest reduction of stall due to branch penalty is around 35%. There are three reasons why the branch penalty cannot be completely eliminated. First, not all branches can be hinted because only one hint can be active at a time for IBM Cell BE. When two branches are located too close to each other, only one of them can be hinted. Even though our techniques can enlarge the possible separation to enable more branches to be hinted, they cannot be applied to all cases. This is because each technique is applied only when it is profitable. Unless two or more branch hints are allowed to be active at a time, this problem cannot be ultimately solved. The second reason is that branch hinting works as a static branch predictor, while most of the branches are dynamically decided to be taken or not. Even though the penalty can be effectively avoided when branch is taken, there is

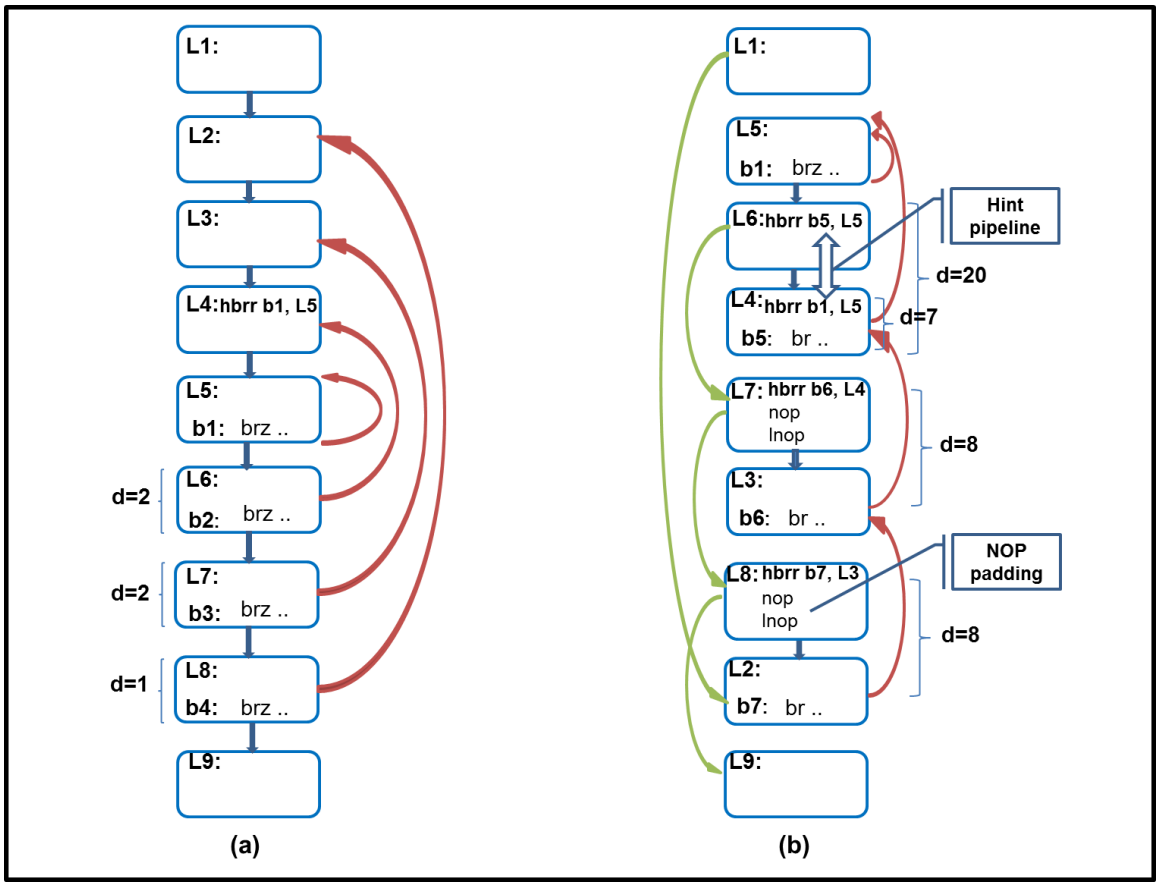

Figure 5.10: (a) GCC Can Only Hint the Innermost Loop Branch. (b) The Proposed Technique Can Hint All of The Four Loop Branches.

still misprediction penalty when the branch is not taken. As a result, unless a branch is heavily taken, to hint the branch may not be always profitable. A typical example is "if-then-else" branches in a loop. The worst case scenario is when the branch is taken for the half of the time. Penalty always exists whether or not we hint the branch, as long as the hint is static. If the compiler assigns the more-likely-taken execution path as a fall-through path, the penalty of "if-then-else" branch can be effectively avoided IBM (2007), but not completely. As it is inside a loop, the penalty gets accumulated and eventually limits the performance. Moreover, the accuracy of branch probability information can be another limiting factor. Branch probabilities affect the decision of which branches should be hinted can be affected, and we solely

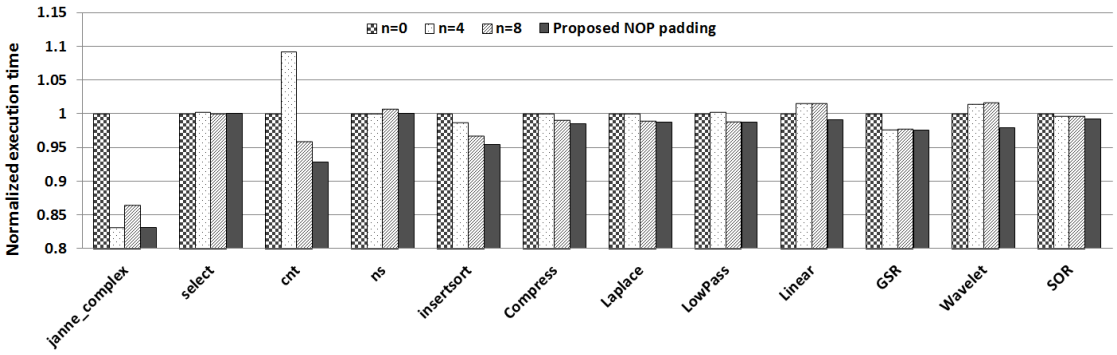

Figure 5.11: Execution Time Comparison Between Our NOP Padding Technique and GCC's "-mhint-max-nops" Option. In All Benchmarks, Our Technique Outperforms GCC. GCC Even Results in Performance Degradation for Several Benchmarks.

rely on static analysis to obtain branch probabilities, which may not be very accurate. Use of profile information may be helpful to improve the result.

5.7.3 Effectiveness of NOP Padding

GCC also has a mechanism where `nop`s are inserted between a branch and its hint in order to increase the separation and improve profit of hinting. However, it has no automatic way of determining how many NOPs to insert, and when compiled with "-mhint-max-nop=n" GCC gcc (2005) will insert at most n `nop`s to ensure the separation is at least eight instructions. On the contrary, our scheme automatically finds out the number of NOPs to be inserted, to maximize profit. Figure 5.11 compares the performance of our NOP Padding approach with that of GCC with $n = 0, 4$, and 8. Note that among the GCC schemes, sometimes $n = 0$ is better, while at other times $n = 8$ is better. This is because GCC does not have any profitability analysis to find out the number of NOPs to be inserted. Another advantage of our technique is that while GCC only inserts `nop`s, we insert `nop` and `lnop` pairs. By doing this, we benefit from the dual-issue nature of SPU Flachs *et al.* (2006). Even when two approaches insert the same number of NOP instructions, the performance penalty of our approach is as half as that of GCC. Therefore, the performance improvement of

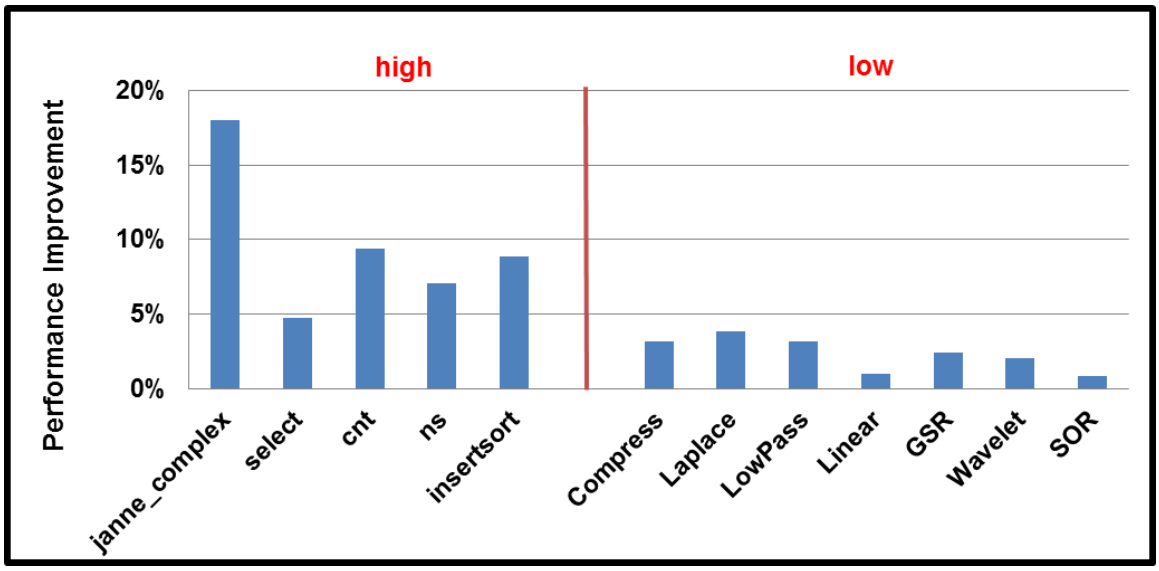

Figure 5.12: Performance Improvement Obtained with the Proposed Heuristic is 18% at Maximum.

our technique always outperforms the one of GCC.

This prudent insertion of NOP instructions is also important in terms of static code size increase. IBM Cell processor is a limited local memory architecture, and each SPU can only access its local memory with the size of 256 KB. Code, global data, and all dynamic data such as stack data and heap data need to reside in local memory. Consequently, it is important not to increase the code size too much. Note that this is static code size which affects the executable file size, and the dynamic code size increase overhead was already considered and included in the branch penalty reduction results. The average code size increase is merely 3.4%, while GCC incurs 11.7% code size increase with the "-mhint-max-nop=8" option.

5.7.4 Performance Improvement

The reduction in branch penalty cycles improves program performance, and the amount of performance improvement depends on the percentage of branch penalty in the total execution time.

Figure 5.12 shows the performance improvement for each benchmark, and as expected, benchmarks in 'high' group show more performance improvement than those in 'low' group, with the peak speedup of 18%. This is natural in the sense that higher proportion of branch penalty makes them more susceptible to performance improvement via branch penalty reduction. However, benchmark **select** has the second highest branch penalty percentage but shows the lowest speedup in the 'high' group. This is because it has multiple "if-then-else" branches in loops, whose penalty cannot be effectively avoided by software branch hinting as mentioned in the previous section. Though the benchmarks in 'low' group show relatively low speedup, it does not mean that our technique is not effective for those benchmarks. Our technique can reduce over 25% of the branch penalty for the benchmark **GSR**, however it is not fully reflected as reduction in execution time because its percentage of stall due to branch penalty is too low.

An important aspect of our technique is that our heuristic never results in a performance decrease. This is because every step of our technique involves profitability analysis. This guarantee, combined with the fact that the code size increase by our technique is minimal, we argue argue that it is always beneficial to apply our branch hinting heuristic.

5.8 Summary

Multi-core systems and power efficiency have been continuously driving modern processor design. Consequently, many complex architectural components are being removed from hardware and required to be implemented in software instead. IBM Cell SPUs removed branch predictor and introduced software branch hinting. Due to a huge branch penalty, branch hint instructions are crucial for performance optimization.

This dissertation present a heuristic algorithm to reduce branch penalty using software branch hinting. The algorithm is based on our proposed branch penalty model and three basic techniques, NOP padding, hint pipelining, and nested loop restructuring. The branch penalty model helps us to estimate the branch penalty, and those techniques not only enable more branches to be hinted, but also reduce more branch penalty.

Chapter 6

DATA MANAGEMENT FOR SMM

This chapter introduces optimization techniques of scratchpad memory management. The local scratchpad memory on an SMM architecture is shared among stack data, heap data, and code. Efficient management of each of the data types is crucial to application performance. In this chapter, stack data management is presented in Section 6.1, code management is presented in Section 6.2, and heap management is presented in Section 6.3. In each of the sections, the motivation and challenges are firstly discussed. Then optimization techniques of data management are proposed and demonstrated in details.

6.1 Stack Data Management

This section presents stack data management on SMM systems. An efficient stack data management scheme is critical for the performance of software, as about 64% of memory accesses in multimedia applications are to stack variables Guthaus *et al.* (2001).

6.1.1 Motivation

As shown in Figure 2.1, execution core can only access its local scratchpad memory, which is shared by text code, stack data, global data and heap data of the program executing on the execution core. All instructions and data must be present in the local memory when it's needed. Consequently, only a portion of the local memory can be used for managing stack data. Stack data management is challenging as its total size can not be determined at compilation time. It is simply because that stack data

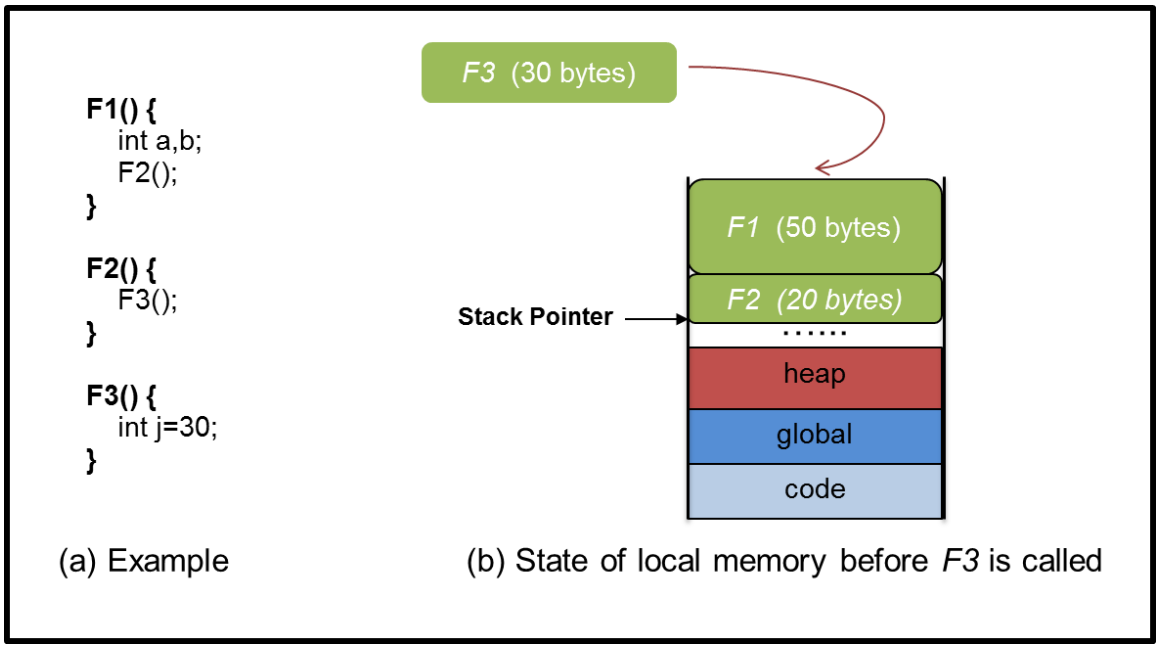

Figure 6.1: Suppose We Want to Execute the Program Shown in (a) On the Execution Core with Local Scratchpad Memory. (b) Shows that We Can Easily Manage the Stack Data of This Program in 100 Bytes, However, Trying to Manage it in Only 70 Bytes Local Memory Requires Data Management.

is dynamic in nature. Namely, function stack frames get allocated and de-allocated during the execution, when functions are called and returned respectively. Even more challenging, the total stack size requirement of the program may not even be known statically but input data dependent. For example, the size of stack data can be large when recursive functions exist in the program.

Though the size of stack data is non-deterministic, the amount of space in the local scratchpad memory is deterministic and ususally limited in a fixed size. In case of the situation that the total stack size is larger than the real memory, explicit management for stack data is required. We can look at the illustration in Figure 6.1. The example in Figure 6.1 (a) has three functions, whose stack frame sizes are shown in parentheses in Figure 6.1 (b). Figure 6.1 (b) shows the status of stack space just before function *F2* calls *F3*. If we have 100 bytes space for stack data, the application will work correctly and use up the entire space. However, if we only have

70 bytes to manage stack data, a space of 30 bytes must be found in the local memory for allocating the stack frame of *F3*. Without any management, stack data can grow downward and overwrite heap data or code, and eventually result in application crash in the best case, or simply an incorrect output in the worst.

A scheme is needed to make space for stack data and maintain the correctness of the application. Inspired by the approach which operates stack data management at the granularity of function frames Kannan *et al.* (2009); Bai *et al.* (2011), we evict some stack data in the stack space to main memory to make space for the coming stack frame. When the evicted frames is needed rightafter, we can bring them back to stack space in the local scratchpad memory. The eviction and fetch of function frames are impemented in API functions *_sstore* and *_sload* (briefly mentions in Table 6.10), that need to be integrated in the managed program.

6.1.2 Challenges

The idea aforementioned is intuitive, but there are two interwined problems to be considered. One is the the granularity of management. As in Bai *et al.* (2011), when there is no space for the incoming function in the local memory, the oldest stack frames from the top are evicted to make space which is barely enough for the incoming function. While it leads to a judicious usage of local memory space for stack management, this could results in stack memory fragmentation after some time. As a result, in order to track the status of stack space, the fine granularity management approach requires 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 APIs are called, which thus could harm the performance of applications. Besides, as the scheme is for manycore architectures, it has to consider bandwidth for data communication among differenct cores. Not

only in SMM architectures, but also in all manycore 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. Consequently, it is better to make small number of large requests than large number of small memory requests. The other one is the locations in the program the two APIs to be inserted. If stack data is managed with stack frame granularity, *_sstore()* and *_sload()* should be inserted right before and after each function call. These functions will not cause any data movement most of the time, but update management information. Specifically, 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.

In conclusion, a coarser management granularity is highly expected, which therefore results in better performance. In addition, an algorithm to analyze the application for judiciously placing *_sstore()* and *_sload()* functions in the managed program is needed.

6.1.3 Smart Stack Data Management

This section presents an approach called Smart Stack Data Management (SSDM) Lu *et al.* (2013). In this new scheme, stack data is managed at the granularity of the whole stack space and the insertion of management functions is optimized. Figure 6.2 shows the flow of SSDM infrastructure. Firstly, the optimized compiler takes in the application and generates its weighted call graph (WCG). Then SSDM greedy algorithm takes the WCG and the given size of local stack space S as inputs and determines the locations to insert *_sstore* and *_sload* in the managed program. Finally, the compiler deploys this location information to embed the runtime library

to original complied program.

Instead of evicting each individual function out of local scratchpad memory one by one, the management library evicts all the contents in the scratchpad memory to the main memory at once. 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 with whole stack granularity has two advantages: 1) the management complexity is largely reduced. Specifically, *_sstore* and *_sload* become simpler, since now the scratchpad is managed as a linear queue, rather than a circular queue. 2) the coarser granularity of stack data management can reduce the number of DMA calls.

Even with efficient management library for stack data, high overhead may still exist in this scheme, if the management functions are not judiciously placed. For example, thrashing of stack space can make this stack management not fascinating. This may happen when stack space is full just before entering a loop with high execution count in which another function is called. In this case, every time when the function is called, the stack state will be written to main memory, and then be

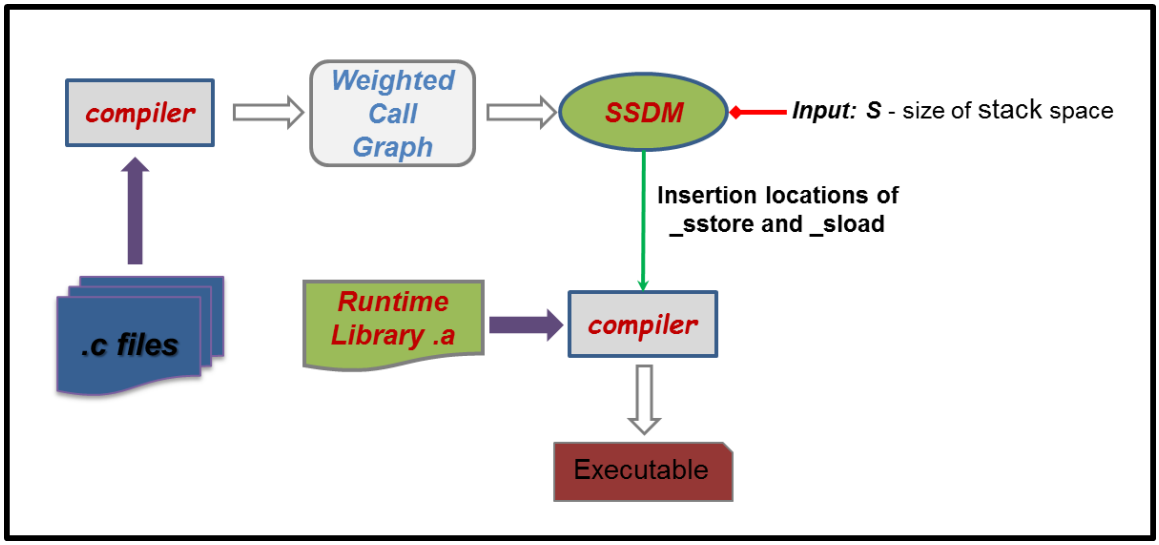

Figure 6.2: An Overview of SSDM Infrastructure

reloaded back to local memory on a return. This could be avoided by carefully placing the functions *_sstore* and *_sload* in the program. This problem of optimal placement of these stack data management functions is formulated in Section 6.1.3, where it is described as that of finding an optimal cutting of a weighted call graph (WCG). As this problem is tractable, a heuristic called SSDM is then present to solve this problem efficiently.

Problem Formulation

Stack data management function placement problem is formulated by using an input called *weighted call graph* (WCG), which integrates flow information, control information, function stack frame sizes, and the number of times a function gets called in the program. The formal definition of WCG can be found in Definition 1.

Definition 1 *(Weighted Call Graph). A weighted call graph* (V, E, W, T) *contains a function node set* V*, a directed edge set* E*, a weight set* W *and a value set* T*. 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 6.3 illustrates weighted call graph with a benchmark called SHA. Without loss of generality, 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. Several related definitions are described as follows:

Definition 2 *(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*

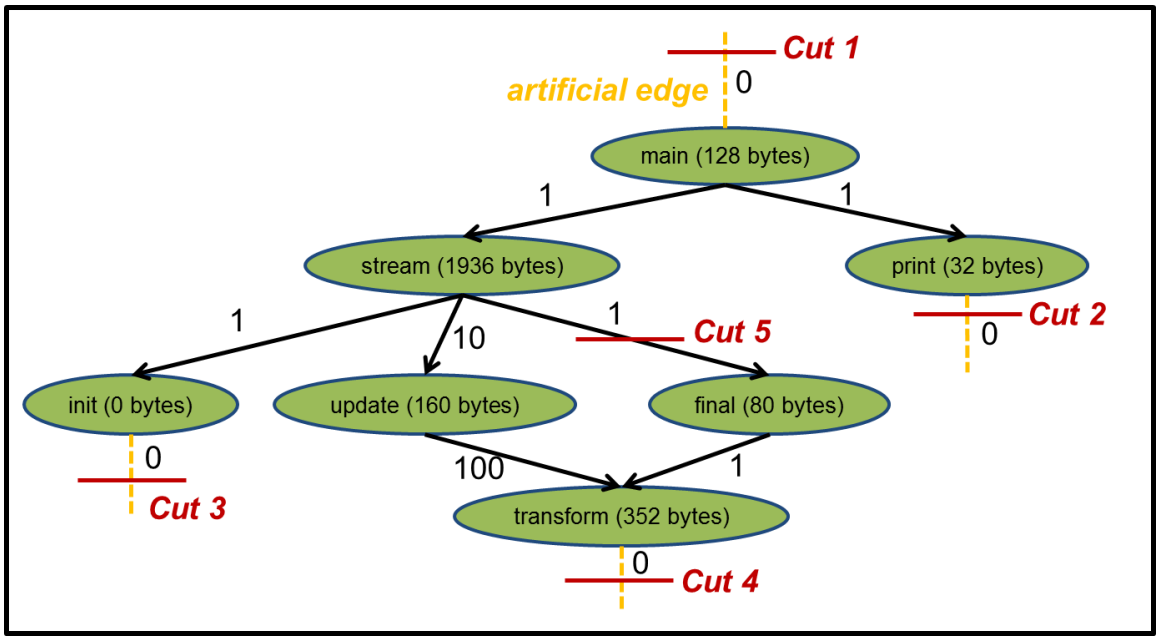

Figure 6.3: WCG with Cuts of Benchmark SHA: The Edge with Dashed Yellow Color Represents an Artificial Edge for Root Node and Leaf Node.

"main" function in a C program.

Definition 3 *(Leaf Node). A* leaf node *is the node that has no out-going edges. Those are functions that do not call any other functions. For example, transform function node is a leaf node.*

Definition 4 *(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 6.3.*

Definition 5 *(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 6.3, a set of cuts have been added on* artificial edges *in advance.*

Definition 6 *(Segment). A* segment *is a list of nodes which represents the collection of nodes on a root-leaf path between two cuts. In Figure 6.3, 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 cannot exceed the size constraint of stack space in the local scratchpad memory, 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:

$$\sum_{f_i \in s} w_{f_i} \leq \mathbb{W} \tag{6.1}$$

The cost of smart stack data management (SSDM) for each segment s has two components: 1) the running time spent on additional instructions caused by *_sstore* and *_sload* function calls, 2) the time spent on data movement between main memory and the local scratchpad memory. Let us assume two cuts on edges e_{start} and e_{end} form a segment $s = \{f_1, f_2, ...\}$ with weights $\{w_{f_1}, w_{f_2}, ...\}$, and the two edges have values t_{start} and t_{end} (the number of function gets called). Then the first cost component can be represented as

$$cost_1 = t_{end} \times \tau_0 \tag{6.2}$$

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

$$cost_2 = 2 \times t_{end} \times (\tau_{base} + \tau_{slope} \times \sum_{f_i \in s} w_{f_i}) \tag{6.3}$$

where τ_{base} is the base latency for any DMA transfer, τ_{slope} is the additional latency increasing rate with data size, and 2 means the stack frames being moved *in* and *out* of the local memory.

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

$$cost_s = cost_1 + cost_2 \tag{6.4}$$

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

$$cost_{WCG} = \sum_{s_i \in S} cost_{s_i} \tag{6.5}$$

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 7 *(**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.*

SSDM Heuristic

In this section, SSDM heuristic is present to solve 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 considering a cut removing, SSDM checks if removing this cut will violate the memory constraint of stack space. To do this, it searches 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 this cut is removed, the functions between any pair of upstream cut and downstream cut forms a new segment. If any of the new segment violates the memory constraint of stack space, the cut should not be removed, otherwise, it moves to next step to calculate how much benefit can be gained by removing this cut.

It first calculates the total management cost of all the segments associate with the cut with Equation 6.2-6.5. Then it assumes this cut is removed, and constructs new segments by combining upward segment and downward segment in the same *root-leaf path*, and calculates their total management cost in the same way. By subtracting the newer one from the older one, the *benefit* of removing this cut can be calculated. It computes 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 3 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 8-11, the segments that associate with each cut x_{ij} on edge e_{ij} ($e_{ij} \in E$) are listed. To do this, SSDM has to find out all root-leaf path P_i, where $e_{ij} \in P_i$. Then it searches upward through each P_i, until meets another cut x_{up}. Similarly, it searches downward through each root-leaf path P_i, until meets 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 6.3, the segments that are associated with cut 5 is the segment <*main*, *stream*> and the segment <*final*,

Algorithm 3 SSDM(WCG(V,E))

Place cuts on recursive edges, if there are recursive functions.
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 $x_{ij} = 1$.
while true **do**
    Define vector $\mathcal{B}$ to store *removing benefit* of each cut.
    for $x_{ij} == 1$ **do**
        violate $\leftarrow$ *false* // mark removing this cut will not violates the weight constraint.
        Define total cost $cost_{before} \leftarrow 0$.
        for segment s_old_i that are associated with x_{ij} **do**
            Calculate cost $cost\_old_i$ with Equation 6.2-6.5.
            $cost_{before} += cost\_old_i$
        end for
        Assume the cut of x_{ij} is removed, and get a new set of associated segments.
        Define total cost $cost_{after} \leftarrow 0$.
        for new associated segment s_new_i **do**
            Check weight constraint with Equation 6.1.
            if weight constraint is violated **then**
                violate $\leftarrow$ *true*; break
            end if
            Calculate cost $cost\_new_i$ with Equation 6.2-6.5.
            $cost_{after} += cost\_new_i$
        end for
        if violate **then**
            continue
        end if
        Calculate the benefit of removing the cut as $B_{ij} \leftarrow cost_{before} - cost_{after}$.
        if $B_{ij} > 0$ **then**
            Store B_{ij} into vector $\mathcal{B}$.
        end if
    end for
    if $\mathcal{B}$ contains no element **then**
        break
    end if
    Find out the largest benefit B_{max} from $\mathcal{B}$, and set the corresponding cut $x_{max} = 0$.
end while

transform>. Then it computes the cost of each segment with Equation 6.2-6.5, and the total cost by summing up the cost of all the associated segments. In Line 12-21, it assumes the cut is removed, and therefore generates 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 6.3, after removing the cut 5, the two associated segments are merged into one segment, which is <*main*, *stream*, *final*, *transform*>. Similarly, it calculates the cost of each new segment with Equation 6.2-6.5, and the total cost of all associated segments after removing the cut. Line 15-18 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 22-24). Line 33 removes the cut with largest positive benefit among all the cuts whose removal will not violate the weight constraint. Line 30-32 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 6.1).

Figure 6.4 illustrates SSDM algorithm. In this example, the stack frames of the example WCG (A) is managed in a 192 bytes stack space. When calculating the stack management cost with Equation 6.2 and Equation 6.3, τ_0 is set to 50 ns, τ_{base} is set to 91 ns, and τ_{slope} is set to 0.075. As stated before, artificial edges are added for this WCG and an artificial cut is attached for each artificial edge as well. At the initialization stage of SSDM heuristic (line 2 in Algorithm 3), cuts are added on all edges (cut 1-cut 4). Next we check the *removing benefit* of all existing cuts, except artificial cuts (line 5-27). 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 $<F_0>$

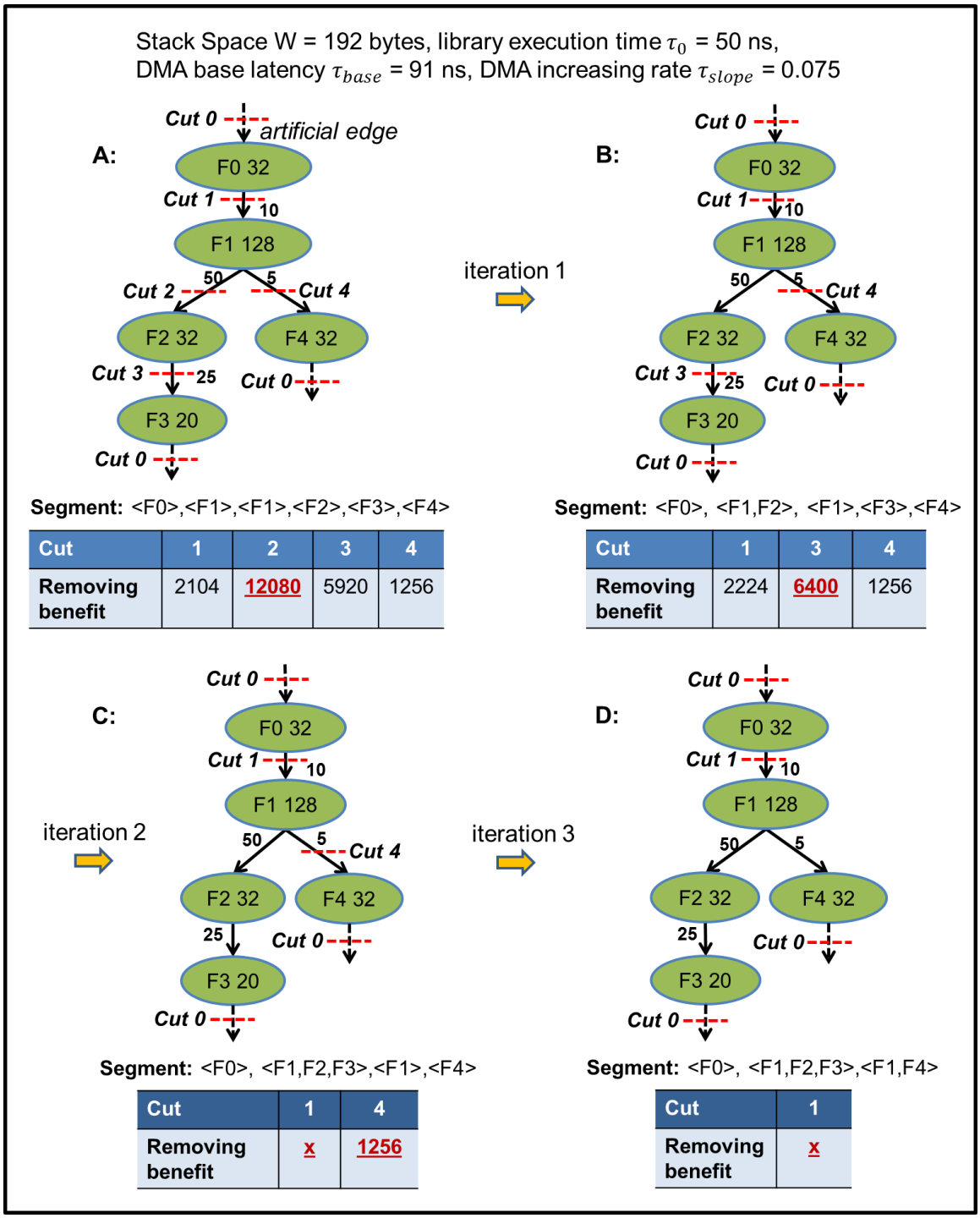

Figure 6.4: Illustration of SSDM Heuristic: the Values on Edges Are the Numbers of Function Calls.

is $2368 = 10 \times 50 + 10 \times 2 \times (91 + 0.075 \times 32)$ (Equation 6.2-6.4), 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 8-11). If cut 1 is assumed 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). $cost_{after}$ (line 12-21) can also be computed as 14080. Thereafter, the removing benefit of cut 1 equals to $2104 = 16184 - 14080$. Similarly, all removing benefit of all cuts are computed, and form the benefit table below WCG (A). As highlighted with *underline*, the largest benefit comes from removing cut 2. Then SSDM removes it and gets WCG (B). We can remove cuts one by one through WCG (B) to WCG (D), where cut 1 can no longer be removed. It's because the removal of cut 1 violates the weight constraint (line 15-18), 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, SSDM stops, and therefore WCG (D) is 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.

The top level SSDM algorithm is described in the previous section. In this section, the sub-problem of assigning the value to weighted call graph (WCG) will be discussed. 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, the application has to be profiled each time a new input is given. As the goal in this dissertation is to design fully automatic compilation techniques, in this section, static WCG construction method is present.

The proposed construction methodology works as follows. Firstly, the basic blocks of the managed application are scanned for the presence of loops (back edges in

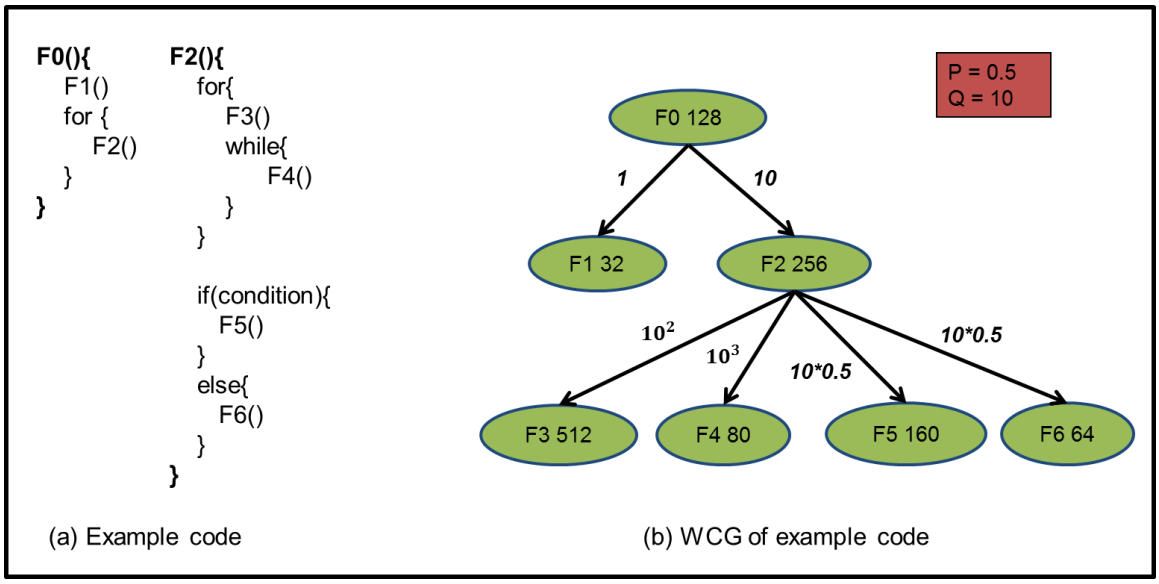

Figure 6.5: An Example Shows Static Edge Weight Assignment.

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, the number of loops (nl) nested for that function is saved. 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 without loops. If the edge is 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 Smith (1981). In addition, Q is set to 10. Figure 6.5 shows the resulted WCG of an example code with the 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.

In summary, this section present a technique for stack data management on Soft-

ware Managed Manycore (SMM) architectures, with function libraries *_sstore* and *_sload*. Smart Stack Data Management (SSDM) manages stack frames at the whole stack space granularity. In addition to having reduced the complexity of runtime library, the problem of efficiently placing library functions at the function call sites is formulated. Finally, a heuristic algorithm to generate the efficient function placement is proposed.

6.1.4 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 Scratchpad Memories (SPMs) in embedded systems. Banakar et al. Banakar *et al.* (2002) demonstrated that this compiler controlled scratchpad memory may result in performance improvement of 18% with a 34% reduction in die area. Consequently, SPMs are extensively used in embedded processors, for example, the ARM architecture ARM (2001). In SPM-based embedded processors, code and data can be managed to use SPM, so that the application can be optimized in terms of both performance and power efficiency. Techniques have been developed to manage code Balakrishnan *et al.* (2002); Angiolini *et al.* (2004); Nguyen *et al.* (2005); Egger *et al.* (2006a,b); Janapsatya *et al.* (2006); Udayakumaran *et al.* (2006); Pabalkar *et al.* (2008), global variables Avissar *et al.* (2002); Kandemir and Choudhary (2002); Nguyen *et al.* (2005); Udayakumaran *et al.* (2006); Pabalkar *et al.* (2008); Li *et al.* (2009), stack data Dominguez *et al.* (2005); Udayakumaran *et al.* (2006) and heap data Poletti *et al.* (2004) on SPMs.

While all these works are related, they cannot be directly applied 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, which are shown

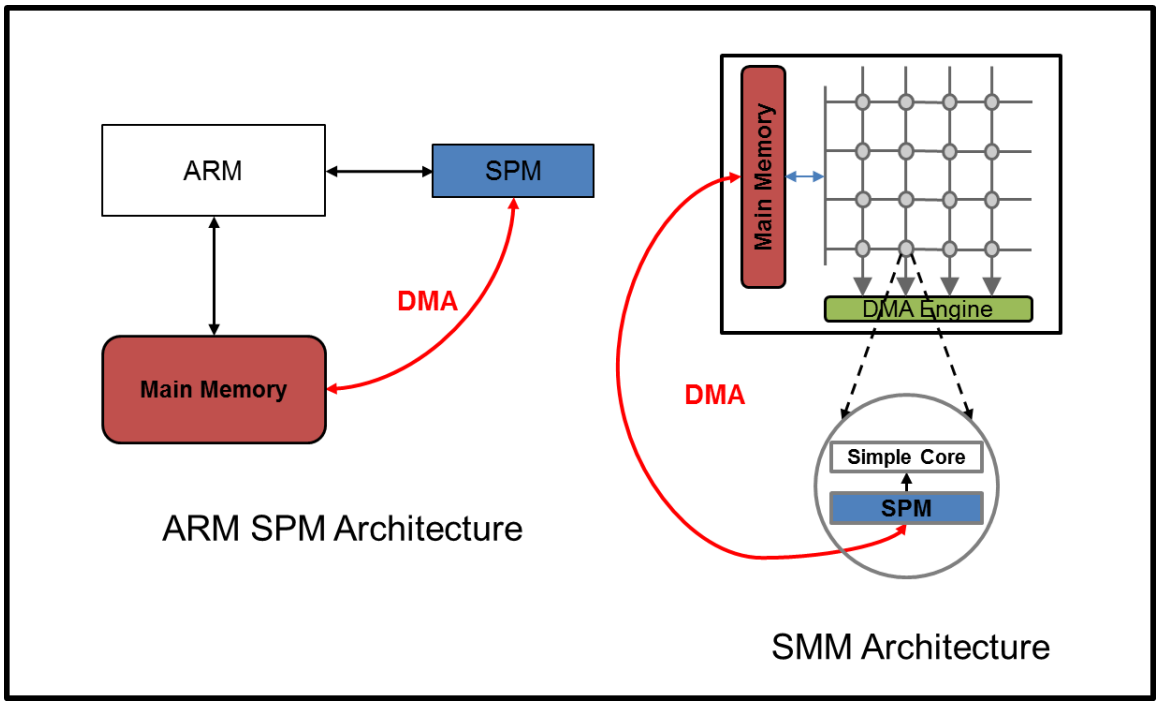

Figure 6.6: In the ARM Processor, SPM is in Addition to the Regular Memory Hierarchy, While in SMM System, the Local Memory is an Essential Part of the Memory Hierarchy on the Execution Core.

in Figure 6.6. Embedded processors, e.g., ARM processors, have SPMs in addition to the regular cache hierarchy, which 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 Banakar *et al.* (2002). In contrast, local memory is the only memory hierarchy of the core on SMM systems. As a result, using SPM is not optional but 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, otherwise the application will not work correctly. While the problem of using scratchpad memory in embedded systems is that of optimization, the problem of using scratchpad memory in SMM processors is to enable the execution of applications.

Only the Circular Stack Management (CSM) scheme in Kannan *et al.* (2009);

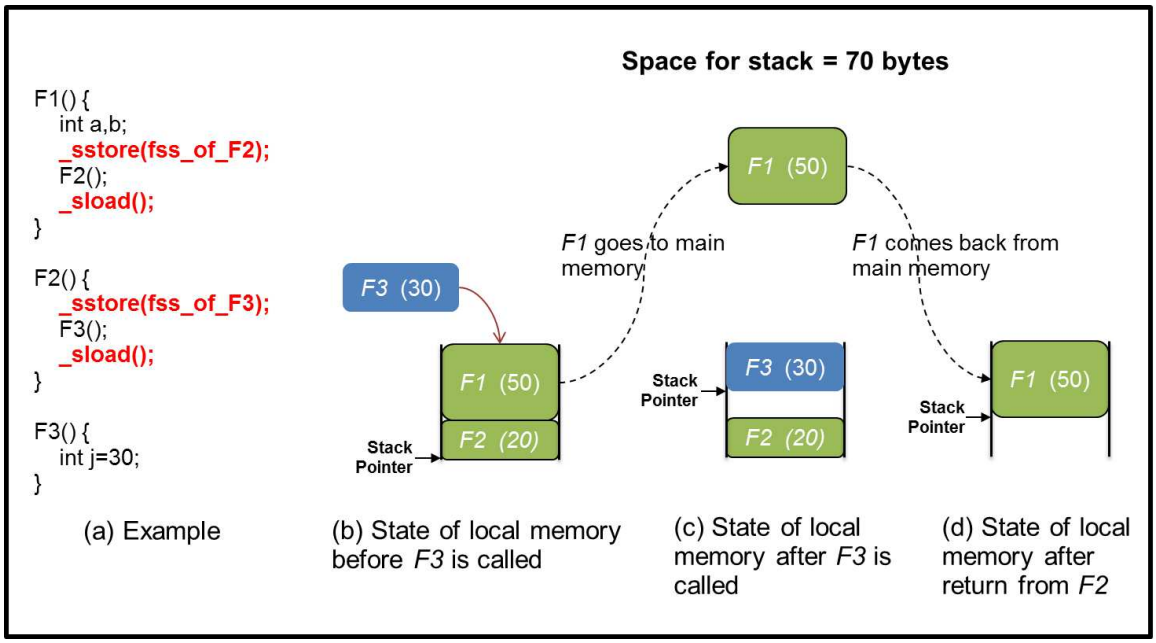

Figure 6.7: Circular Stack Management: The Function Frames Can be Managed in a Constant Amount of Space in Local Memory Using a Circular Management Scheme. If We Have Only 70 Bytes of Space on the Local Memory to Manage Stack Data, Frame F1 Must be Evicted to the Main Memory to Make Space for F3. Before the Execution Returns to F1, it Must be Brought Back to the Local Memory.

Bai *et al.* (2011) maps all stack data to the SPM, and will therefore work for SMM architectures. CSM exports function frames to the main memory if there is no more space on the local scratchpad memory and fetches them back when needed, at the granularity of function frames. Figure 6.7 illustrates the CSM mechanism. Consider the same application and function frame sizes as in Figure 6.1, and the problem is to manage its stack data in 70 bytes of space on local memory. Figure 6.7 (b) shows that the local memory is full after F1 calls F2, and thus there is no more space for stack frame of F3. To make space for F3, CSM evicts the stack frame of F1 to the main memory (shown in Figure 6.7 (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 correctly. However, after F2 returns, execution returns to F1, whose function frame is not in the local memory currently

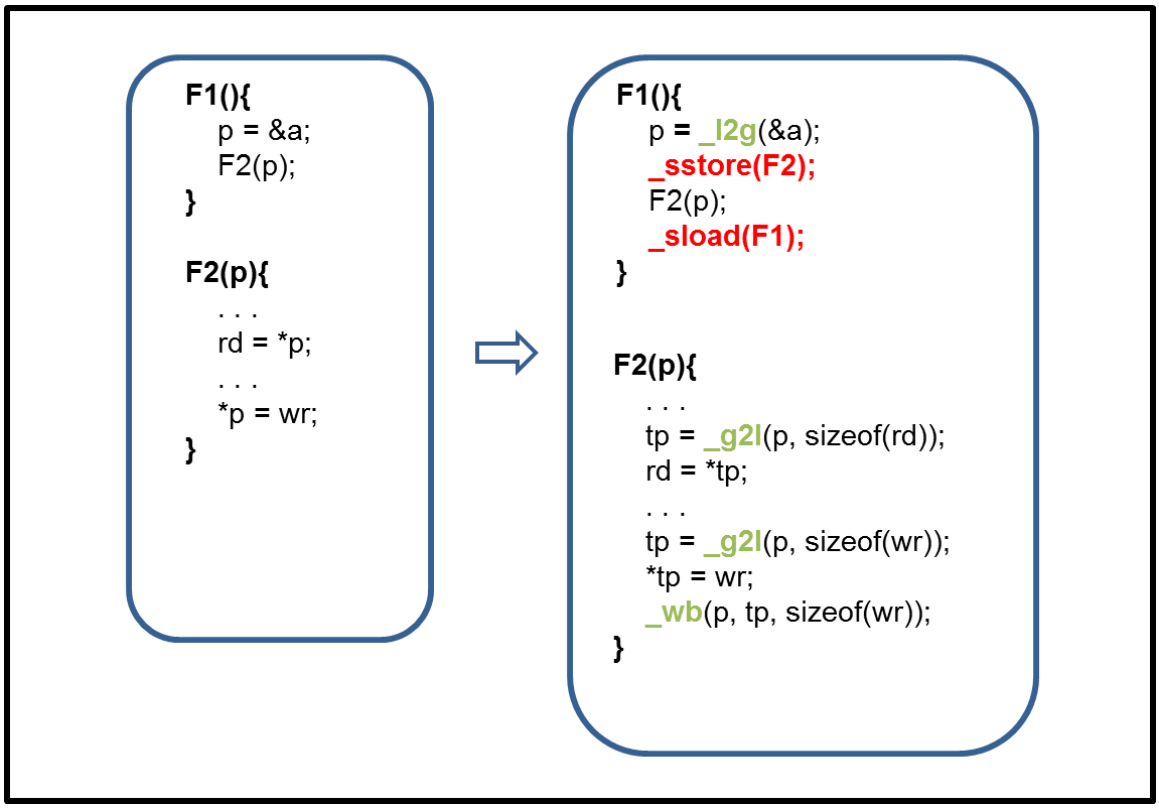

Figure 6.8: Pointer Management - Function F2 Accesses the Pointer p, Which Points to a Local Variable 'a' of Function F1. Since 'a' is a Local Variable on the Stack of F1, It Has a Local Address. When F2 is Called, if F1 is Evicted From the Local Memory, Then the Pointer p Will Point to a Wrong Value. This is Fixed by Assigning a Global Address to the Pointer When It is Created (Through **_l2g**), and Then When Needed, It is Accessed Through **_g2l**. Finally It is Written Back Using **_wb**.

and must be brought back. This is shown in Figure 6.7 (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 6.7 (a) shows these functions inserted in the original program in Figure 6.1 (a). The stack data management API function *_sstore(fss)* guarantees enough space to accommodate the stack frame with the size *fss*. If not, it evicts as many oldest functions as required to 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.

If a function accesses stack variables of another (ancestor) function through pointers (that may be passed to it as function parameters, or in other data structures), then there may be a problem. The problem, as shown in Figure 6.8 is that the pointer to a stack variable will be to a local address, since the stack is created in the scratchpad. However, when the pointer to a stack variable of an ancestor function is accessed, that function stack frame may have been evicted by the stack data management. Then the pointer will point to a wrong value. Bai *et al.* (2011) extended the stack management approach to handle pointers correctly. To resolve pointers, they converted the local addresses of the pointers to their global addresses at the time of their definition (through the use of *_l2g* function stub), and at the time of pointer access, the data pointed to is brought into the local memory (through the use of *_g2l* function stub), and after the program is done accessing, it is finally written back to the global memory (through the use of *_wb* function stub).

In Section 6.1, we identified, fixed several limitations of the CSM technique, and improved its applicability and generality. In addition, a more efficient stack data management technique was proposed. The comparison results will be present in Section 6.1.5.

6.1.5 Experimental Results

Experimental Setup

Stack data management techniques are demonstrated 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 Flachs *et al.* (2006). Our approach is implemented as a library with the GCC 4.1.1. We compile and run benchmarks from the MiBench suite Guthaus *et al.* (2001), whose details are listed in Table 6.1. These

Table 6.1: 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

benchmarks are not multi-threaded, but we made them multi-threaded by keeping all the input 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. Therefore, each benchmark has two threads, one runs on the PPE and the other runs on SPE. In our last experiment on scaling, we run multiple threads of the same functionality on the SPEs. The runtime on PPE is measured by *_mftb()* and the runtime on SPE is counted by *spu_decrementer()*, both of which are provided as the library with IBM Cell SDK 3.1.

Impact of Stack Space

This section present the performance of SSDM technique under tight size constraints, where the benchmark *Dijkstra* is chosen. It has many recursive function calls within loop structures, making it a good candidate for showing the impact of different stack region sizes. We increase the region size from 160 bytes to 416 bytes with the step size

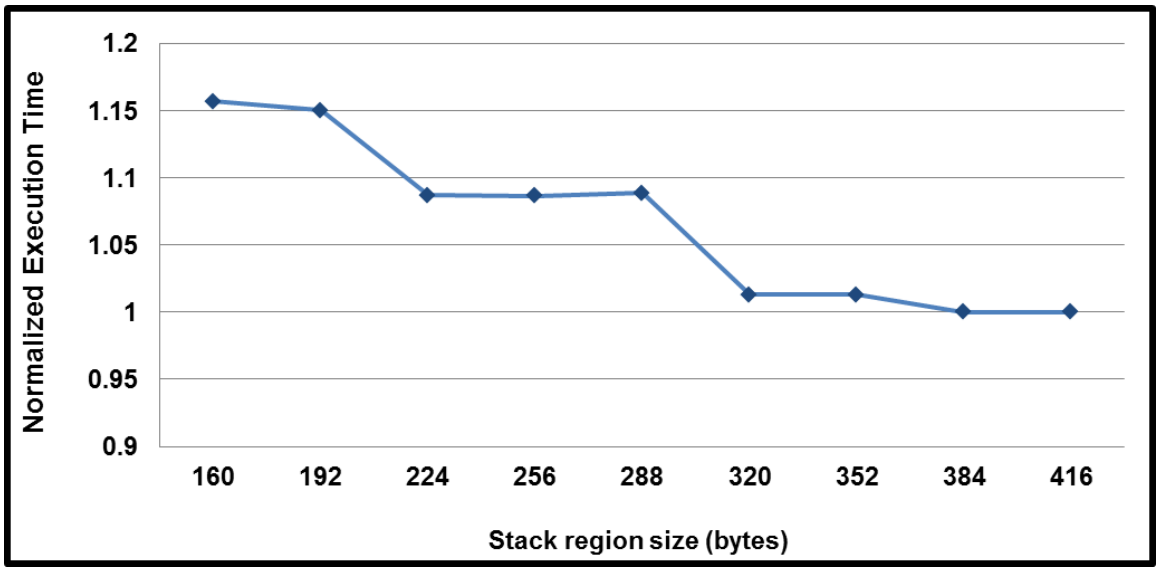

Figure 6.9: Performance Improves When Stack Region Size Increases.

of 32 bytes, and show all results in Figure 6.9. In the figure, the execution time with different stack region sizes are normalized to the smallest one. The execution time decreases when we increase stack region size. When the size reaches 384 bytes, the performance rarely 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. As a result, although the region size is large enough, no more benefit can be obtained as only the insertion for recursive function *print_path* is left.

Scalability of SSDM

Figure 6.10 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. 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 results in the competition of DMA requests. As shown in Figure 6.10, the execution time increases gradually as we scale the number of cores, but no more than 1%. Benchmark *SHA*

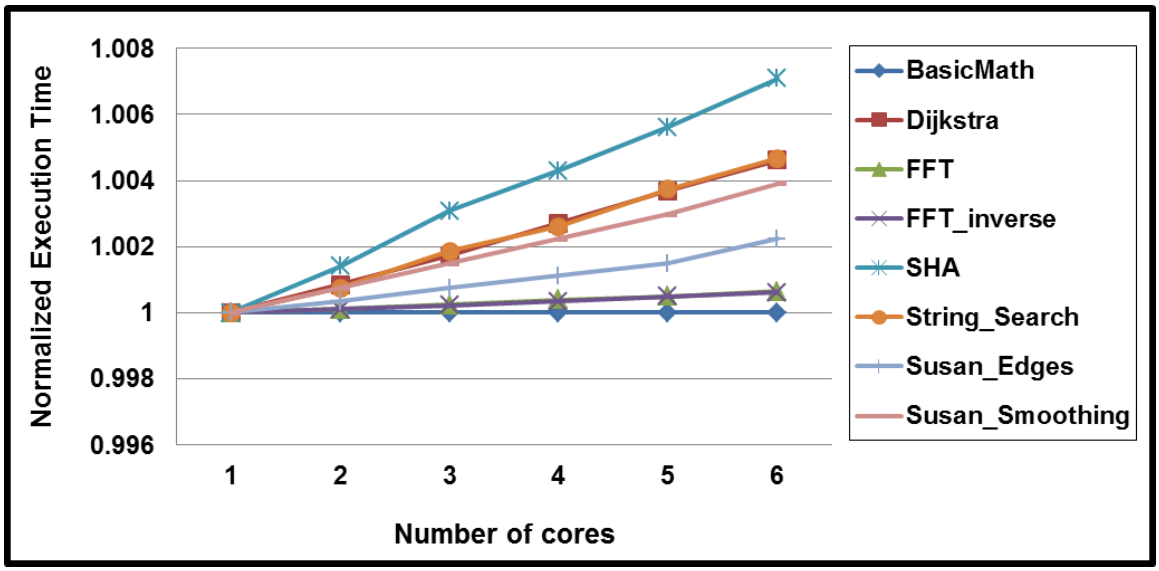

Figure 6.10: SSDM is Scalable, Since Performance Regression is Negligible When the Number of Cores Increases.

increases most steeply, because 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.

Thorough Comparison between CSM and SSDM

Overall Comparison

The experiment for each application in this section is conducted under the scratchpad size specified in Table 6.1. The efficiency of SSDM technique is evaluated by comparing it against CSM presented in Section 6.1.4 Bai *et al.* (2011). We first utilize PPE and 1 SPE available in the IBM Cell processor and compare our SSDM performance against the CSM result Bai *et al.* (2011). The y-axis in Figure 6.11 stands for the execution time of each benchmark normalized to its SSDM_P result, where the number of function calls used in Weighted Call Graph (WCG) is estimated from profiling information. In SSDM_S, we used a compile-time scheme to assign weights

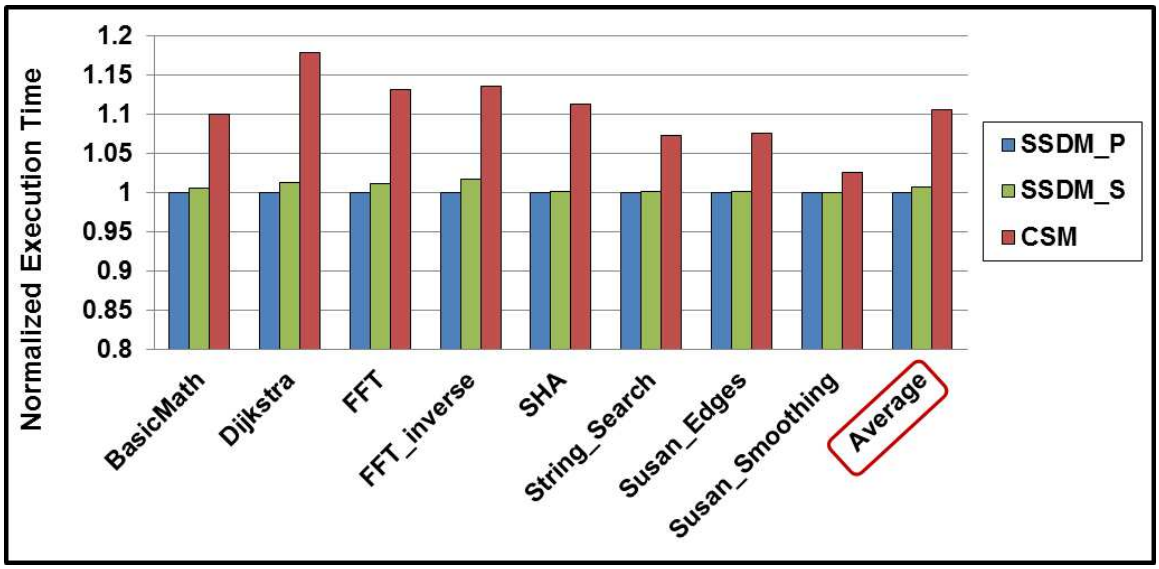

Figure 6.11: Performance Comparison Between SSDM and CSM.

on edges. As observed from Figure 6.11, both the non-profiling-based scheme and the profiling-based scheme achieve almost the same performance. Compared with the CSM technique, SSDM demonstrates up to 19% and an average 11% performance improvement.

The overhead of stack data management comprises of 1) time for data movement between global memory and local memory, 2) execution time of the additional instructions in the stack management libraries. Figure 6.12 compares the execution time overhead of CSM and that of SSDM. Results show that an average 11.3% of the execution time was spent on stack data management with CSM, while the overhead of approach SSDM is reduced to a mere 0.8% – a reduction of 13X. The performance gain comes from several aspects. In the following subsections, we break down the overhead and explain the effect of our techniques on its different components.

Management Library Size

SSDM library is less complicated than that of CSM, since CSM needs to handle memory fragmentation while SSDM doesn't have this circumstance. Consequently,

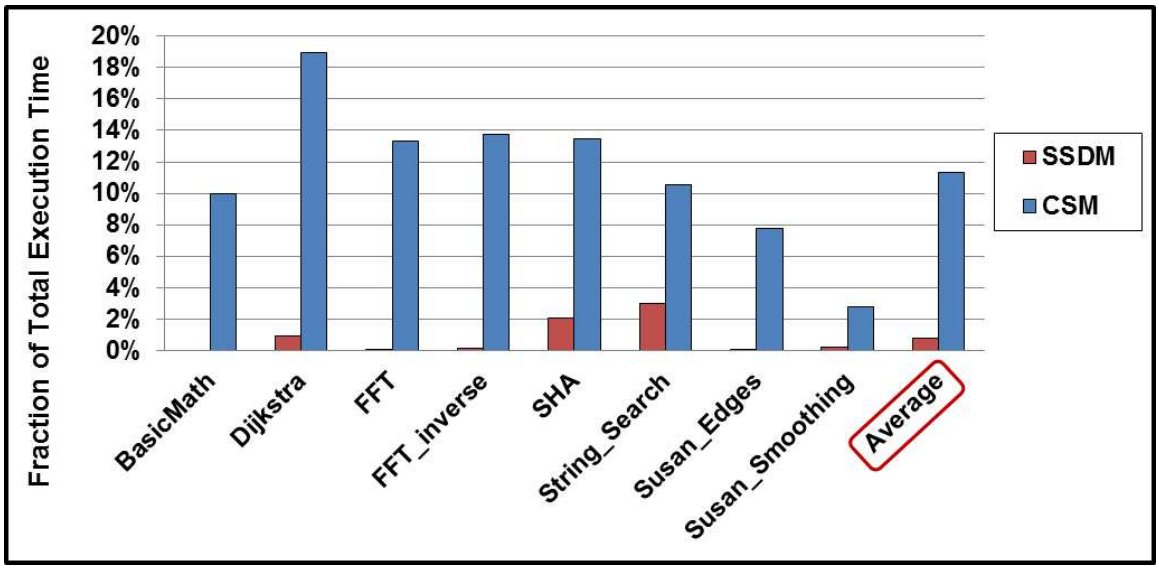

Figure 6.12: Overhead Comparison Between SSDM and CSM.

Table 6.2: Library Code Size of Stack Manager (In Bytes)

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

the library functions of SSDM contain fewer instructions than that of CSM. Table 6.2 compares the function footprint between SSDM and CSM, from which we can find SSDM library has much smaller code size than CSM does. Small library size is significantly important for improving the management performance in two ways. First, because the management 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 6.1.5.

Table 6.3: Comparison of Number of DMAs

Benchmark	CSM	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

Management Granularity

SSDM technique manages stack data at the stack space level granularity, which is different from the management scheme of CSM which manages data at the function level granularity. Therefore, the number of DMA calls in SSDM is reduced. Table 6.3 shows the number of DMAs in both stack data management approaches. 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*. CSM 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 6.4: Number of *_sstore* and *_sload* calls

Benchmark	_sstore		_sload	
	CSM	SSDM	CSM	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

Redundant Management Elimination

Thanks to our compile-time analysis, SSDM scheme can greatly reduce the number of library function calls. In Table 6.4, we compare the number of *_sstore* and *_sload* function calls in SSDM and CSM. 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 6.1.3. Therefore, it tries to avoid (if possible) placing *_sstore* and *_sload* around a function call that executes many times (e.g., within a loop) while CSM always inserts management functions at all function call sites.

The management overhead can be measured by extra instructions cause by stack management functions. Table 6.5 compares the average additional instructions incurred by each library call across all benchmarks. As demonstrated in Table 6.5, SSDM outperforms CSM. *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

Table 6.5: Dynamic Instructions Per Function

	_sstore		_sload		_l2g	_g2l		_wb	
	F	NF	F	NF		hit	miss	hit	miss
CSM	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.

when stack data is not in the local memory. In CSM 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, CSM may need additional instructions to transfer table entries.

6.1.6 Summary

In this section, a technique called Smart Stack Data Management (SSDM) that built upon Circular Stack Management (CSM) is proposed for stack data management on Software Managed Manycore (SMM) architectures. It manages stack frames at the whole stack space granularity. In addition to having reduced the complexity of runtime library, we formulate the problem of efficiently placing library functions at the function call sites. Eventually, a heuristic algorithm to generate the efficient function placement is proposed.

6.2 Effective Code Management

6.2.1 Motivation

On desktops or clusters with general purpose processing units, the system loads the complete compiled assembly instruction running on it into the main memory and then execute it. Even if a huge program could not be fully loaded, most of instructions can be put into the memory, and instruction cache could automatically fetch the required ones when needed. The process is transparent to software developers. However, Software Managed Manycore (SMM) architecture has a limited memory on each processing unit. 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 (i.e., instruction cache). To enable the execution of large applications on SMM architecture, it is necessary to use code overlay IBM (2008). In addition, code overlay could also be used for achieving performance improvement. As the local memory is shared by code and data of the mapped program, 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 could be improved because of larger memory resource.

6.2.2 Code Overlay Mechanism

Usually, the overlay organization is generated manually by developpers 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

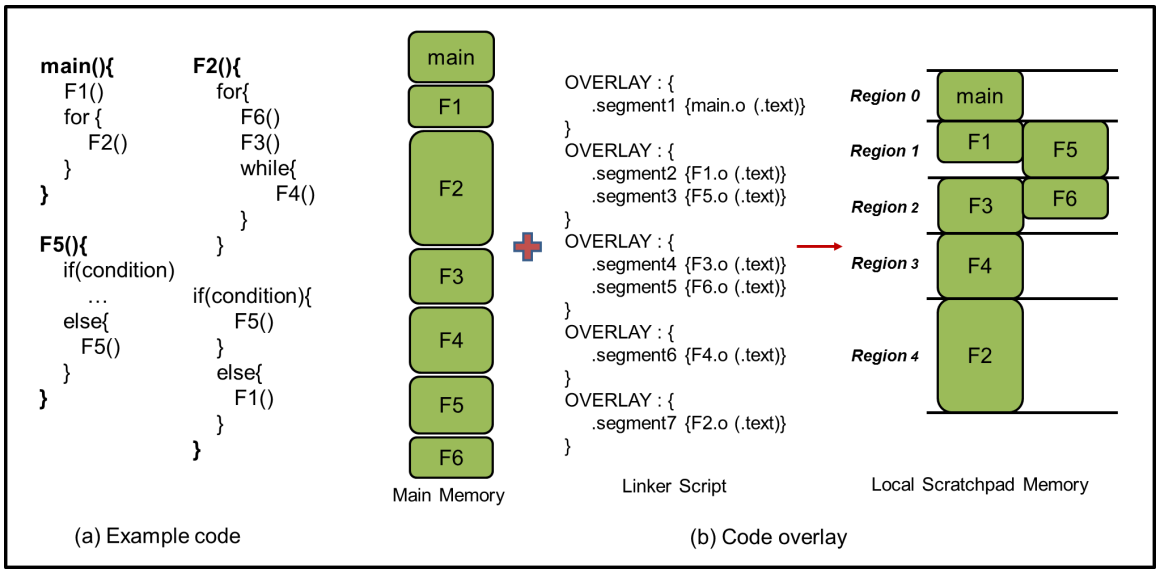

Figure 6.13: Code Overlay on Scratchpad Memory: When Task Assigned to the Execution Core Requires Larger Memory Than the Available Space, Code Needs to Be Mapped Between External Shared Main Memory and the Local Scratchpad Memory of the Core.

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 6.13 shows one example of code overlay for SMM architectures. Functions mapped to the same region will be located in the same physical address, and must replace each other during run time IBM (2008). 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.

Code overlay comprises of an overlay manager and a 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), 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. Otherwise, the instructions of f are loaded into the mapped memory region, to its specific memory address during run-time, by performing special DMA operations. 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 scheme in this dissertation works at the granularity of one function object. The new instructions to a region overwrite the existing instructions present in that region. Before jumping to the target address once the code segment has been loaded, the overlay manager also ensures successful completion of the DMA process to avoid any unwanted behavior in the program execution.

6.2.3 Objective of Code Overlay

For code overlay to work best, there are two intractable problems to be considered: 1) determining the number of regions, and 2) mapping all functions to regions. In terms of application performance, it is best to place each function into a separate region, so that it will not interfere with any other objects, but that may requires the largest code space in the local memory. On the contrary, mapping all functions into one region uses the minimum amount of code space, while incurs the most instruction transfers and therefore biggest runtime overhead.

Definition 8 *(**Optimal Code Overlay**). The task of optimizing code overlay is,*

to organize the application functions into regions that will leads to the least data transfers, given a predefined size of code space.

6.2.4 Cost Calculation of Code Overlay

As mentioned in Section 6.2.1 and Section 6.2.2, when two functions are mapped into a same region, they would swap each other during the execution time, which therefore lead to performance regression. Therefore, there is a need to estimate this swap cost in order to develop any code overlay mapping. 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 the cost. Proposing a correct and comprehensive cost calculation is of utmost importance, as it is the foundation upon which any mapping algorithm can be proposed. Next two sections address the cost estimation problem by deploying a graphical code representation and presenting a cost calculation algorithm.

Graphical Code Representation

Correctly calculating the management overhead and efficiently mapping code requires 1) the deep understanding the structure of the managed application, and 2) representing the flow information and control information in an effective form. This information can be built into an enhanced Control Flow Graph (CFG) known as Global Call Control Flow Graph (GCCFG) proposed by Pabalkar et al. Pabalkar *et al.* (2008).

Definition 9 *(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*

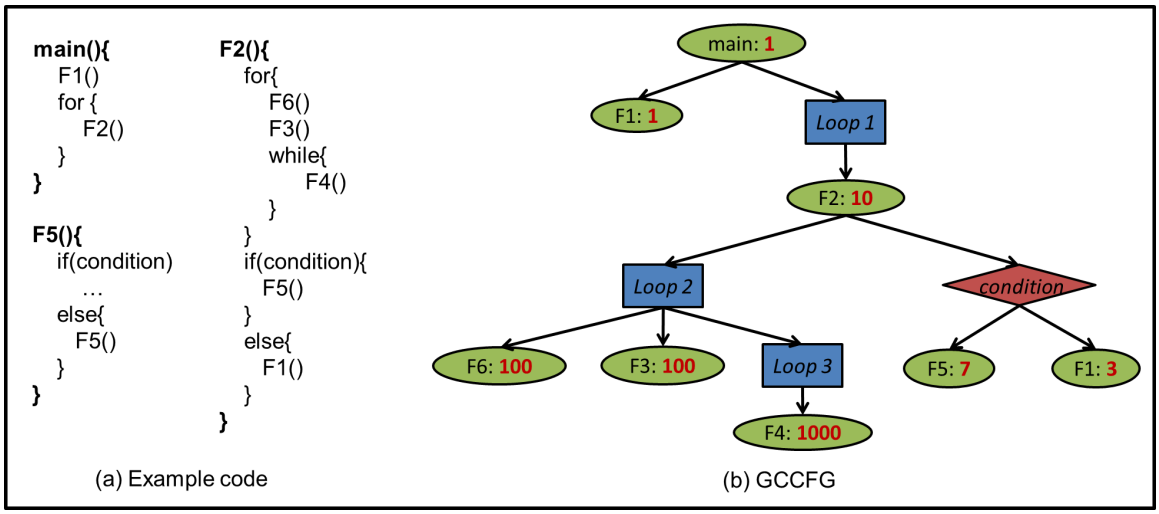

Figure 6.14: The GCCFG for the Example Code

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. Figure 6.14 illustrates the GCCFG of an example code, where direct recursive function calls F_5 is ignored. This is because the code necessary to run the called recursive function is already in memory, resulting in no instruction transfers.

This paragraph presents the complete algorithm to construct the GCCFG of an application. The input of the algorithm is all control flow graphs (CFG) of the program. Then all the CFGs are integrated into a GCCFG in two steps. First,

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. Second, 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 left child should be the first function that called and the right one should be the second one. 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.

Profiling and static estimation both can assign weight for *function nodes* in GC-CFG. The former 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 instance, the number of iterations of a *while* loop with an input dependent condition could be easily obtained. The static compile-time weight assignment scheme is not trivial but significantly important, since it removes the expensive and prohibitive task of profiling. Furthermore, the experimental results show the estimation of weight will not degrade too much performance. The methodology for estimating the number of function calls on each function node is described as follows. 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). Then the weights on the functions is assigned 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 which are not in any loop. 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*. A traditional scheme described by Smith Smith (1981) is adopted to predict the branch probability. The impact of Q is negligible as long as it is larger than 1 (details is shown in Section 6.2.7). As a result, Q is chosen to be 10 in this dissertation. The previous Figure 6.14 is the resulted GCCFG of the example code with our static weight assignment scheme.

Cost Calculation Heuristic

Making efficient interference cost calculation is of utmost importance, as it is highly frequently required by CMSM (Code Mapping for Software Managed multicores). 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, this algorithm has bad complexity. Therefore, this dissertation presents 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

Algorithm 4 Algorithm cost (GCCFG, v_1, v_2)

```
1: v_current = v_initial
2: while v_current ≠ v_final do
3:     if v_1 is found and v_2 is not found then
4:         if M(v_current)==v_1 or M(v_current)==v_2 then
5:             reset all weights
6:         else
7:             if v_current is LCA(v_1, v_2) then
8:                 assign weight_1
9:             end if
10:        end if
11:    end if
12:    if v_current == v_1 then
13:        assign weight_1
14:    end if
15:    if v_current.nextNode == loopNode then
16:        find next function node, then assign weight
17:    end if
18:    if v_1 found && v_2 found then
19:        assign weight_2
20:        totalWeight += min(weight_1, weight_2)
21:    end if
22:    v_current ← v_current.nextNode()
23: end while
24: return totalWeight
```

are interfered by each other Pabalkar *et al.* (2008); Jung *et al.* (2010). However, the interference between such functions depends upon mappings of other functions in-between during the execution. As a result, 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 compute the *interference cost* between two functions. As outlined in Algorithm 4, the interference cost between functions is calculated when traversing 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, the first edge weight (line 13) between v_1

and the next node is assigned. 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 least common ancestor (LCA) of v_1 and v_2 after v_1 is found, the first edge weight is re-assigned (lines 7-8). When v_2 is found after v_1 is found while it is traversing the GCCFG, it assigns the second edge weight and adds 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, it calculates interference count again with switched order of two functions and takes the maximum value of two computing. 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 algorithm visits each node in the GCCFG only once, thus the runtime complexity of interference cost calculation is $O(V_f)$.

6.2.5 CMSM Heuristic

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 Pabalkar *et al.* (2008); Verma and Marwedel (2006). Therefore, a greedy algorithm for code overlay is expected. Algorithm 5 presents the proposed CMSM heuristic. It starts with a mapping, in which each function is mapped to a separate region respectively (line 1). Next, all combinations of two regions are tried to be merged until the total space meets memory constraints (while loop, lines 3-7). In order to achieve this, two "balanced" regions with minimal merge cost is firstly found

through function FindMinBalancedMerge() in line 4. Then two regions are merged and the region information is updated in the set SPMregions (line 5-6). Function FindMinBalancedMerge() is described in Algorithm 5, where a region pair (R_1, R_2) is chosen (Algorithm 5, line 12-21), and its merge cost is calculated in line 15. The cost calculation is done with Algorithm 4. Besides, there is a balance factor $\frac{max-min}{(max+min)^2}$. It is designed to place the functions having close object sizes into the same region. This factor 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.

The *while loop* in line 3 in Algorithm 5 merges two regions at a time. 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 algorithm is $O(|V_f|^4)$.

6.2.6 Related Work

To the best of our knowledge, work Pabalkar *et al.* (2008); Baker *et al.* (2010); Jung *et al.* (2010); Jang *et al.* (2012) are similar to our effort for code management on SMM systems and Jung *et al.* (2010) is the most related one. Two mapping algorithms were proposed in Jung *et al.* (2010). One is function mapping by updating and

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 ← 0, R_src ← 0;
3: while SPMSize() > 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 ← DBL_MAX, tmpCost ← 0;
12: for all combination of regions R_1, R_2 ∈ SPMregions do
13:     size1 ← RegionSize(R_1), size2 ← RegionSize(R_2);
14:     max ← max(size1, size2), min ← min(size1, size2);
15:     tmpCost ← cost(GCCFG, R_1, R_2) * (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
```

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.

In addition, the work Pabalkar *et al.* (2008); Baker *et al.* (2010); Jang *et al.* (2012) provide several different heuristics for code overlay mapping on SMM architectures.

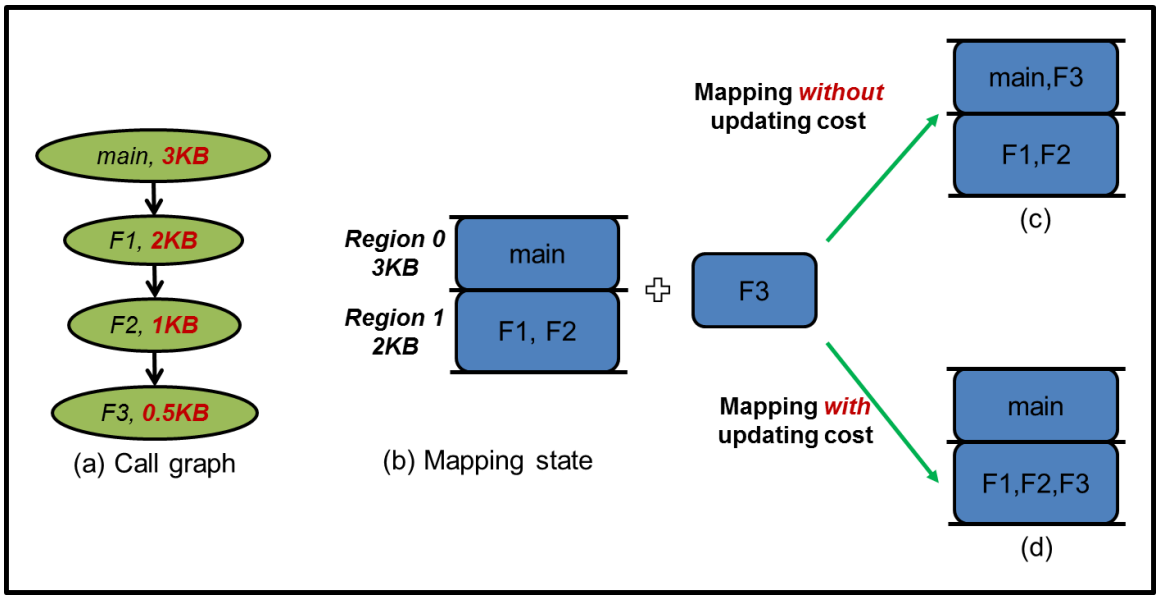

Figure 6.15: Cost Between Functions Depends on Where Other Functions are Mapped, and Updating the Costs as We Map the Functions Can Lead to a Better Mapping.

However, they are all not efficient enough, which is mainly because of inaccurate or incorrect cost calculation. They statically calculate the code mapping cost and generate a mapping. They never dynamically update the cost during the course of mapping algorithm, which is insufficient and results in inferior mapping. Figure 6.15 (a) shows a simple 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[1] 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

[1]The interference 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.

follows. When F2 is called, 1 KB of 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 Pabalkar *et al.* (2008); Baker *et al.* (2010); Jang *et al.* (2012) 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 6.15 (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. When mapping function F3 (size of F3 is 0.5 KB), we can map it 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 6.15 (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 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 6.15 (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 our proposal, the limitation aforementioned is addressed by deploying a graphical code representation (Section 6.2.4), and proposing a cost calculation algorithm (6.2.4).

6.2.7 Experimental Results

Experimental Setup

IBM Cell processor Flachs *et al.* (2006) is used as our hardware platform for conducting experiments. 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 in Table 6.6 are from Mibench suite Guthaus *et al.* (2001). All those information is obtained 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 deploy 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 6.2.7.

Overall Performance Comparison

While the conclusion scale for all benchmarks, Figure 6.16 shows the execution time of the binary compiled using each heuristic for only two representative applications.

Table 6.6: 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

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 the code size constraint is relaxed, 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 6.6, all three algorithms again achieve very similar code mapping. This is because there are quite few functions mapped to one region when the code

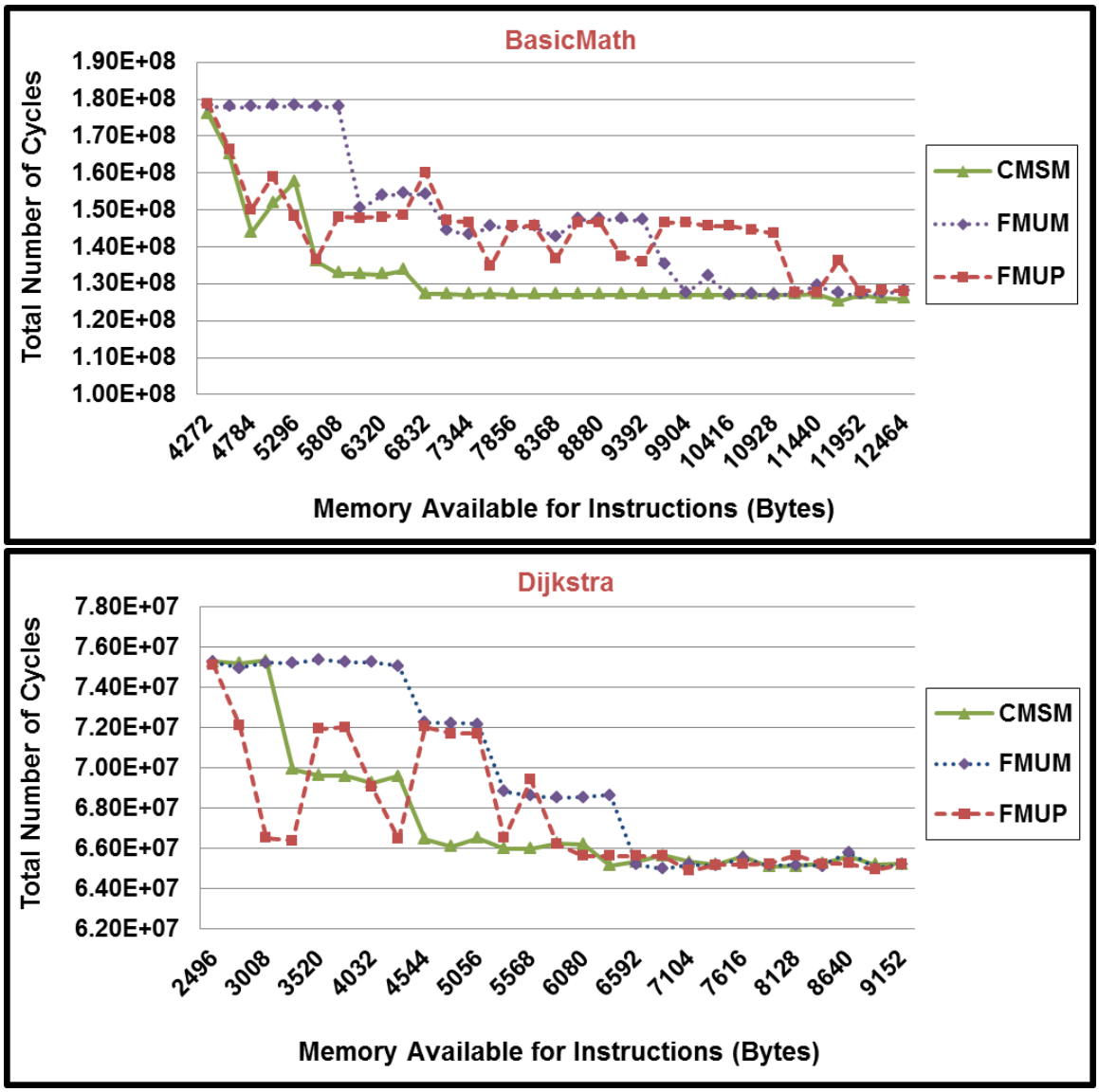

Figure 6.16: Performance Comparison Against FMUM and FMUP

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 instance, when memory available for instructions of benchmark "dijkstra" is 3520 bytes in Figure 6.16, CMSM is worse than FMUP. This is because FMUP has to do very few steps, while CMSM needs to do many iterations of

merges. The more steps a heuristic has to take, the errors in each step accumulate, and eventually lead to a worse mapping. Although our heuristic does not consistently gives good results, it gives better results most of the times. We tested three approaches for all code size constraints from the minimum to the maximum. On average over all benchmarks, CMSM gives a better result than other two algorithms 89% of time. Another important observation from Figure 6.16 is that, applications are tend to have less execution time when their code space become larger. A large code space usually results in 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.

Accuracy of Weight Assignment

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

Scalability of CMSM

Figure 6.17 shows the examination of 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 show them on y-axis. In this experiment, we executed the identical application on different number of cores. According to the

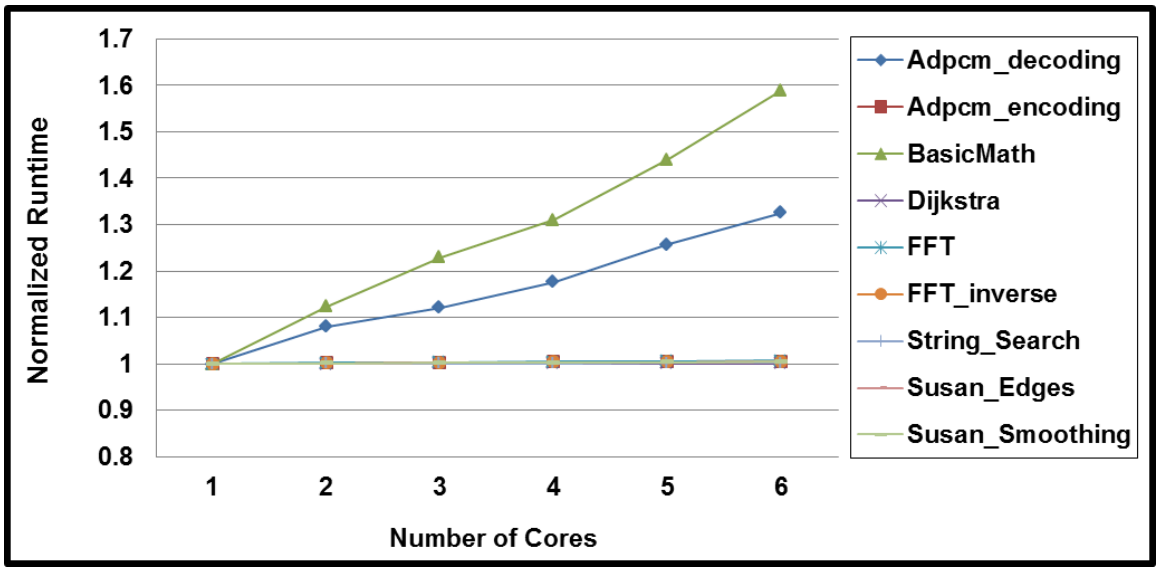

Figure 6.17: Scalability of CMSM on Multicore Processors

figure, 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.

6.2.8 Summary

Software Managed Multicore (SMM) processors 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 section presents a framework to manage code between main memory and the local memory, at the granularity of function object. We addressed the cost estimation problem in previous work by devising a correct cost calculation model and an algorithm for the same. Since code mapping problem has been proved to be

NP-complete, a heuristic called CMSM is proposed for the same problem.

6.3 Heap Data Management

This section presents heap data management on SMM systems. An efficient heap data management scheme is critical for the performance of software, since heap accesses may account for a significant fraction of all the memory accesses that the application makes.

6.3.1 Motivation and State of the Art

A lot of researches have been proposed to manage data on SMM systems. Among them, heap management is extremely difficult because of the dynamic nature of heap data. However, since heap data access may account for a significant fraction of all the memory accesses in an application, it is important to manage heap data in an efficient way. Bai and Shrivastava (2010) was the first software scheme for heap data management on SMM. In Bai and Shrivastava (2010), a mapping between the global memory and the local memory was established and maintained with a heap management table. Although this scheme manages heap data in a correct way, high performance overhead was incurred, due to the large number of extra management instructions in the code. In addition, this method is semi-automatic, in the sense that it requires manual library function insertion by developers. In 2013, a fully automated heap management technique was published in Bai and Shrivastava (2013). This technique employs a modified GCC compiler and a runtime library to fully unburden programmers from manually inserting API functions. Meanwhile, a more optimized data structure was leveraged to reduce performance overhead. We consider Bai and Shrivastava (2013) as the state-of-the-art. However, Bai and Shrivastava (2013) still suffers from high performance overhead caused by large amount of management instructions, complicated

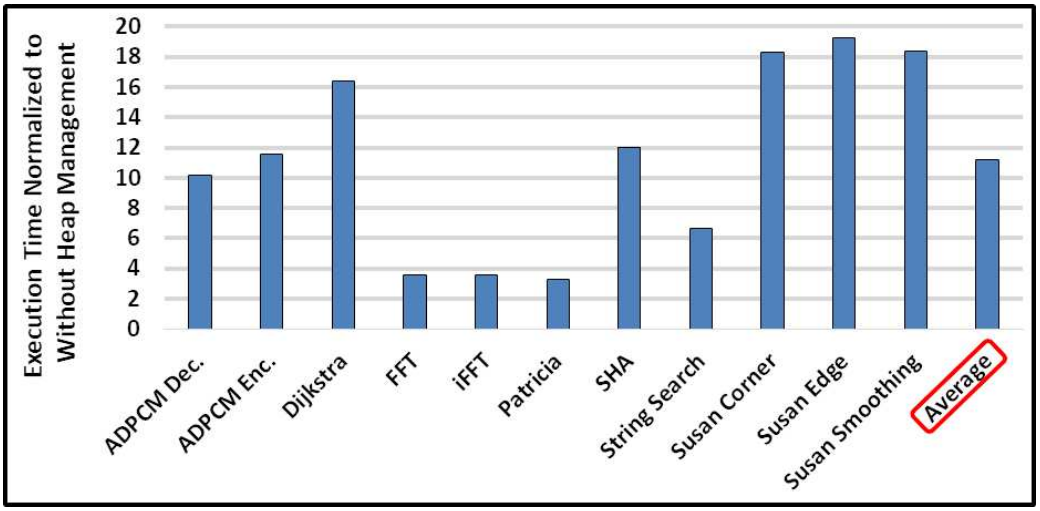

Figure 6.18: Performance Overhead with the State-of-the-art Heap Management.

management data structures, and substantial miss rate.

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

In the state of the art Bai and Shrivastava (2013), a $g2l$ function was implemented to translate a global address to a local address on SPM. It takes a main memory address as the input, and checks if the given address is in heap. If the address is not in the heap region, the input address is immediately returned. Otherwise, the set index of the input main memory address is calculated. A sequential search is done to compare the tag of the input address with the tags saved in the entries of the corresponding set in the management table. If a match happens and the status of the

matching entry is valid, a hit happens. Otherwise, if a miss happens, the enclosing data block of the input address will be copied from the main memory into the SPM. If no available entry can be found in the set, the data block pointed by the victim index will be replaced by the new data block, and the corresponding entry in the management table is updated with the new tag accordingly. The evicted data block must be written back to the main memory if it has been modified. The victim index is increased by 1 and modulo 4 (the number of entries in each set). Eventually, the SPM address is calculated based on the set index and its offset within the data block, and used in the memory access.

Though the state-of-the-art Bai and Shrivastava (2013) has correctly managed heap data, high performance overhead has been incurred. Figure 6.18 shows its management overhead on some typical embedded applications. It is important to note that this technique not only incurs high overhead when heap management is needed, but also inflicts high overhead on the benchmarks even without any heap accesses, i.e., `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search`. The high overhead is caused by two main reasons:

i) Unnecessary invocation of heap management function $g2l$**.** $g2l$ is called before each memory access (even without heap data access), which introduces not only management overhead, but also branch operations, and potentially more memory operations at every memory access.

ii) Over-complicated heap management instructions. This is because the state of the art implements $g2l$ in a set associative manner. The function has to sequentially search all the entries in the set at every heap access. It also complicates the calculation of the set index due to the involve of a translation from a main memory address to the corresponding local SPM address. The set index of the input main memory address is calculated with Equation (1), where mem_addr is the input main memory address,

block_size is the size of a data block, and *set_num* is the number of sets. The SPM address is then calculated with Equation (2), where *spm_base* is the start address of the data region, *set_assoc* is the set associativity (4 in this case), and *entry_index* is the index of the entry in the set specified by *set_index*. The complexity of the calculations has contributed to significant instruction overhead.

$$set\_index = ((mem\_addr >> log(block\_size)) \wedge (mem\_addr >> (log(block\_size) + 1)))\&(set\_num - 1) \quad (\mathbf{1})$$

$$spm\_addr = (set\_index * set\_assoc + entry\_index) * block\_size + spm\_base + mem\_addr\%block\_size \quad (\mathbf{2})$$

6.3.2 Efficient Heap Data Management

In order to reduce the overhead of heap management on SMM architectures, we proposed the following approaches:

i) Detecting heap access at compile time. This optimization identifies heap access statically and invokes heap management function *g2l* only when there is a heap data access. It also eliminates the unnecessary runtime checking within the management function once the memory access is determined to be a heap data access at compile time.

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

iii) De-duplicate management calls. The common part of management instruc-

tions $g2l$ are executed before all management calls. This optimization is particularly beneficial when management functions are called within loop nests, as the common operations are hoisted outside of the loops.

iv) Adjusting block size. All the aforementioned optimizations are generic, and thus are useful for all applications. However, in embedded systems, where profiling information can be obtained, heap data management can be further optimized. Depending on the type of cache misses that an application suffers from, the block size can be statically adjusted to avoid these misses. Given the size and set associativity of a software cache, adjusting block size will change the mapping between main memory locations and SPM memory locations. If the cache misses that an application encounters are mostly conflict misses, the block size can be reduced so that the number of sets could be increased to lower the chances of conflicts. On other hand, if an application obverses more cold misses, then the block size should be increased to refrain from such misses.

Static Heap Access Detection

In order to perform heap management only when there is a heap access, we developed a static heap access detection technique to identify heap accesses at compile-time. Figure 6.19 illustrates the effect of this optimization. To be noted that both the previous approach and our approach are implemented in IR level. Therefore, the codes in (b) and (c) are the source-code representation of the transformed IR. The original program defines a structure, which consists of two integer pointers `x` and `y`. It then instantiats a global variable `t` as an instance of the structure, and assigns `t->x` with an heap object created by a `malloc` function. The program then points `t->y` to the fourth integer element starting from the address in `t->x`. Later `t->y` is used to access the heap object. The program also defines a pointer `p` that refers to

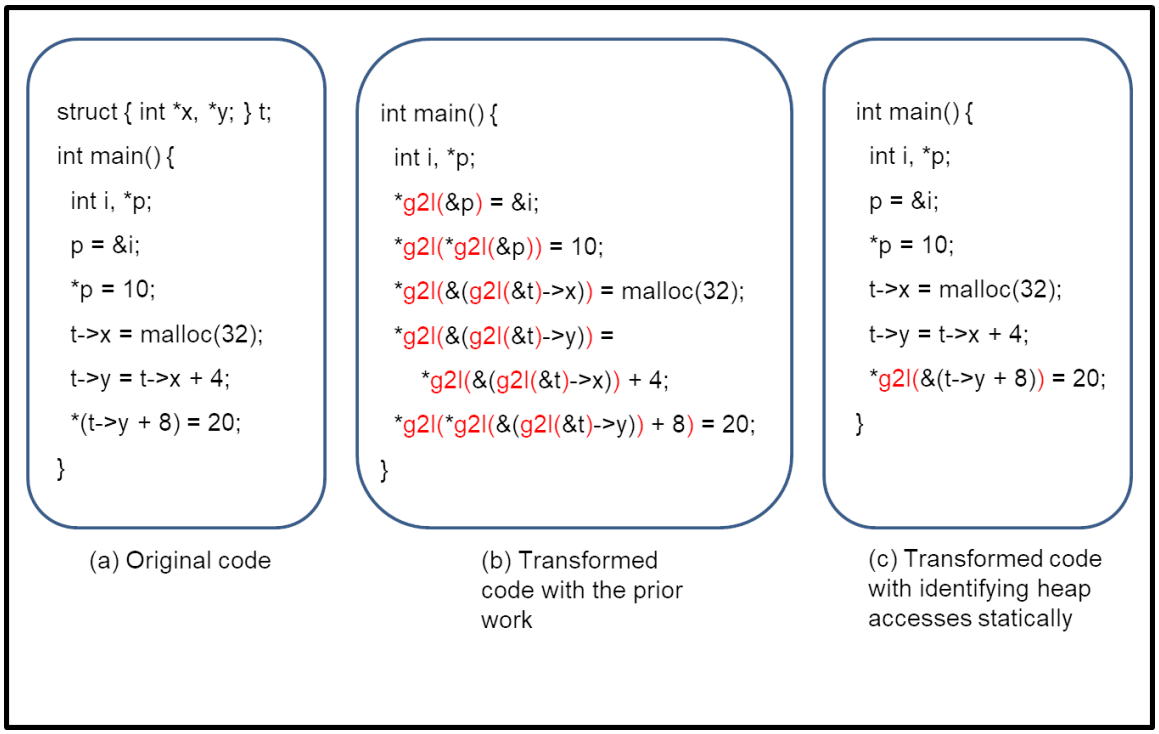

Figure 6.19: The Previous Approach Inserts *g2l* Before Every Memory Access, While Ours Tries to Identify Heap Accesses Statically and Skip Unnecessary *g2l*s.

a stack variable. Even though only `t->x` and `t->y` are pointing to heap data in this program, the previous heap management technique proposed in Bai and Shrivastava (2013) would insert a *g2l* call at every memory access as shown in Figure 6.19(b), including memory accesses to stack and global data (via `p` and `t` respectively). On the other hand, with static heap access detection, we only insert *g2l* for heap data accesses. Algorithm 6 illustrates the logic of inserting *g2l* functions.

To find out heap accesses, we developed an algorithm 7 to identify heap pointers. This algorithm identifies not only pointers that directly point to heap objects created by memory allocation (e.g., *malloc* or *calloc*), but also their aliases. The analysis starts

Algorithm 6 *g2l* Function Insertion

```
1: function InsertManagementFunction(Function F)
2:     for each instruction I in F do
3:         if I is a load or store to any heap pointer or one of its aliases then
4:             insert a g2l call at the heap access
5:         end if
6:     end for
7: end function
```

at `getHeapPtr`. In this procedure, the analysis first executes `getAlloc` procedure, taking *main* function as an input (line 2). The `getAlloc` procedure identifies all the invocation of memory allocators in the input function `F`, and records the pointers that are used to store the created heap objects (line 8 and 9). If `F` calls any other functions `F'`, `getAlloc` recursively accesses and identifies the memory allocations in `F'` (line 11 and 12). Once all the heap objects that were created by memory allocation are identified, the analysis continues to identify all the possible alias of these heap pointers by executing the `getAlias` procedure on *main* function (line 4). The `getAlias` procedure goes through each instruction in the input function `F`, recognizing any instruction that performs pointer arithmetic on a heap pointer and assigns the result to another pointer. The destination pointer of such an instruction is identified as an alias of the heap pointer. Similar to the `getAlloc` procedure, in case `F` calls any other function `F'`, the `getAlias` procedure recursively calls itself on `F'` to identify aliases created in `F'`. Since each iteration of the `getAlias` procedure may recognize new aliases, this procedure is repeated until no new aliases can be recognized (line 3 to 5).

Once all heap pointers are recognized, we can identify heap accesses and insert *g2l* functions. All the memory accesses (i.e. loads ans stores) via any of the heap pointers

Algorithm 7 Identify heap pointers

```
1: function getHeapPtr
2:     getAlloc(main)
3:     repeat
4:         getAlias(main)
5:     until cannot find new aliases
6: end function
7: function getAlloc(Function F)
8:     for each instruction inst in F do
9:         if inst is a call to any memory allocator then
10:            Record destination pointer P as a heap pointer
11:        else
12:            if inst is a call to any user function F' then
13:                getAlloc(F')
14:            end if
15:        end if
16:    end for
17: end function
18: function getAlias(Function F)
19:    for each instruction inst in F do
20:        if inst is an assignment statement with one operand P be a heap pointer then
21:            Record destination pointer P' as an alias of P
22:        else
23:            if inst is a call to any user function F' then
24:                getAlias(F')
25:            end if
26:        end if
27:    end for
28: end function
```

that were identified in Algorithm 7 are considered as potential heap accesses. A $g2l$ function is inserted right before the memory instruction to translate the memory address to an SPM address. The SPM address is then used to replace the original memory address in future usages.

There are cases when the compiler cannot determine whether a pointer refers to heap data. In Figure 6.20(a), the pointer `c` can either refer to heap data or

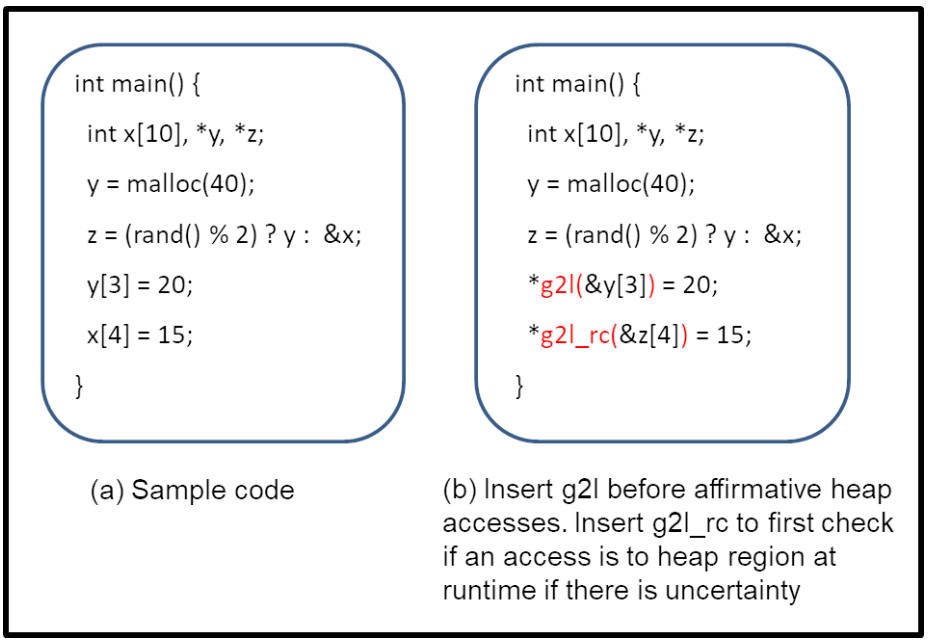

Figure 6.20: When It Cannot Be Determined at Compile-time Whether There is a Heap Access, We Check it at Run-time.

stack data, depending on the outcome of function `rand`. To cope with such cases, a new management function called *g2l_rc* is introduced to check if the memory access happens at heap region. When the compiler is very sure that the next instruction accesses heap data, the *g2l* function is called, which does not have any runtime checking. If the compiler cannot tell whether there would be a heap access, *g2l_rc* is called instead. Otherwise, if the compiler can determine that no access to heap data would happen, no management function will be inserted. Figure 6.20(b) shows the above logic. *g2l* is called before accessing the data referred by pointer `y`, because it can be told at the compile time that `y` points to heap data . *g2l_rc* is invoked before accessing `z`, because it might refer to heap data. No heap management function is added when accessing `x` since it can be decided at compile time that `x` accesses stack data.

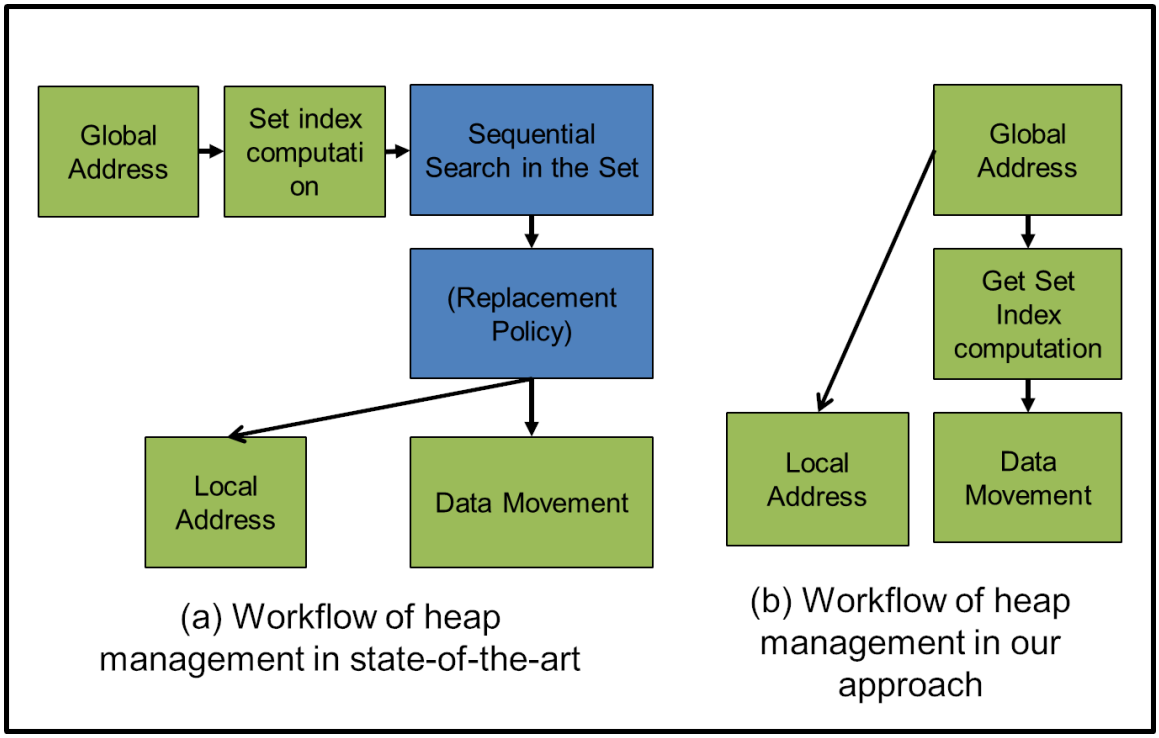

Figure 6.21: Comparison of Heap Management Workflow.

Management logic simplification

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

Since this process happens within each management function call, it is performance critical. With a direct-mapped cache on software, this process can be noticeably simplified to execute much fewer instructions at runtime, compared to using a set-associative cache. Figure 6.21(a) and Figure 6.21(b) show two examples using the previous approach and our approach respectively.The previous approach as illustrated in Figure 6.21(a) calculates set index with Equation (1). It then searches the corresponding set for the requested data block. Once the data block is found, the SPM address is computed with Equation (2). To be noted that this equation requires indexes of both the set and the entry in the set, which in turn rely on the calculation of the SPM address through the sequential searching shown in Figure 6.21(a). On the other hand, our approach in Figure 6.21(b) simplifies the calculation of the set index of a memory address into $set\_index = global\_addr >> log(block\_size)\%set\_num$. Since each set has only one entry, sequential searching is not necessary. The software can simply go ahead and calculate the final SPM address as $spm\_addr = spm\_base + mem\_addr\%(set\_num * block\_size)$. In addition, the calculation of SPM does not depend on any previous steps. Elimination of such dependency allows the compiler to better parallelize the management functions.

De-duplicating and Combining Management Calls

Our technique further reduced management overhead by de-duplicating management functions ($g2l$). The state-of-the-art technique divided SPM into two memory regions as heap management table and data region. Our approach makes similar usage of SPM space. Every $g2l$ thus contains some common instructions which is to load the start address of the heap management table and data region at the beginning of its execution, before executing any other call-specific instructions. However, when heap managements are frequently invoked, those common instructions are executed

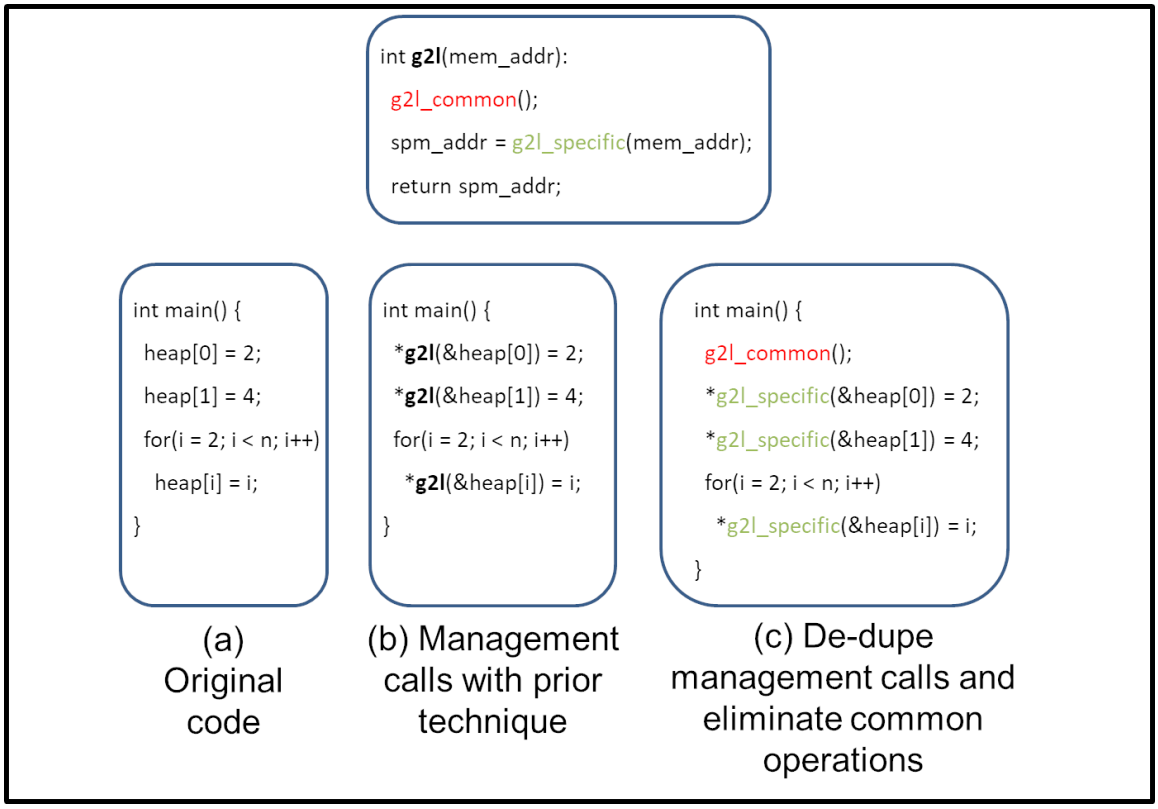

Figure 6.22: De-dupe Management Calls and Move Common Operations to the Beginning of the Caller Function.

repeatedly. To avoid unnecessary execution of those common instructions, we hoist those instructions outside of the *g2l* function and execute it only at the very beginning.

Figure 6.22 illustrates the idea. Figure 6.22(a) shows the original code. Figure 6.22(b) is the transformed code before de-duplication. Each *g2l* call first executes the common instructions redundantly, and then execute specific instructions for that call. We represent the common instructions and specific instructions in a *g2l* with function $g2l\_common$ and $g2l\_specific$ respectively in the example, but they are plain instructions in the actual implementation. In Figure 6.22(c), we de-dupe the *g2l* calls,

Algorithm 8 De-dupe Heap Management

```
1: function inlineManagementFunction(Function F)
2:     for each function F do
3:         if F has any call to g2l then
4:             insert common operations of g2l at the beginning of F
5:             for each g2l call I in F do
6:                 de-dup the call
7:                 remove the common operations
8:             end for
9:         end if
10:    end for
11: end function
```

move and execute the common instructions at the beginning of the caller function. After the optimization, only call-specific instructions are executed at where a *g2l* was called. While this optimization should definitely improve performance, its importance is maximized when *g2l* was originally called within loop nests, as this example shows —instead of repeatedly and excessively executing the common steps in a loop nest, moving these common instructions to be outside can significantly reduce such overhead.

The algorithm of this optimization is shown in Algorithm 8. In addition, at compile time, the modified compiler goes through every function in the program, de-duplicating *g2l* calls with call-specific instructions, and moves the common instructions to the beginning of the function.

Adjusting Block Size for Embedded Applications

All the aforementioned optimizations are generic, and thus are useful for all applications. However, in embedded systems, where profiling information can be obtained,

heap data management can be further optimized. Depending on the type of cache misses an application suffers from, the block can be statically adjusted to avoid these misses.

When the capacity and associativity of a cache are given, the size of block size decides the number of sets. Different choices of block size may end up causing drastically different performance. We can therefore analyze the access pattern and find a block size that can achieve good performance. When a program is susceptible to cache thrashing, we can decrease block size to lower the chance of such undesirable situation. Cache thrashing refers to excessive conflict cache misses that happens when multiple main memory locations competing for the same cache blocks. It may happen when more than two heap objects with aggregate types (e.g., arrays) are accessed within the same loop. On the other hand, we can increase block size to improve spatial locality under certain circumstances.

We proposed a heuristic that goes through all innermost loops in a program and adjusts block size based on profiling. Whenever it identifies more than two heap objects are accessed within the loop, it reduces the block size to increase the number of sets to avoid cache thrashing; otherwise, it increases the block size to increase spatial locality.

6.3.3 Experimental Results

Experimental Setup

Our techniques are implemented as intermediate representation (IR) passes on LLVM 3.8 Lattner and Adve (2004).Benchmarks were compiled with different heap management techniques and were ran on Gem5 Binkert *et al.* (2011). If not stated explicitly, the block size in the software cache is set to 64 bytes by default.

Table 6.7: Maximum Heap Usage of Benchmarks

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

We emulated the SMM architecture on Gem5. the SPMs were simulated by modifying the linker script and reserving part of the memory address space. A DMA instruction is implemented to copy data between the SPM and the main memory. DMA cost is modeled as a constant startup time and the time for actual data movement. The startup time is set to 291 cycles, and the rate for transferring data is set to 0.24 cycles/byte. The CPU frequency is set to 3.2 GHz. All these parameters are based on the IBM Cell processor Kistler *et al.* (2006).

The proposed techniques were evaluated across Mibench benchmark suite Guthaus *et al.* (2001). Table 6.7 lists the maximum usage of heap data in the benchmarks, i.e., the maximum sum of sizes of heap objects at any moment. Benchmarks that have zero heap usage do not have any heap accesses.

Execution Time Reduction

As shown in figure 6.23, in overall, our approach can reduce execution time by 80% on average with the first three generic optimizations, i.e., without adjusting block size. When we apply all four optimizations, the execution time can be reduced by 83% on average.

compile-time heap access detection has been proved to contribute the largest reduction of execution time, as shown in Figure 6.23. This technique is especially

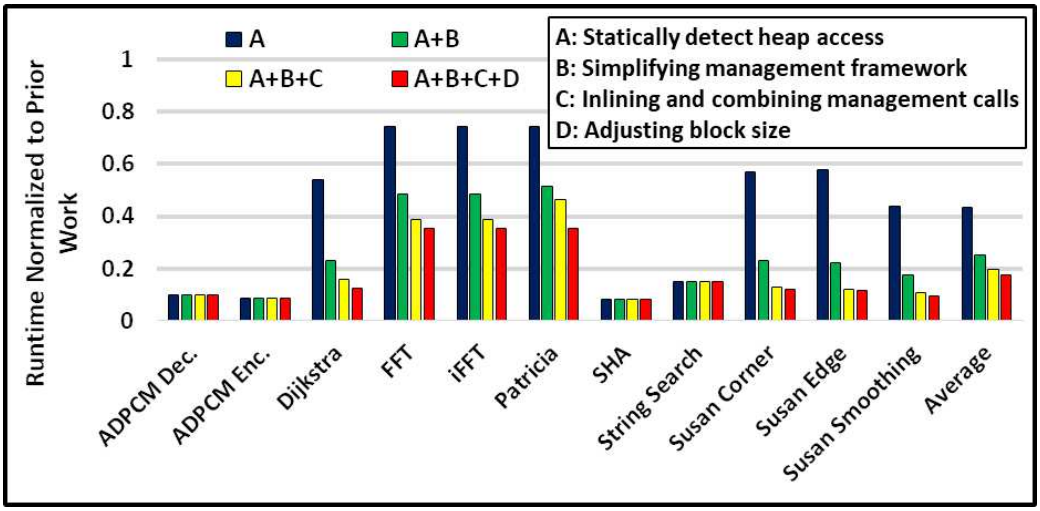

Figure 6.23: The Execution Time of Our Approach Normalized to the Previous Work with Optimizations Incrementally Added.

effective on benchmarks that do not have any heap accesses, i.e., `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search`. Overall, it reduces the execution time by 57% on average, due to reduced management calls and less executed instructions in each call. Table 6.8 shows the number of calls to the $g2l$ function before and after statically detecting heap accesses over state-of-the-art. The number of management calls are significantly reduced across all the benchmarks, and they are completely eliminated in benchmarks that do not have any heap access.

Another reason that compile-time heap access detection can reduce management overhead significantly is that it can also eliminate runtime checking at $g2l$s, and thus reduces the number of instructions. Table 6.9 shows the average number of instructions each $g2l$ executes under different cases, with different optimization techniques being incrementally applied one by one. There are 3 possible cases when a $g2l$ function is called: a cache hit, a cache miss with an unmodified data block to be evicted, and a cache miss with a dirty data block to be evicted. The memory access may

either be a read access or a write access, which added up to 6 different combinations in overall. The table clearly shows a constant difference of 6 instructions between the Previous Work column and the Statically Detecting Heap Accesses column across all the cases.

Replacing the 4-way set-associative cache with a direct-mapped software caches has been proved to reduce execution time by 42% on average (on top of compile-time heap access detection). It can be observed that the average dynamic instruction count of *g2l* calls has been reduced significantly under all the situations, as shown in Table 6.9. For example, the average instructions executed in the sixth case is reduced from 166 to 58 after simplifying management framework. Since a direct-mapped cache causes more cache misses compared to a 4-way set-associative cache, we consider increased cache misses as part of management penalty. Figure 6.24 shows the reduced CPU cycles due to less management instructions normalized to the increased CPU cycles caused by increased cache misses. Experimental results show that compared with the performance gained by simplified management instructions, the increased

Table 6.8: Number of *g2l* Calls with and without Heap Access Detection Technique

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

Table 6.9: Instructions Executed per *g2l* with and without Different Optimization Techniques

Case	Previous Work	Statically Detect Heap Accesses	Simplify *g2l*	de-dupe and Combine *g2l*
read hit	52	46	19	8
write hit	59	53	23	10
read miss w/o write-back	145	139	41	36
write miss w/o write-back	145	139	44	37
read miss w/ write-back	172	166	58	45
write miss w/ write-back	172	166	58	45

cycles caused by increased cache misses is negligible. For example, in `Patricia`, the reduced cycles are more than 10000000 times than the increased cycles.

De-dupe and combing management calls can further reduce execution time by 21% thanks to the elimination of redundant operations. For example, as Table 6.9 shows, the average instructions executed in the sixth case is reduced from 166 to 58 after simplifying management framework, and is further reduced from 58 to 45 after de-dupe and combing management calls. To be noted that we apply this optimization after statically detecting heap accesses. So if heap management calls are all eliminated after that step, de-dupe and combining management calls will not improve performance. For example, the management calls of `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search` are reduced to 0 after the compiler statically finds out there are no heap accesses in these benchmarks. Those benchmark would not benefit from this optimization.

The block size was set to 64 bytes by default. When analyzing the effectiveness of optimally adjusting block size, we analyzed benchmark profiles and adjusted the block size from 16 bytes to 1024 bytes as needed. The decision on block size was based on profiling information. Adjusting block size could further reduce execution time by 11% (on top of the previous three optimization techniques).

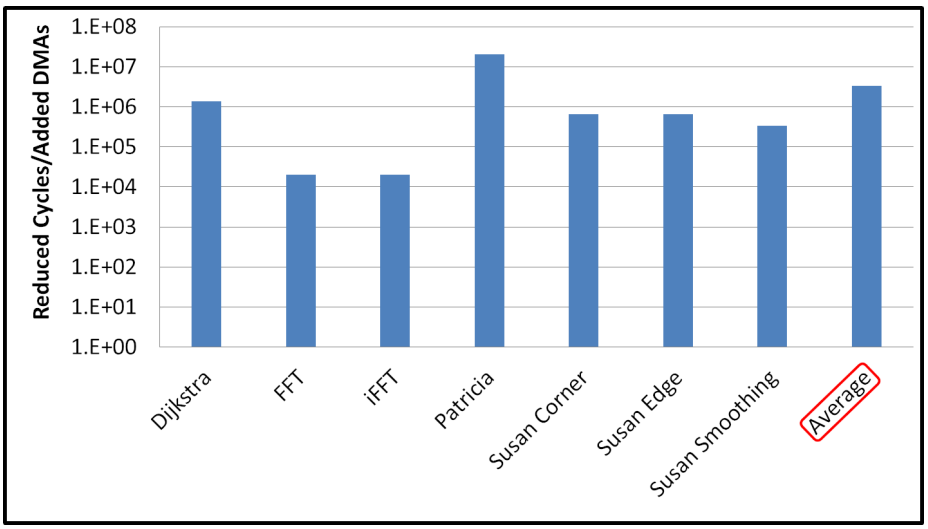

Figure 6.24: A Direct-mapped Cache other than a 4-way Set-associative Cache Reduces More Execution Time Thanks to Simplified Management Functions, Compared to the Extra Time Introduced due to Increased Cache Misses.

6.4 Compiler and Runtime Infrastructure

In the era of transitioning the intelligence from hardware to software, a compiler that automatically performs the insertion of branch hinting instructions and efficient data management of the application through automatic analysis is highly expected. This is the objective of our compiler and runtime system, and this dissertation. The advantages of such a compiler based approach include 1) programmability improvement: developers can write their code as if hardware caching is provided, so that they can focus on software logistics which eventually expedites the development cycle. 2) portability enhancement: the same application code can be reused on different versions or even different SPM-based architectures, with slight modification of architecture configuration to the compiler. 3) delivery of comparable or even better application performance than hardware caching. With deliberately designed compiler analyses, we can greatly reduce the overhead incurred by data movements between

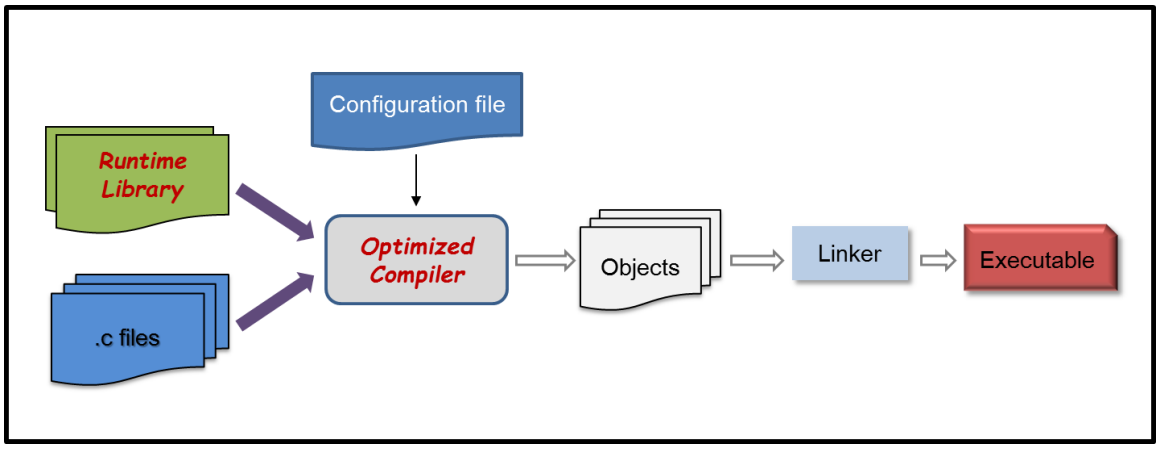

Figure 6.25: General Compilation Flow for Data Management on SMM Architectures

the local memory and main memory in applications.

Despite of its attractivenesses, a satisfactory compiler-based data management for SMM architectures is not that intuitive to design, since finding the optimal solution to minimize the memory transfers between the local memory and main memory is an intractable problem. Instead, we develop heuristics that will deliver high-quality results with reasonable compilation cost. The general flow of our compiler-based approach is shown in Figure 6.25, which comprise of a optimized compiler and the corresponding runtime library. Our compiler takes in source files written for the cache-based architecture, our data management libraries and a configuration file that indicates the size of local memory, performs necessary compiler analyses which inserts memory transfer requests (typically DMA instructions), and generates an executable that can be run on an SMM architecture.

Table 6.10 presents all APIs for data and code management on SMM architectures. *_sstore* and *_sload* functions manage function stack frames. *__ovly_load* is in charge of loading "to-be-execute" instructions from main memory to the local scratchpad memory. The last five functions process heap data in applications. One thing deserves to be mentioned is that all these functions will be automatically placed by our compiler

Table 6.10: Runtime Library for Data and Code Management

Category	Library	Functionality
stack	*_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
code	*__ovly_load*	load function instructions from main memory to the local memory
heap	*_malloc*	allocates space in local memory and main memory, and eventually returns a global address
	_free	frees space in main 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

at the proper locations. The function implementation details and their places to be inserted in the managed applications are explained in Section 6.1, Section 6.2, and Section 6.3.

Chapter 7

SUMMARY

Designing manycore architectures requires us to completely redesign the processors. Simply increasing the number of cores will not work. This is because power consumption increases cubically with frequency of operation, while most computing systems are limited by power, energy and thermal constraints. High performance computing centers and data centers are designed with the constraint of total power draw, embedded platforms are often designed around battery capacity, and the rest systems in the middle are designed with thermal constraints.

Software Managed Manycore (SMM) architectures, which shift the intelligence from run-time to compile-time, have emerged as a solution to scaling the manycore processors. However, it is not trivial to design SMM architectures. Once hardware components are removed from hardware, the corresponding logic has to be implemented in the software. The software intelligent has to be not only correct, but also efficient. Reducing software overhead becomes a critical part in the designing of SMM architectures. This dissertation explored the design of SMM in two aspects, the branch prediction mechanism, and data management mechanism. On one hand, an SMM architecture removes hardware branch predictor and merely uses software branch hinting. On the other hand, caches are removed on each core and are replaced with local scratchpad memories.

This dissertation presents compiler-based automatic techniques that help to better analyze the application, understand the control flow, and direct the hardware to execute in an optimized way Lu *et al.* (2011, 2013, 2015); Bai *et al.* (2013). Our techniques improved the performance and overcame the gap left by the absence of the

hardware component from the perspectives of: 1) software branch hinting, 2) smart stack data management, 3) efficient code mapping, and 4) efficient heap management.

Extensive experiments have been conducted to evaluate the proposed techniques. Our efficient software branch hinting technique can reduce the branch penalty as much as 35.4% over the previous approach. The smart stack data management technique can reduce the overhead by 13X over the state-of-the-art stack data management technique Bai *et al.* (2011). The efficient code mapping can reduce runtime in more than 80% of the cases, and by up to 20% on our set of benchmarks, compared to the state-of-the-art code assignment approach Jung *et al.* (2010). The efficient heap management technique can reduce execution time by 80% on average.

REFERENCES

"ARM Architecture version 5 (ARMv5TE)" (http://www.arm.com/, 2001).

"GNU Toolchain 4.1.1 and GDB for the Cell BE's PPU/SPU", `http://www.bsc.es/plantillaH.php?cat_id=304` (2005).

"IBM Full-System Simulator for Cell BE", `http://www.alphaworks.ibm.com/tech/cellsystemsim` (2006).

Abts, D., S. Scott and D. J. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1", in "Proc. of IPDPS", pp. 11.2– (2003).

Agarwal, A. and M. Levy, "The Kill Rule for Multicore", in "Proc. of DAC", pp. 750–753 (2007).

Angiolini, F., F. Menichelli, A. Ferrero, L. Benini and M. Olivieri, "A Post-compiler Approach to Scratchpad Mapping of Code", in "Proc. of CASES", pp. 259–267 (2004).

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

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

Bai, K. and A. Shrivastava, "Heap Data Management for Limited Local Memory (LLM) Multi-core Processors", in "Proc. of CODES+ISSS", pp. 317–326 (2010).

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

Bai, K., A. Shrivastava and S. Kudchadker, "Stack Data Management for Limited Local Memory (LLM) Multi-core Processors", in "Proc. of ASAP", pp. 231–234 (2011).

Baker, M. A., A. Panda, N. Ghadge, A. Kadne and K. S. Chatha, "A Performance Model and Code Overlay Generator for Scratchpad Enhanced Embedded Processors", in "Proc. of CODES+ISSS", pp. 287–296 (2010).

Balakrishnan, M., P. Marwedel, L. Wehmeyer, N. Grunwald, R. Banakar and S. Steinke, "Reducing Energy Consumption by Dynamic Copying of Instructions onto On-chip Memory", in "Proc. of ISSS", pp. 213–218 (2002).

Ball, T. and J. R. Larus, "Branch Prediction for Free", in "Proc. of PLDI", pp. 300–313 (ACM, New York, NY, USA, 1993).

Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel, "Scratchpad Memory: Design Alternative for Cache on-chip Memory in Embedded Systems", in "Proc. of CODES+ISSS", pp. 73–78 (2002).

Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, "The Gem5 Simulator", SIGARCH Comput. Archit. News (2011).

Borkar, S., "Major Challenges to Achieve Exascale Performance", in "Salishan Conference on High-Speed Computing", (2009).

Bournoutian, G. and A. Orailoglu, "Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive Mobile Processors", in "Proc. of CODES+ISSS", pp. 89–98 (2011).

Briejer, M., C. Meenderinck and B. Juurlink, "Extending the Cell SPE with Energy Efficient Branch Prediction", in "Proc. of EuroPar", pp. 304–315 (2010).

Chaiken, D., J. Kubiatowicz and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme", in "Proc. of ASPLOS", pp. 224–234 (1991).

Choi, B., R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism", in "Proc. of PACT", pp. 155–166 (2011).

Dinechin, B. D. D., P. G. de Massas, G. Lager, C. Lger, B. Orgogozo, J. Reybert and T. Strudel, "A Distributed run-time Environment for the Kalray MPPA-256 Integrated Manycore Processor", Procedia Computer Science (2013).

Dominguez, A., S. Udayakumaran and R. Barua, "Heap Data Allocation to Scratchpad memory in Embedded Systems", J. Embedded Comput. **1**, 4, 521–540 (2005).

Egger, B., C. Kim, C. Jang, Y. Nam, J. Lee and S. L. Min, "A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization", in "Proc. of CASES", pp. 223–233 (2006a).

Egger, B., J. Lee and H. Shin, "Scratchpad Memory Management for Portable Systems with A Memory Management Unit", in "Proc. of EMSOFT", pp. 321–330 (2006b).

Eichenberger, A. E., 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 (2006).

Eichenberger, A. E., K. O'Brien, K. 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 and M. Gschwind, "Optimizing Compiler for the CELL Processor", in "Proc. of PACT", pp. 161–172 (2005).

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

Garcia-Guirado, A., R. Fernandez-Pascual, A. Ros and J. Garcia, "Energy-Efficient Cache Coherence Protocols in Chip-Multiprocessors for Server Consolidation", in "Proc. of ICPP", pp. 51–62 (2011).

Goodman, J. R., "Using Cache Memory to Reduce Processor-memory Traffic", in "Proc. of ISCA", pp. 255–262 (1998).

Gschwind, M., H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe and T. Yamazaki, "Synergistic Processing in Cells Multicore Architecture", IEEE Micro **26**, 2, 10–24 (2006).

Gustafsson, J., A. Betts, A. Ermedahl and B. Lisper, "The Mälardalen WCET Benchmarks – Past, Present and Future", pp. 137–147 (2010).

Guthaus, M. R., J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "Mibench: A Free, Commercially Representative Embedded Benchmark Suite", Proc. of the Workload Characterization pp. 3–14 (2001).

Hardware, T., "Raw Performance: SiSoftware Sandra 2010 Pro (GFLOPS)", (2010).

Heinrich, M., V. Soundararajan, J. Hennessy and A. Gupta, "A Quantitative Analysis of the Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols", IEEE Trans. Comput. **48**, 2, 205–217 (1999).

Hofstee, H., "Power efficient processor architecture and the Cell processor", in "International Symposium on High-Performance Computer Architecture", pp. 258–262 (2005).

Howard, J., S. Dighe, S. 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. 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).

Hung, S.-H., C.-H. Tu and W.-L. Yang, "A Portable, Efficient Inter-core Communication Scheme for Embedded Multicore Platforms", J. Syst. Archit. **57**, 2, 193–205 (2011).

IBM, "Cell Broadband Engine Programming Handbook including PowerXCell 8i", `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED` (2007).

IBM, "Programmer's Guide: Software Development Kit for Multicore Acceleration Version 3.1", Tech. rep. (2008).

IBM, "IBM Cell SDK 3.1", `http://www.ibm.com/developerworks/power/cell` (2009).

Intel, "Intel Core i7 Processor Extreme Edition and Intel Core i7 Processor Datasheet, Volume 1", in "White paper", (2010).

Intel, *Intel Core i7-7700K Processor* (http://ark.intel.com/products/97129, 2017).

Itanium, "Dual-Core Intel Itanium Processor 9000 and 9100 Series", `http://download.intel.com/design/itanium/downloads/314054.pdf` (2007).

Janapsatya, A., A. Ignjatović and S. Parameswaran, "A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric", in "Proc. of ASP-DAC", pp. 612–617 (2006).

Jang, C., J. Lee, B. Egger and S. Ryu, "Automatic Code Overlay Generation and Partially Redundant Code Fetch Elimination", ACM Trans. Archit. Code Optim. **9**, 2, 10:1–10:32 (2012).

Jiménez, D. A. and C. Lin, "Dynamic Branch Prediction with Perceptrons", in "HPCA", p. 197 (2001).

Jung, S. C., A. Shrivastava and K. Bai, "Dynamic Code Mapping for Limited Local Memory Systems", in "Proc. of ASAP", pp. 13–20 (2010).

Kahle, J. A., M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. Shippy, "Introduction to the Cell Multiprocessor", IBM J. Res. Dev. **49**, 589–604 (2005).

Kalamatianos, J. and D. R. Kaeli, "Improving the Accuracy of Indirect Branch Prediction via Branch Classification", SIGARCH Comput. Archit. News **27**, 1, 23–26 (1999).

Kandemir, M. and A. Choudhary, "Compiler-directed Scratch Pad Memory Hierarchy Design and Management", in "Proc. of DAC", pp. 628–633 (2002).

Kannan, A., A. Shrivastava, A. Pabalkar and J.-e. Lee, "A Software Solution for Dynamic Stack Management on Scratch Pad Memory", in "Proc. of ASP-DAC", pp. 612–617 (2009).

Kistler, M., M. Perrone and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed", IEEE Micro (2006).

Kolson, D., A. Nicolau and N. Dutt, "Elimination of Redundant Memory Traffic in High-Level Synthesis", IEEE Trans. on Comp-aided Design **15**, 1354–1363 (1996).

Kongetira, P., K. Aingaran and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor", IEEE Micro **25**, 2, 21–29 (2005).

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

Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, "The Directory-based Cache Coherence Protocol for the DASH Multiprocessor", in "Proc. of ISCA", pp. 148–159 (1990).

Li, L., H. Feng and J. Xue, "Compiler-directed Scratchpad Memory Management via Graph Coloring", ACM Trans. Archit. Code Optim. **6**, 3, 1–17 (2009).

Lin, J.-P., J. Lu, J. Cai and A. Shrivastava, "Efficient heap data management on software managed manycore architectures", in "Proceedings of 2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)", (2019).

Lu, J., K. Bai and A. Shrivastava, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)", in "Proc. of DAC", pp. 149–156 (2013).

Lu, J., K. Bai and A. Shrivastava, "Efficient Code Assignment Techniques for Local Memory on Software Managed Multicores", ACM Trans. Embed. Comput. Syst. **14**, 4, 71:1–71:24 (2015).

Lu, J., Y. Kim, A. Shrivastava and C. Huang, "Branch Penalty Reduction on IBM Cell SPUs via Software Branch Hinting", in "Proc. of CODES+ISSS", pp. 355–364 (ACM, 2011).

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

O'Brien, K., "Issues and Challenges in Compiling for the CBEA", in "Proc. of LCTES", pp. 134–134 (2007).

Pabalkar, A., A. Shrivastava, A. Kannan and J. Lee, "SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories", in "Proc. of HPC", pp. 569–582 (2008).

Parikh, D., K. Skadron, Y. Zhang, M. Barcella and M. R. Stan, "Power Issues Related to Branch Prediction", in "HPCA", p. 233 (2002).

Poletti, F., P. Marchal, D. Atienza, L. Benini, F. Catthoor and J. M. Mendias, "An Integrated Hardware/Software Approach for Run-time Scratchpad Management", in "Proc. of DAC", pp. 238–243 (2004).

Rotta, R., T. Prescher, J. Traue and J. Nolte, "In-Memory Communication Mechanisms for Many-Cores – Experiences with the Intel SCC", (2012).

Simoni, R. and M. Horowitz, "Dynamic Pointer Allocation for Scalable Cache Coherence Directories", in "Proc. of ISSMM", pp. 72–81 (1991).

Sinharoy, B. and S. W. White, "Use of Software Hint for Branch Prediction in the Absence of Hint Bit in the Branch Instruction", `http://www.freepatentsonline.com/6971000.html` (2005).

Smith, J. E., "A Study of Branch Prediction Strategies", in "Proc. of ISCA", pp. 135–148 (1981).

Sodani, A., R. Gramunt, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal and Y. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product", IEEE Micro **36**, 2, 34–46 (2016).

Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessors", Computer **23**, 6, 12–24 (1990).

Stephen, A. S., S. Felix, V. Krishnan and Y. Sazeides, "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor", in "ISCA", pp. 295–306 (2002).

Tilera, "TILE PROCESSOR ARCHITECTURE OVERVIEW FOR THE TILEPRO SERIES", `http://www.mellanox.com/repository/solutions/tile-scm/docs/UG120-Architecture-Overview-TILEPro.pdf` (2013).

Totoni, E., B. Behzad, S. Ghike and J. Torrellas, "Comparing the Power and Performance of Intel's SCC to state-of-the-art CPUs and GPUs", in "Proc. of ISPASS", pp. 78–87 (2012).

Udayakumaran, S., A. Dominguez and R. Barua, "Dynamic Allocation for Scratchpad memory Using Compile-time Decisions", Trans. on Embedded Computing Sys. **5**, 2, 472–511 (2006).

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

Wagner, T. A., V. Maverick, S. L. Graham and M. A. Harrison, "Accurate Static Estimators for Program Optimization", in "Proc. of PLDI", pp. 85–96 (ACM, New York, NY, USA, 1994).

Wu, Y. and J. R. Larus, "Static Branch Frequency and Program Profile Analysis", in "Proc. of MICRO", pp. 1–11 (ACM, New York, NY, USA, 1994).

Xu, Y., Y. Du, Y. Zhang and J. Yang, "A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs", in "Proc. of ICS", pp. 285–294 (2011).