ABSTRACT

Title of dissertation: Symbiotic Subordinate Threading (SST)

Rania Mameesh, Doctor of Philosophy, 2007

Dissertation directed by: Dr Manoj Franklin
Electrical and Computer Engineering Department

Integration of multiple processor cores on a single die, relatively constant die sizes, increasing memory latencies, and emerging new applications create new challenges and opportunities for processor architects. How to build a multi-core processor that provides high single-thread performance while enabling high throughput through multi-programming? Conventional approaches for high single-thread performance use a large instruction window for memory latency tolerance, which requires large and complex cores. However, to be able to integrate more cores on the same die for high throughput, cores must be simpler and smaller.

We present an architecture that obtains high performance for single-threaded applications in a multi-core environment, while using simpler cores to meet the high throughput requirement. Our scheme, called Symbiotic Subordinate Threading (SST), achieves the benefits of a large instruction window by utilizing otherwise idle cores to run dynamically constructed subordinate threads (a.k.a. *helper threads*) for the individual threads running on the active cores.

In our proposed execution paradigm, the subordinate thread fetches and pre-

processes instruction streams and retires processed instructions into a buffer for the main thread to consume. The subordinate thread executes a smaller version of the program executed by the main thread. As a result, it runs far ahead to warm up the data caches and fix branch miss-predictions for the main thread. In-flight instructions are present in the subordinate thread, the buffer, and the main thread, forming a very large effective instruction window for single-thread out-of-order execution. Moreover, using a simple technique of identifying the subordinate thread non-speculative results, the main thread can integrate the subordinate thread's non-speculative results directly into its state without having to execute their corresponding instructions. In this way, the main thread is sped up because it also executes a smaller version of the program, and the total number of instructions executed is minimized, thereby achieving an efficient utilization of the hardware resources. The proposed SST architecture does not require large register files, issue queues, load/store queues, or reorder buffers. In addition, it incurs only minor hardware additions/changes. Experimental results show remarkable latency-hiding capabilities of the proposed SST architecture, outperforming existing architectures that share similar high-level microarchitecture.

We performed two extensions of our SST scheme, and came up with two additional microarchitectures. In the first extension, we developed a simple way to allow the subordinate thread be aware of its own speculation. A speculative-aware subordinate thread is capable of identifying instructions that are more likely to produce invalid values, and so may skip their execution. In the second extension, we allow a subordinate thread to have its own subordinate thread. The main thread

and multiple subordinate threads are arranged in a hierarchy based on the degree of their speculation, with the most speculative subordinate thread at the bottom of the hierarchy and the least speculative thread (the main thread) at the top of the hierarchy. This new microarchitecture, named Hierarchical Symbiotic Subordinate Threading, combines the benefit of the speed of highly speculative subordinate threads with the accuracy of not-too-speculative subordinate threads.

Symbiotic Subordinate Threading (SST)

by

Rania Mameesh

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Advisory Commmittee:

Dr Manoj Franklin, Chair/Advisor
Dr Amr Baz
Dr Charles Silio
Dr Donald Yeung
Dr Peter Petrov

© Copyright by

Rania Mameesh

2007

ACKNOWLEDGMENTS

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to thank my advisor, Professor Manoj Franklin for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past five years.

I would also like to thank my committee members, Dr Amr Baz, Dr Charles Silio, Dr Donald Yeung, and Dr Peter Petrov for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript.

I owe my deepest thanks to my family - my parents and brother who have always stood by me and guided me through my career, and have pulled me through against impossible odds at times. Words cannot express the gratitude I owe them.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Lastly, thank you all and thank God!

TABLE OF CONTENTS

List of Tables vi

List of Figures vii

1 **Introduction** **1**
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 4
- 1.3 Roadmap . . . 7

2 **Background** **8**
- 2.1 Single-Chip Multi-core Processors . . . 8
  - 2.1.1 Motivation for Building Single-Chip Multiprocessors . . . 9
  - 2.1.2 Single-Chip Multi-core Architecture Models . . . 11
- 2.2 Multithreading . . . 14
  - 2.2.1 Multiprogramming . . . 14
  - 2.2.2 Parallel Processing . . . 15
- 2.3 Subordinate Threading . . . 16
  - 2.3.1 Uses of Subordinate Threading . . . 17
  - 2.3.2 Subordinate Thread Construction Techniques . . . 19

3 **Symbiotic Subordinate Threading (SST) — The Concepts and Implementation Details** **25**
- 3.1 A Simple Methodology for Distilling The Subordinate Thread . . . 25
- 3.2 A Simple and Efficient Way of Pruning The Main Thread . . . 29
  - 3.2.1 Basic Idea . . . 29
  - 3.2.2 Skipping Non-Memory Instructions . . . 30
  - 3.2.3 Skipping Memory Accesses (Only LOAD Instructions) . . . 32
  - 3.2.4 An Example . . . 34
- 3.3 Communicating Subordinate Thread Results and Decoded Information to the Main Thread . . . 37
- 3.4 Putting it All Together: The SST Microarchitecture . . . 41
  - 3.4.1 Basic Operation . . . 43
  - 3.4.2 Memory System . . . 44
  - 3.4.3 Recovery of the Subordinate Thread from Miss-speculation . . 45

4 **Experimental Results of SST** **47**
- 4.1 Performance Evaluation of SST Against Slipstream Processor . . . 49
  - 4.1.1 Average IPC Improvement of SST . . . 50
  - 4.1.2 Instruction Distribution in The Main Thread . . . 52
  - 4.1.3 Less Work Done by the Subordinate Thread on Wrong Paths . 54
  - 4.1.4 Performance Improvement with a Highly Speculative Subordinate Thread Versus a Not-Too-Speculative Subordinate Thread 56
  - 4.1.5 Improvement in the Subordinate Thread L2 Cache Miss Rate 57

4.1.6 Improvement in the Main Thread L1 DCache Miss Rate . . . 59
4.1.7 Reduction in the Main Thread Branch Miss-predictions 60
4.2 Performance Evaluation of SST Against DCE 63
4.2.1 IPC Improvement of SST without Memory Symbiosis (100 Cycles for Main Memory Access) 64
4.2.2 IPC Improvement of SST with Memory Symbiosis (100 Cycles for Main Memory Access) 65
4.2.3 IPC Improvement of SST with Memory Symbiosis (300 Cycles for Main Memory Access) 66
4.2.4 Reduction in the Subordinate Thread L2 Cache Miss Rate . . 68

5 An Optimized Implementation of SST 70
5.1 A Partially Speculative-Aware Subordinate Thread 71
5.2 The Subordinate Thread Recovers from Miss-Speculation By Switching Roles with the Main Thread 75
5.3 New SST Microarchitecture 79
5.4 Experimental Results 79
5.4.1 IPC Improvement 82
5.4.2 Branch Miss-predictions in The Main Thread 84
5.4.3 Branch Miss-predictions in the Subordinate Thread 85
5.4.4 L2 Cache Miss Rate 87
5.4.5 Reduction in the Total Number of Executed Instructions . . . 88

6 HSST: Hierarchical Symbiotic Subordinate Threading 91
6.1 A Motivating Example 92
6.2 Implementation Details of HSST 96
6.2.1 Spawning Subordinate Threads 98
6.2.2 Distilling the Subordinate Thread 100
6.2.3 Result Integration 102
6.2.4 Recovering the Subordinate Thread Corrupted State 103
6.3 Experimental Results 103
6.3.1 Performance Improvement 105
6.3.2 Advantages of Result Integration 108
6.3.3 Improvement in L2 Cache Miss Ratio 112
6.3.4 Experimenting with More than Two Subordinate Threads . . 115

7 Related Work 117
7.1 SST and Run-ahead execution 117
7.2 SST and Leader/Follower Architectures 120
7.3 SST and Result Reuse 125
7.4 SST and Clustered Architecutures 126

8 Future Work 127
8.1 Making the Fastest Thread the Leader 127
8.2 Hybrid HSST Processor 128

8.3 Exploiting Program Behavior Changes Using Dual Thread Execution Models . . . 129
8.4 Division of Work . . . 130
8.5 Power Studies . . . 131
8.6 Simulation Work . . . 132

9 **Summary and Conclusions** **133**

Bibliography 137

LIST OF TABLES

4.1 Microarchitectural Simulation Parameters For Smaller Cores 48

4.2 Microarchitectural Parameters with Larger Cores 63

5.1 Microarchitectural Simulation Parameters for Old & New SST 81

6.1 HSST Microarchitectural Parameters 104

6.2 Superscalar Microarchitectural Parameters 105

LIST OF FIGURES

3.1 SST top level design. 26

3.2 Identifying the backward slice of a branch instruction. 27

3.3 RSB update scenarios. 32

3.4 MSB addressing. 33

3.5 (a) Loop example from benchmark *perl*; (b) Example of reducing the number of executed instructions by the main thread. 35

3.6 FIFO queue. 38

3.7 SST microarchitecture. 42

3.8 Fast recovery of the subordinate thread state. 43

4.1 % IPC improvement achieved with symbiotic subordinate threading (SST) over the slipstream processor (main thread does not skip instructions). (a) SST with low speculation subordinate thread, and main thread does not skip load instructions; (b) SST with low speculation subordinate thread, and main thread skips load instructions; (c) SST with high speculation subordinate thread, and main thread does not skip load instructions; (d) SST with high speculation subordinate thread, and main thread skips load instructions 51

4.2 Instruction distribution in main thread for two schemes: (a) SST with high speculative subordinate thread; (b) SST with not too speculative subordinate thread 54

4.3 Work done by the subordinate thread on wrong paths for four schemes: (a) Slipstream with a highly speculative subordinate thread; (b) SST with a highly speculative subordinate thread; (c) Slipstream with a not too speculative subordinate thread; (d) SST with a not too speculative subordinate thread. 55

4.4 Distribution of average L2 cache misses obtained with: (a) Single thread; (b) Slipstream processor; and (c) SST. 58

4.5 Main thread L1 dcache: (a) misses incurred and saved with SST when memory symbiosis is applied; and (b) accesses incurred and saved with SST when memory symbiosis is applied. 60

4.6 Main thread % of branch miss-predictions incurred when using: (a) The branch predictions obtained from a branch predictor for all branch instructions (single thread); (b) The branch predictions obtained from the subordinate thread for all branch instructions (slipstream); (c) The non-data-speculative branch outcomes of the subordinate thread, and the predictions obtained from the branch predictor for all other branch instructions (SST). 62

4.7 IPC obtained with memory latency 100 cycles for: (a) Single thread scheme; (b) DCE scheme; and (c) SST scheme (main thread consumes the results of the subordinate thread for only non-memory instructions. 65

4.8 IPC obtained with memory latency 100 cycles for: (a) Single thread scheme; (b) DCE scheme; and (c) SST scheme (main thread consumes the results of the subordinate thread for all types of instructions). . 66

4.9 IPC obtained with memory latency 300 cycles for: (a) Single thread scheme; (b) DCE scheme; and (c) SST scheme (main thread consumes the results of the subordinate thread for only non-memory instructions. 67

4.10 Distribution of average L2 cache misses obtained with memory latency 300 cycles for: (a) Single thread scheme; (b) DCE scheme; and (c) SST scheme (main thread consumes the results of the subordinate thread for only non-memory instructions). 67

5.1 Subordinate thread and main thread switch roles after recovery of the subordinate thread from miss-speculation. 76

5.2 New SST Microarchitecture. 78

5.3 IPC for 5 schemes: (a) Single thread (a superscalar that combines two cores in one); (b) DCE with speculative unaware subordinate thread; (c) SST with speculative unaware subordinate thread; (d) DCE with speculative-aware subordinate thread; (e) SST with speculative-aware subordinate thread. 82

5.4 Percentage IPC improvement over a single thread (a superscalar that combines two cores in one) for four schemes: (a) DCE with speculative unaware subordinate thread; (b) SST with speculative unaware subordinate thread; (c) DCE with speculative-aware subordinate thread; (d) SST with speculative-aware subordinate thread. . . . 84

5.5 Percentage branch miss-predictions incurred by the main thread for five schemes: (a) Single thread (a superscalar that combines two cores in one); (b) DCE with speculative-unaware subordinate thread; (c) SST with speculative-unaware subordinate thread; (d) DCE with speculative-aware subordinate thread; (e) SST with speculative-aware subordinate thread. 86

5.6 Percentage of incorrect branch outcomes of the subordinate thread for four schemes: (a) DCE with speculative-unaware subordinate thread; (b) SST with speculative-unaware subordinate thread; (c) DCE with speculative-aware subordinate thread; (d) SST with speculative-aware subordinate thread. 87

5.7 L2 cache miss rate (only complete misses) in the main thread for five schemes: (a) Single thread (a superscalar that combines two cores in one); (b) DCE with speculative-unaware subordinate thread; (c) SST with speculative-unaware subordinate thread; (d) DCE with speculative-aware subordinate thread; (e) SST with speculative-aware subordinate thread. 88

5.8 Distribution of skipped and executed instructions in the main thread and the subordinate thread for two schemes: (a) SST with speculative-unaware subordinate thread; (b) SST with speculative-aware subordinate thread. 89

6.1 Example from benchmark *perl* showing the code snippet for: (a) Main thread; (b) Subordinate thread of main-subA model; and (c) Subordinate thread of main-subB model. 93

6.2 Pros and cons of high and low speculation subordinate threads. . . . 95

6.3 HSST High Level Microarchitecture: (a) HSST similar to a cache hierarchy; (b) HSST block diagram; and (c) Components of Thread Controller (TC). 97

6.4 HSST detailed microarchitecture design. 99

6.5 IPC obtained for four schemes: (a) Single thread (superscalar); (b) SST with subA (main-subA); (c) SST with subB (main-subB); and (d) HSST with both subA and subB. 107

6.6 IPC obtained for four schemes: (a) Single thread (superscalar); (b) SST with subA (main-subA); (c) SST with subB (main-subB); and (d) HSST with both subA and subB. 107

6.7 Distribution of instruction outcomes in main thread for three schemes: (a) SST with subA (main-subA); (b) SST with subB (main-subB); and (c) HSST with subA and subB. 108

6.8 Average branch miss-predictions in main thread for four schemes: (a) Single thread (superscalar); (b) SST with subA (main-subA); (c) SST with subB (main-subB); and (d) HSST with both subA and subB. . . 110

6.9 Percentage of branch instructions that were a miss-prediction and the main thread obtained their correct outcomes from the subordinate thread, for three schemes: (a) SST with subA (main-subA); (b) SST with subB (main-subB); and (c) HSST with both subA and subB. . 110

6.10 L2 cache miss ratio in main thread for four schemes: (a) Single thread; (b) SST with subA (main-subA); (c) SST with subB (main-subB); and (d) HSST with both subA and subB. 112

6.11 IPC obtained for four schemes: (a) SST with a single subordinate thread; (b) HSST with two subordinate threads; (c) HSST with three subordinate threads; and (d) HSST with four subordinate threads. . . 113

6.12 Average branch miss-predictions in main thread for four schemes: (a) SST with a single subordinate thread; (b) HSST with two subordinate threads; (c) HSST with three subordinate threads; and (d) HSST with four subordinate threads. 113

6.13 Average incorrect branch results of four subordinate threads with different levels of speculation: (a) Subordinate thread at speculation level 1 (subA); (b) Subordinate thread at speculation level 2 (subB); (c) Subordinate thread at speculation level 3 (subC); and (d) Subordinate thread at speculation level 4 (subD). 114

6.14 L2 cache miss ratio in main thread for four schemes: (a) SST with a single subordinate thread; (b) HSST with two subordinate threads; (c) HSST with three subordinate threads; and (d) HSST with four subordinate threads. 114

Chapter 1

Introduction

1.1 Motivation

Recent trends in microarchitecture reveal a move towards multi-core architectures that can efficiently leverage the billion transistor chips promised by future technologies. All major high-performance microprocessor vendors have announced or are already selling chips with two to nine cores. Future generations of these processors will undoubtedly include more cores on a single chip multiprocessor (CMP) [1, 2]. In 2001, IBM introduced the dual-core POWER-4 [47] processor, and in 2004 it introduced the POWER-5 processor, in which each core supports 2-way simultaneous multithreading (SMT) [48]. In 2005 IBM introduced the Cell Broadband Engine Architecture known as Cell processor [72], which combined eight synergistic processor elements with a dual-issue POWER processor element. In 2004 Sun announced the Niagara processor [49], which included eight cores, each of which is a four-way SMT. AMD, Fujitsu, and Intel have also released their dual-core chip multiprocessors [50, 51, 52].

Multiprogrammed environments as well as parallel applications benefit the most out of multiple cores. However, the performance of individual serial programs does not improve and may even suffer a penalty because of increased contention for shared resources such as caches in a multi-core environment. Moreover, the cost and

complexity of software increases if applications are manually parallelized to obtain a benefit from multiple cores. Finally, many general purpose applications, that are easy to parallelize, exhibit limited scalability. Therefore, they may not be able to take advantage of additional cores beyond a certain point.

Improving the performance of single threads in a multi-core environment has proven to be difficult for several reasons. First, multi-core architectures favor simpler and smaller cores to address the application needs for parallelization and the power budget, which limits the opportunity to exploit the available ILP with wide-issue cores. Also, achieving high single-thread performance in the presence of relatively increasing memory latencies has traditionally required large and complex cores to sustain a large number of instructions in flight while waiting for memory. On the other hand, special-purpose hardware accelerators that are located outside the core can improve a thread's performance by eliminating control and memory bottlenecks (e.g. advanced branch predictors and data prefetchers), but they often result in significant chip area additions and additional complexity. In light of these trends, architectural techniques that allow the use of additional cores to speed up single threads are becoming an attractive alternative [31].

Subordinate threading is one such technique that utilizes multi-core architectures for single-thread performance because of its ability to overcome the hurdles imposed by unpredictable branches and long-latency memory accesses. The basic idea is to spawn *subordinate threads* (also called *helper threads*), which are shorter versions of the main thread that execute in parallel with the main thread. Because they are shorter, they advance faster than the main thread, and perform many useful

actions on behalf of the main thread, thereby speeding up the main thread computation. Useful actions performed by the subordinate thread include instruction and data pre-fetching to reduce cache misses [3, 4, 6, 7, 32, 8, 9, 10, 11, 12], and precomputing the outcome of hard-to-predict branches [27, 28, 29]. Moreover, it has been shown that the main thread can also benefit significantly from directly consuming subordinate thread results that are guaranteed to be correct [4, 35].

This dissertation describes and evaluates a new hardware-based architectural framework, named Symbiotic Subordinate Threading (SST), that allows otherwise idle cores in a CMP to function as helper engines for the individual threads running on the active cores. Our model exploits various sources of subordinate threading benefits: cache pre-fetching, branch pre-computation, and result reuse. The subordinate thread runs ahead of the main thread, performing cache pre-fetches and resolving branch miss-predictions ahead of the main thread demand and forwarding all of its results to the main thread. The main thread consumes the subordinate thread results that are guaranteed to be correctly executed by the subordinate thread without executing their corresponding instructions. Speeding up the main thread in this manner has several advantages. First, the overall speed of the processor increases, because it is dependent on how fast the main thread moves forward. Second, a faster main thread detects the subordinate thread's miss-speculations earlier, thereby cutting down the amount of time spent by the subordinate thread on wrong-path or wrong-data instructions. Third, both threads are making efficient use of the resources, by executing a relatively less overlapping portions of the program in parallel. Finally, because of the provision for early detection of violations, the

subordinate thread is now free to do more aggressive speculations. This symbiotic relationship between the two threads speeds up both of them, resulting in significant improvements in performance.

1.2 Contributions

This dissertation makes five major contributions, outlined below:

1. Symbiotic Subordinate Threading (SST): A key contribution of this dissertation is the development of a minimal dual-core SST model on a CMP platform that achieves significant performance benefits. The model uses simple hardware structures to facilitate forwarding of results from the subordinate thread to the main thread as well as determining if those results can be consumed by the main thread without executing their corresponding instructions. At the heart of our SST model is the formation of the subordinate thread dynamically. We provide a simple and efficient way of distilling the subordinate thread dynamically with minimum hardware requirements. Recovering the subordinate thread from the wrong path is another major concern and is addressed with minimum overhead. Our scheme is purely at the hardware level so it does not require any compiler intervention. (Chapters 3,4 and 5).

2. Understanding SST: Insight is provided regarding the sources of symbiotic subordinate threading performance. This focuses exploration of the architecture and leads to the following key results: (a) Significant performance improvement is

achieved with symbiotic subordinate threading, up to 27% improvement in speed. (b) A significant improvement in L2 cache misses is achieved in the subordinate thread. Also, a significant improvement in L1 dcache misses is achieved in the main thread. (c) The number of branch miss-predictions incurred by the main thread are reduced with SST. (d) Increased cooperation of the main thread and the subordinate thread is evident. First, the number of instructions executed by the main thread is reduced, up to 40%. Second, the subordinate thread wrong-path work is reduced significantly (Chapter 4).

4. Comparison between SST and other schemes: We perform comparisons between SST scheme and other already existing schemes that share the same high level implementation as SST. Those schemes are the slipstream processor, and dual-core execution model (DCE) [18, 36]. Both the slipstream processor and the DCE scheme are pure hardware mechanisms for speeding up single thread performance just like SST. They provide the same means as SST for forwarding results of the subordinate thread to the main thread but do not provide the means to identify the correct results of the subordinate thread as SST does. Hence, the main thread in slipstream and in DCE consumes the subordinate thread results as value and control predictions and so must validate them by executing all instructions. However, in SST the main thread consumes the correct results of the subordinate thread without executing their corresponding instructions. We show that SST, outperforms those techniques with a relatively simpler hardware additions. The average performance improvement of SST is 27% and 14% over the Slipstream processor and the DCE

scheme, respectively (Chapter 4).

3. An optimized implementation of SST: We provide another implementation of SST in which the subordinate thread is aware of its own speculation. By letting the subordinate thread know which registers and memory locations are speculative, it can avoid executing instructions that uses data-speculative input values. In that sense, the subordinate thread distills itself and only executes instructions that will yield correct results. This is especially useful in reducing the number of times the subordinate thread miss-speculates and goes on the wrong path. It also provides the benefit of reducing the total number of instructions executed by both the main thread and the subordinate thread (Chapter 5).

5. Hierarchical Symbiotic Subordinate Threading (HSST): This is another key contribution of this dissertation, extending the SST scheme to include more than one subordinate thread. Our HSST execution paradigm allows a subordinate thread to have its own subordinate thread. Collectively, the main thread and the subordinate threads form a hierarchy, with the main thread at the top of the hierarchy. As we traverse the hierarchy downwards the subordinate thread speed and speculation increase because it executes fewer instructions, and so, its ability to explore more instructions than its instruction window allows, increases. Results generated by a thread are consumed by its parent thread just like in SST with a single subordinate thread. We explored HSST with two subordinate threads, three subordinate threads and four subordinate threads. Our results yield that as we

add more subordinate threads, the penalties associated with squashing and recovering the subordinate threads increase such that they offset the benefits when we go beyond two subordinate threads. With two subordinate threads we achieved an average performance improvement of 15% over an SST scheme that uses a single subordinate thread (best of the two) (Chapter 6).

1.3 Roadmap

Background material is covered in Chapter 2. In Chapter 3, we describe how the main thread is pruned in order to be faster. This introductory Chapter provide insight into the implementation details of symbiotic subordinate threading (first and second contributions respectively). The sources of performance improvement achieved with symbiotic subordinate threading are discussed in Chapter 4 as well as comparing its performance against already existing schemes (second and third contributions). An optimized implementation of symbiotic subordinate threading is presented in Chapter 5 (fourth contribution) in which the subordinate thread is speculative aware. Hierarchical symbiotic subordinate threading is discussed in Chapter 6 (fifth contribution). Chapter 7 describes the related work. We propose the future work in Chapter 8. Chapter 9 concludes the dissertation.

Chapter 2

Background

This chapter provides the necessary background to better understand this dissertation. First we discuss single-chip multi-core processors, which is the current trend for maintaining microprocessor performance growth by providing significant benefits for both parallel and throughput oriented computing. We then discuss multithreading as a way to boost processor throughput by dividing the program workload into multiple threads that run simultaneously on the multiple cores available on the chip, thereby making efficient use of processor resources and boosting performance through exploiting thread level parallelism (TLP). Finally, we discuss subordinate threading and their benefits towards improving single-thread performance. Subordinate threading techniques utilize otherwise idle cores on a single-chip to run subordinate threads that perform some useful actions on behalf of the main thread.

2.1 Single-Chip Multi-core Processors

Execution models that can support multiple threads on a single-chip such as simultaneous multithreading (SMT), chip multiprocessing (CMP), and chip multithreading (CMT) [30, 1, 2], have received much attention from the research community in the computer architecture field. On the multiple processing elements (cores)

available in a modern processor, one can run multiple programs in parallel, or multiple threads from the same program in parallel to overlap useful computations, or subordinate threads to assist the execution of the main computation thread. In this section we discuss the technological constraints that lead to single-chip multi-core processors, mainly, the superscalar's diminishing returns and the demand for a decentralized microarchitecture, in addition to the low power budget constraint, and the demand for low inter-processor communication latency. We then discuss the existing single-chip multi-core processor architectures.

2.1.1 Motivation for Building Single-Chip Multiprocessors

Earlier in 1996, Olukotun et. al [1] showed that a better use of silicon area is a multiprocessor constructed from simpler processors and that building a complex wide issue superscalar CPU is not the best use of silicon resources. We list some of the motivating reasons for building a single-chip multi-core processor.

Diminishing Performance of the Wide-Issue Superscalar Model: The superscalar processor yields diminishing returns in performance as the issue width increases, due to the increased complexity of the issue queue and limitations in instruction level parallelism. The net effect of all the comparison logic and encoding associated with a wide instruction issue queue is that it takes a large amount of die area to implement. Moving to the circuit level, a wide instruction issue queue requires longer wires that span the length of the structure, resulting in longer delays. Farkas et. al. found that an eight-issue machine only performs 20% better than a

four-issue machine when the effect of cycle-time is included in the performance estimates [53]. This leads to the need for a microarchitecture constructed from simpler processors to maintain the performance growth of microprocessors.

Application Demand: From the applications perspective, the microarchitecture that works best depends on the amount and characteristics of parallelism present in the applications. Applications fall into two categories. The first category consists of applications with low to moderate amounts of parallelism (under 40 instructions per cycle), most of which are integer applications. The second category consists of applications with large amounts of parallelism, greater than 40 instructions per cycle. The floating point applications fall into the second category and most of the parallelism is in the form of loop-level parallelism. These two categories require different execution models. Integer applications work best on a moderately superscalar processor with very high clock rates because there is little parallelism to exploit. On the other hand, a decentralized multiprocessor paradigm best suites the floating point programs because it exploits the vast amount of parallelism present in those programs. Multi-core microarchitectures will work well on integer programs because each individual processor is a simple superscalar processor with very high clock rates. Also, multi-core microarchitectures can exploit the parallelism of the floating point applications by running multiple threads in parallel from the same program on the available cores.

Low Power Budget Requirement: Finally, power considerations also favor simpler processors but with low frequency. For workloads with adequate thread level parallelism (TLP), doubling the number of cores and halving the frequency delivers roughly equivalent performance, while reducing power consumption by a factor of four [2]. However, for applications with limited TLP, speculative parallelism or subordinate threading have to be exploited for obtaining good single-thread performance under a low-power budget; otherwise single-thread performance will be negatively affected due to low frequency.

Low Communication Latencies Requirement: In multiprogramming and conventional parallel processing environments, communication between threads is through shared memory and has latencies typically in the hundreds of CPU cycles [1]. Because of the high inter-thread communication latencies, threads are constructed such that they rarely have to communicate, and this implies that fine-grain parallelism cannot be exploited. The addition of low-latency inter-processor communication between processors on the same chip allows the multi-core processor to better exploit the available parallelism in applications.

2.1.2 Single-Chip Multi-core Architecture Models

The most common use for CMP and CMT is to execute multiple threads in parallel to increase throughput. The widespread use of visualization and multimedia applications tend to increase the number of active processes or independent threads on a desktop or a server in a particular point of time. One way to increase throughput

is to execute threads simultaneously from multiple applications. Another way is to execute multiple threads in parallel that come from a single application, such as transaction processing. Multi-core processors can also be used to accelerate the execution of a single thread of control. We next discuss the trade-offs between CMP and CMT in what they can offer regarding throughput and single-thread performance.

CMP: Each core on a CMP processor runs only a single-thread. To increase throughput, cores are made simpler and smaller to accommodate more threads. Hence, layout efficiency increases, resulting in more functional units within the same silicon area plus faster clock rates. The problem with CMP is that the hardware partitioning of on-chip processors restricts performance. The hardware partition results in smaller resources since the level-1 caches, TLBs, branch predictors, and functional units are divided among the multiple processors. Hence, single-threaded programs cannot use resources from the other processor cores and the smaller level-1 resources per core cause increased miss rates [54].

CMT: CMT processors provide support for many simultaneous hardware threads of execution in various ways, including SMT and CMP. Recall that, in an SMT processor, the physical processor core appears to the operating system as if it is a symmetric multiprocessor containing several logical processors. Hence, the physical processor core executes instructions from more than one instruction stream (thread). This increases throughput through thread-level parallelism and tolerates processor

and memory latencies to increase processor efficiency. The problem with SMT is that complexity and circuit delays grow faster with issue width. In addition, multiple threads on a single core share the same level-1 cache, TLB, and branch predictor units, which causes contention. The resulting increase in cache misses and branch miss-prediction rates limits performance. Merging CMP and SMT combines the advantages of both the individual techniques. CMT has the CMP advantages of more functional units and a faster clock than a wide-issue processor. Also, the addition of SMT increases the efficiency of the underlying CMT, because there is no hardware partitioning of processor resources, which allows a number of instructions from multiple threads to access the functional units, hence increasing the functional unit utilization.

Trade-offs: More smaller cores makes the throughput of CMPs higher than that of SMTs; however, a wide-issue SMT delivers higher single-thread performance. Given the significant area cost associated with high-performance cores, for a fixed area and power budget, the CMT design choice is between small number of high performance (high frequency, aggressive out-of-order, large issue width) cores or multiple simple (low frequency, inorder, limited issue width) cores. For workloads with sufficient TLP, the simpler core solution may deliver superior chip-wide performance at the fraction of the power. However, the simpler core solution will not work well for applications with limited TLP, unless other means for parallelization are used. In this dissertation, we realize the low area and low power budget, so we believe that future CMTs will use simpler cores. Hence, we focus in this dissertation on subordinate

threading to speed up the performance of a single thread that lacks sufficient TLP.

2.2 Multithreading

Multithreading boosts the processor throughput and improves single-program performance, through exploiting thread-level parallelism that resides in programs. It has been studied extensively in both academia and industry [63, 42, 64]. To make use of the available transistor budget, processor manufacturers such as IBM, Intel, and AMD started integrating more cores and/or threads on a single chip to support multithreading. Many studies in academia have been carried out to examine the potential of using multithreading processors such as SMT and CMP. We expect that multithreading will continue to benefit single-program performance as well as processor throughput, as long as the transistor count on a chip continues to grow. Below, we discuss some of the multithreading execution paradigms mainly multiprogramming and parallel processing. We also, discuss how each of them exploits the available thread-level parallelism in programs.

2.2.1 Multiprogramming

Multiprogramming utilizes multiprocessor systems and increases the overall processor throughput by running multiple independent programs simultaneously. Also, in a multiprogramming environment, communication or synchronization between threads is not frequent, thereby, thread-level parallelism can be easily extracted from programs. The parallelism exploited by multiprogramming is from

different programs. However, because multiprocessor systems serve a large number of threads that often share critical hardware resources, those critical hardware resources are often saturated with so many threads. This results in diminishing throughput as more threads are fed into the system. Moreover, some times, we are interested in speeding up a single program and not only achieving high throughput. However, multiprogramming often sacrifices single-program performance in order to achieve higher throughput.

2.2.2 Parallel Processing

In parallel processing, the program is divided into subprograms, which all run in parallel on a multiprocessor system. In this way, single-program performance is boosted. One way to improve the performance of a single program is We discuss two different parallel processing paradigms. The first one is conventional parallel processing and the other is thread-level data speculation technique.

In conventional multiprocessor systems, when a program is partitioned into multiple subprograms, each subprogram usually runs almost independently, thereby exploiting thread-level parallelism in a single program. In such a system, the threads are completely non-speculative and overlap useful computations, which improves the processor throughput. The partitioning is done by a compiler or a programmer such that the threads are independent. The programmer or compiler, also takes care of handling the synchronization among the different threads.

In thread-level speculation, the program is partitioned into multiple threads

speculatively. Thread-level speculation exploits thread-level parallelism by running the multiple threads in parallel. When, partitioning the threads, it is assumed that there are no memory dependences between threads. Each thread commits its results sequentially in the original program order, and this ensures correct program execution. Dependence violations are detected by a special hardware, which recovers the threads from any memory dependence violations. This hardware, also holds intermediate results until a thread commits. True dependencies between store and load operations prevent the threads from running and exploiting thread-level parallelism. In thread-level speculation a finder-grain thread synchronization is needed and is supported by the hardware, as in a chip multiprocessor.

In thread-level speculation, complicated dependence structures often limit successful exploitation of thread-level parallelism. This leads to subordinate threading, as a means of boosting single-thread performance when thread-level parallelism is scarce and partitioning a program into speculative threads is difficult due to complicated dependence structures present in the program.

2.3 Subordinate Threading

With integrating more processor cores on a single-chip multiprocessor, communication delays have been reduced considerably. In subordinate threading, one or more subordinate (helper) threads run in parallel with the main thread to help its execution. We identify two unique characteristics of subordinate threading. First, subordinate threads help speed up the execution of the main computation thread.

However, they do not affect the processor throughput. Subordinate threads help the main thread execution by running far ahead of the main thread, such that they do work on its behalf. Second, the execution of subordinate threads are decoupled from that of the main thread and their code does not have to be extracted from the original program code. Subordinate threads open up a lot of opportunities for exploiting otherwise idle cores on a chip-multiprocessor for single-program performance as we will show in this dissertation. Below, we present some of the previously proposed uses of subordinate threading to assist the execution of a single program. We then describe some of the tradeoffs of constructing effective subordinate threads.

2.3.1 Uses of Subordinate Threading

Tolerating Long-Latencies on Behalf of the Main Thread: Subordinate threads improve the performance of the main computation thread by hiding the latencies of critical instructions such as load instructions that miss in the cache or miss-predicted branch instructions. Subordinate threads help the main computation thread by executing a slice of the main computation thread. Because they execute fewer instructions than the main thread, they are able to run ahead of it and trigger long-latency events much earlier. They also overlap those latencies with useful computations. Some examples include data pre-fetching [7, 32, 33, 34, 8, 3, 4, 10, 12], instruction pre-fetching [6, 14], branch outcome pre-computation [27], and virtual function call target prediction [5]. Some subordinate threads only trigger cache-misses but they never completely service it, instead they run ahead to find other

independent cache-misses and trigger them [59, 19, 11]. For the subordinate threads to be effective they have to accomplish their task in a timely fashion. If they are too slow, the main thread will not benefit and if they are too fast, they may throw pages out of the cache that are needed by the main thread.

Executing the Exception Handler Code in Parallel with the Main Thread: Subordinate threads can also be used to run the exception handler code of faulting instructions. This relieves the main thread from executing this code, and so it can continue to execute in parallel other instructions that are independent from the one that caused the exception [65]. If the code being executed does not contain many exceptions, or if there is not enough independent instructions from the faulting instruction to overlap with the exception handler code, then performance may not improve much.

Used as an Accurate Value and Branch Predictor to the Main Thread: Subordinate threads that are distilled such that they execute hard-to-predict branch instructions and their backward slices, or critical load instructions that miss in the cache and their backward slices, produce near accurate results. Those results can serve as near perfect predictions in the main thread, thereby allowing the main thread to do progress in the event of a cache miss and reducing the number of branch miss-predictions in the main thread [36, 18, 45]. However, because the subordinate thread may execute instructions speculatively, it may introduce incorrect branch predictions that otherwise would not occur if the main thread followed the prediction

obtained from the branch predictor.

Incorporating Fault Tolerance: Subordinate threads can also be used to improve fault tolerance. They are a redundant copy of the main computation thread that runs on another core, thereby helping in detection and recovery from faults that occur during the program execution [66, 67]. This type of subordinate thread executes the same code as the main thread, and so it is totally redundant, and therefore it does not contribute to the processor performance. However, in the slipstream processor, the subordinate thread, called A-stream in slipstream terms, is used for both performance improvement as well as fault tolerance [15].

Implementing Hardware Structures and Algorithms in Software: Using subordinate threading, one can implement complicated hardware structures or algorithms in software, such as a cache pre-fetcher algorithm [46], and run them as helper threads on spare cores. In this way, the hardware complexity of the processor for supporting those new complicated structures is vastly reduced. Hence, reducing the testing and validation cost of the processor hardware. In this case, the subordinate thread code is not derived from the original program, rather it is general purpose and serves any of the individual threads running on the active cores.

2.3.2 Subordinate Thread Construction Techniques

One of the important issues in subordinate threading is generating subordinate threads that perform their required task effectively. In this dissertation we focus

on those subordinate threads that enhance single-thread performance. That means, constructing subordinate threads must take into consideration that the subordinate thread has to produce accurate results at the right time. There are several ways for constructing subordinate threads. One way, is constructing subordinate threads manually by the programmer [3]. The disadvantage of manual construction, is that it is labor intensive and is error-prone. Hence, automating the construction is more fruitful.

Kim [73] classifies the various approaches of constructing effective subordinate threads automatically based on how and when in the program's lifetime subordinate threads are constructed. There are four possible approaches to extracting subordinate threads. First, in *compiler-based extraction* the compiler analyzes the program code and generates subordinate threads at the source-level [32]. The second approach is *linker-based extraction*, which generates subordinate threads using binary analysis [8, 29]. The third approach is *dynamic optimizer-based extraction.* In this approach, binary-level code is analyzed and extracted similar to linker-based extraction. However, the extraction of binary-level code occurs at runtime using dynamic optimization techniques. Finally, the fourth approach is *hardware-based extraction* [7, 10, 18]. In this approach, subordinate threads are extracted at runtime from instruction traces. This requires, runtime analysis of retired instructions using special hardware structures.

Each of the four approaches makes use of different analysis techniques in different phases of a program's lifetime, to generate effective subordinate threads. Therefore, each approach exhibits very different characteristics. Below, we describe

some of the tradeoffs between the main two approaches to extracting subordinate threads, compiler-based extraction and hardware-based extraction; a more detailed treatment is available in [73]. We also discuss the operating system intervention with compiler-based subordinate threads versus hardware-based subordinate threads.

Run-time Versus Compile-time Information: In hardware-based extraction, the runtime information is used to accurately identify long-latency events (cache misses and hard-to-predict branches) in a program. However, the size of the hardware structure responsible for detecting dependences among instructions to help in extracting independent subordinate thread code is not sufficiently large, resulting in a limited scope of analysis. The runtime information, however, cannot be utilized in compiler-based approaches. They need to collect off-line profiles instead for identifying long-latency events. Compiler-based approaches operate in the earlier phases of the program lifetime, and so they utilize the high-level information of the program.

Dependence on the Machine Platform: When the subordinate thread is generated at runtime (hardware-based extraction), it becomes dependent on the hardware platform, i.e., the machine implementation. This is because, hardware-based extraction requires a special hardware structure for analyzing retired instruction traces. Therefore, this special hardware structure must be redesigned for every new processor design. On the other hand, compiler-based approaches generate a source code that can be compiled for any processor design. Therefore, compiler-based ex-

traction is completely independent of the platform, thereby generating code that is portable.

Effect on Transparency to the User: Runtime extraction is transparent to the user, hence, hardware-based approaches are completely transparent to the user. In hardware-based approaches, all the necessary hardware for runtime analysis and generation of subordinate threads is implemented within the processor. On the other hand, the compiler-based approach is less transparent for several reasons. First, it requires additional compilation steps such as code analysis and off-line profiling. Second, it requires the program source code, which is sometimes unavailable. Third, it requires changes to the instruction set architecture (ISA).

Effect on Hardware Complexity: In compiler-based extraction, the subordinate threads are generated using software, thereby reducing the hardware complexity. Compiler-based approaches require some hardware support though for supporting multithreading. On the other hand, hardware-based approaches generate subordinate threads using hardware, hence increasing the hardware complexity. Adding new hardware, has the disadvantage of increasing the the testing and validation cost of the hardware. However, this depends on how much special hardware is required for hardware-based extraction.

Operating System Independence: The operating system is the one that schedules the compiler-based subordinate threads to begin executing on the hardware. It may take a thread up to 50 thousand cycles to begin executing on hardware since

the time it got scheduled by the operating system due to context switching. Recall that, subordinate threads must accomplish their task at a time suitable to the main thread. If they are too slow, they will not be able to hide the latency associated with memory or branch miss-predictions. If they are too fast, they may throw out of the cache blocks that are needed by the main thread. Because subordinate threads must be timely, the operating system must schedule them at the same time as the main thread. On the other hand, pure hardware-based subordinate threads are launched independent from the operating system. They are triggered on spare idle cores by the hardware, and begin execution with no delay once they are triggered. That makes hardware-based subordinate threads more flexible and more event-driven, so they are launched only when needed.

From the above discussion, we can conclude that each approach (compiler-based and hardware-based) exhibits its own advantages and disadvantages. In this dissertation, however, we focus on the hardware-based generation of subordinate threads due to the following reasons. First of all, while compiler-based approaches have been evaluated previously in many research proposals, hardware-based construction of subordinate threads is relatively new and has not been fully investigated. Also, we believe that it is possible to support hardware-based extraction of subordinate threads with moderate additions/changes to the hardware of existing multithreading processors, as we will show in this dissertation. Third, while compiler-based extraction utilizes high-level program information of the earlier phases of the program lifetime, by supporting hardware-based extraction we are also utilizing the runtime information to construct subordinate threads for improving single-thread

performance. Fourth, with so many cores integrated on a single chip, there is more opportunity to use otherwise idle cores to improve the performance of the active cores. Finally, it is much faster to switch the mode of a core to act as a subordinate engine for another active core than to make the operating system launch a subordinate thread.

We discuss in the following chapters our proposed subordinate threading model, named, Symbiotic Subordinate Threading, which is a hardware-based approach of subordinate threading.

Chapter 3

Symbiotic Subordinate Threading (SST) — The Concepts and Implementation Details

In this chapter we describe our proposed SST scheme. The basic SST is a dual-core subordinate threading scheme in which one core is the main thread and the other core acts as the helper engine (subordinate thread) for the main thread. The high level view of SST is shown in Figure 3.1. Each thread, main or subordinate, runs on a separate core on a chip multiprocessing platform (CMP) [1]. Each core has its own data cache (dcache), instruction cache (icache), functional units (FUs), issue queue, reorder buffer (ROB), branch predictor, and register file (RF). Both threads share a unified L2 cache, which is updated only by the main thread. The subordinate thread forwards all of its outcomes to the main thread via a first-in-first-out (FIFO) queue. The mechanisms for distilling the subordinate thread and the main thread as well as the means for communicating the subordinate thread outcomes to the main thread will be described shortly, followed by the detailed design of SST.

3.1 A Simple Methodology for Distilling The Subordinate Thread

In order for the subordinate thread to run ahead of the main thread, it must speculate more often and skip instructions. There have been several techniques

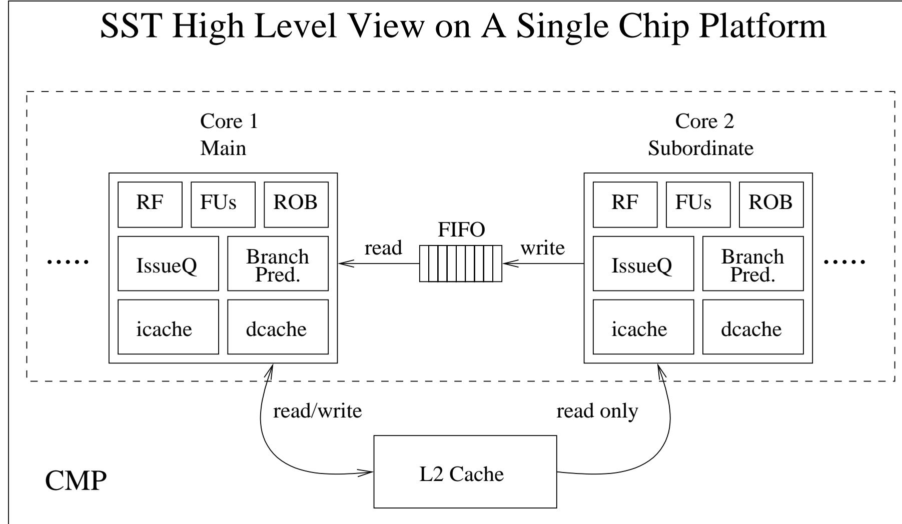

Figure 3.1: SST top level design.

for distilling the subordinate thread, and they can be divided into dynamic (at run time) and static (by the compiler). In our implementation of the subordinate thread we distill it dynamically using the hardware, to utilize the runtime information about the program and the data. Also, our hardware mechanism does not require caching recurring code regions of the subordinate thread as conventional hardware mechanisms require.

The subordinate thread we use skips highly predictable branches and their backward slices. This allows it to concentrate on the hard-to-predict branch instructions. It also identifies critical memory instructions and retires them early from the pipeline similar to runahead execution [19], so that they do not block the pipeline. The criticality of a memory instruction is determined by the number of cycles it spends at the head of the ROB, waiting for main memory.

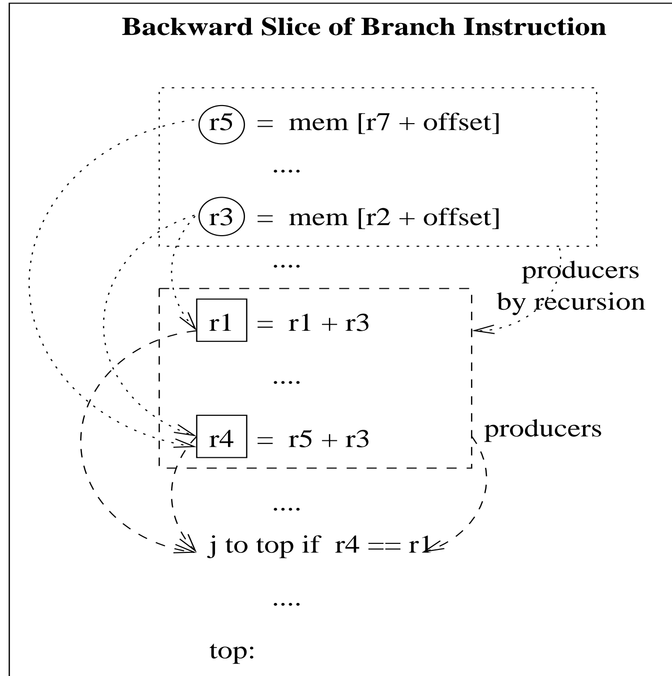

Figure 3.2: Identifying the backward slice of a branch instruction.

Identifying highly predictable branches: We use a simple methodology for identifying highly predictable (non-critical) branch instructions and distill them out of the subordinate thread along with their backward slices. We use a branch predictor to identify highly predictable branches. Branch instructions that are critical usually have low prediction confidence in the branch predictor, and the highly predictable ones usually have high prediction confidence in the branch predictor. When the subordinate thread skips a highly predictable branch instruction, it follows the predicted outcome and direction of the highly predictable branch at the fetch stage and marks all pipeline instructions in front of the highly predictable branch instruction if they belong to its backward slice. Figure 3.2 shows a branch and its backward slice. The algorithm for identifying the backward slice of a branch in-

struction begins by identifying the producer instructions of each input operand of the branch instruction (those producer instructions are shown with a square at their output operand in Figure 3.2). Next, by recursion, the algorithm is applied on the producer instructions to identify the instructions that produce their input operands (shown with a circle at their output operand in Figure 3.2), and so on.

Identifying long latency memory instructions: Our subordinate thread also does not wait for long latency memory instructions to complete. When a memory instruction reaches the head of the ROB, a counter is reset and is incremented every cycle the memory instruction spends at the head of the ROB. When that counter reaches a specific maximum value, the subordinate thread concludes that this memory instruction is critical (long latency). It marks all subsequent instructions in the pipeline that are dependent on its outcome. It then supplies a speculative value (most likely an invalid value) for the output operand of the memory instruction and retires it.

Handling Instructions Marked to Be Skipped: The backward slices of highly predictable branches, as well as long latency memory instructions and their dependency chains free all the resources they hold. Hence they do not finish executing and pass into the pipeline as no-ops. Once they reach the head of the ROB, their ROB entry is reclaimed and their decoded information is written onto the FIFO queue. Because they pass into the pipeline as no-ops, they leave the pipeline much earlier, and so, more instructions can be brought into the pipeline, resulting in a

wider instruction window for the subordinate thread.

Note that the subordinate thread passes every instruction onto the FIFO queue, even if it did not execute it. In case of skipped branches, it also passes the predicted branch outcome that it followed. This is essential information that is passed to the main thread to help it monitor the subordinate thread path, as will be discussed later.

3.2 A Simple and Efficient Way of Pruning The Main Thread

In this section we introduce an algorithm that helps the main thread consume the results that were correctly produced by the subordinate thread without having to execute their corresponding instructions. This involves recording the subordinate thread speculative state (registers or memory addresses that contain speculative values in the subordinate thread), to aid in identifying outcomes of the subordinate thread that were computed using speculative input values from those that did not involve any speculative input values. We will show that our technique for pruning the main thread is independent from the subordinate thread type. A working example is presented at the end of this section for clarification.

3.2.1 Basic Idea

In order for the subordinate thread to run ahead of the main thread, it skips instructions. The output registers of those skipped instructions contain data-speculative values. Some of the instructions the subordinate thread executes are

dependent on the ones it skipped. The outcomes of those dependent instructions are speculative in nature (which could be correct or incorrect). We categorize the instructions executed by the subordinate thread into two classes: those producing *data-speculative* outcomes and those producing *non-data-speculative* outcomes.

Data-speculative outcomes are those that are obtained when the subordinate thread uses at least one input register that is data-speculative. An input register is data-speculative in the subordinate thread if it is produced by an instruction that was skipped by the subordinate thread, or if it was produced by a data-speculative instruction in the subordinate thread. Data-speculative outcomes could be incorrect and so the main thread does not consume them.

Non-data-speculative outcomes are those that are obtained when the subordinate thread uses non-data-speculative input registers. In other words, the values of their input registers match those of the main thread, and the outcomes will match those produced by the main thread. Therefore, they are correct outcomes and the main thread can consume them without executing their corresponding instructions.

3.2.2 Skipping Non-Memory Instructions

We propose an algorithm for keeping track of the architected registers of the subordinate thread that contain data-speculative values and those that contain non-data-speculative values. We introduce a bitmap called the *Register Speculation Bitmap (RSB)* with as many bits as the number of architected registers. This bitmap specifies whether each architected register in the subordinate thread contains a data-

speculative value or not. It is kept and updated by the main thread during dispatch and writeback. A '1' in a bit position indicates that the corresponding register has a data-speculative value. Initially, all registers in the subordinate thread contain non-data-speculative values, and so all the bits of the RSB are initialized to zeroes (non-data-speculative). Also, every time the subordinate thread re-starts, after a control or data miss-prediction, it is given a fresh copy of the register file that contains no data-speculative values, and so all the bits of the RSB are reset (cleared).

Scenarios for Updating the RSB: There are three scenarios for updating the bits of the RSB by the main thread. First, when the subordinate thread skips an instruction, the bit corresponding to its output register is marked by the main thread at dispatch stage as data-speculative in the RSB. When the subordinate thread executes an instruction, if any of its input registers has been marked in the RSB, then also the output register is marked by the main thread at dispatch stage as data-speculative in the RSB. Finally, if none of the input registers of the instruction have been marked in the RSB and the subordinate thread has not skipped the instruction, then the bit corresponding to the output register of the instruction is reset to '0' (non-data-speculative) by the main thread at dispatch stage. Only in this final case, the main thread can consume the result of the instruction from the subordinate thread without re-executing it. In the other two cases, the main thread must execute the instruction, and validate its outcome against the result it obtained from the subordinate thread. If they match, the main thread unmarks the bit corresponding to the output register of the instruction in the RSB, otherwise

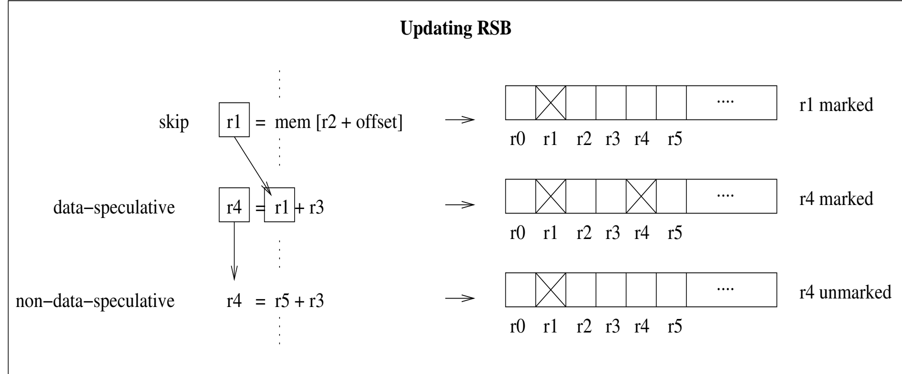

Figure 3.3: RSB update scenarios.

it remains marked. This validation and unmarking of the RSB bits is done by the main thread at writeback.

Figure 3.3 shows a code snippet in which the first instruction is skipped by the subordinate thread. Its output register (r1) is marked in the RSB as data-speculative. Later on, another instruction that uses r1 as input operand is also data-speculative, and so its output register (r4) is marked as data-speculative in the RSB. The last instruction shown in the code snippet is not data-speculative because both its input operands (registers r3 and r5) are not marked in the RSB and therefore they are non-data-speculative. Therefore, the output operand (r4) is not data-speculative and so is unmarked in the RSB.

3.2.3 Skipping Memory Accesses (Only LOAD Instructions)

We propose a similar mechanism to the one mentioned in the previous subsection for keeping track of the subordinate thread memory addresses whose memory values are data-speculative and those whose memory values are non-data-speculative.

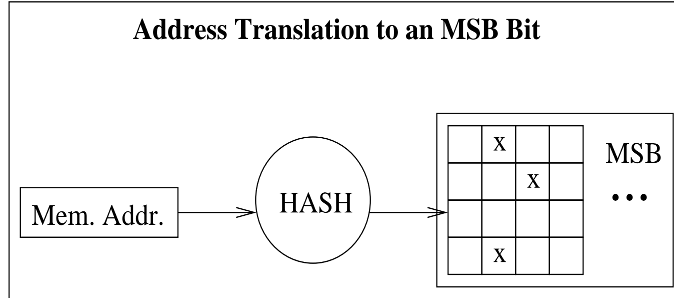

Figure 3.4: MSB addressing.

We introduce another bitmap, the *Memory Speculation Bitmap (MSB)*, similar to the RSB. A perfect MSB would contain as many bits as the number of unique memory addresses. That would be a huge bitmap however, and so we compromise that with a much smaller bitmap that is indexed by hashing the memory address as shown in Figure 3.4. It is kept and updated by the main thread in the dispatch stage. Initially, the subordinate thread begins with a memory state that is non-data-speculative, and so all the bits of the MSB are initialized to zeroes (non-data-speculative). All of the bits of the MSB are also reset (cleared) every time the subordinate thread re-starts (from control or data miss-prediction), because its dcache is all invalidated upon re-starting.

A bit in the MSB is marked by the main thread as data-speculative if the memory address of a store instruction maps to that bit. We consider all store instructions to be data-speculative in the subordinate thread, because the subordinate thread is not allowed to update the memory hierarchy except its L1 dcache, and so all writes it does to its L1 dcache are lost when blocks are thrown out of it. As a result, a load that follows a store in the subordinate thread may read from the

same store address before the main thread makes the update and therefore may read a stale value. Hence, any load in the subordinate thread with the same address as a previous store is considered to be reading a data-speculative value. The main thread must therefore execute all store instructions. It also must execute load instructions whose memory addresses map to a marked bit in the MSB. However, it may consume results produced by the subordinate thread for load instructions whose MSB bits corresponding to their hashed memory addresses are not marked as data-speculative.

The main thread does not unmark bits in the MSB (except when a subordinate thread recovers from a data or control miss-prediction, in which case all the bits of the MSB are unmarked), because more than one address may map to the same bit in the MSB; so if one of them is speculative, the corresponding bit is marked. Future references to the same bit by different addresses will result in the main thread executing those instructions, as it cannot determine which of these addresses had marked the bit.

3.2.4 An Example

Consider the loop example from benchmark *perl* shown in Figure 3.5a. The subordinate thread and the main thread are spawned at the same time and they both begin execution from instruction 1. The subordinate thread skips highly predictable branches and their backward slices.

After a few iterations of the loop, the jump instruction (instruction 7) settles

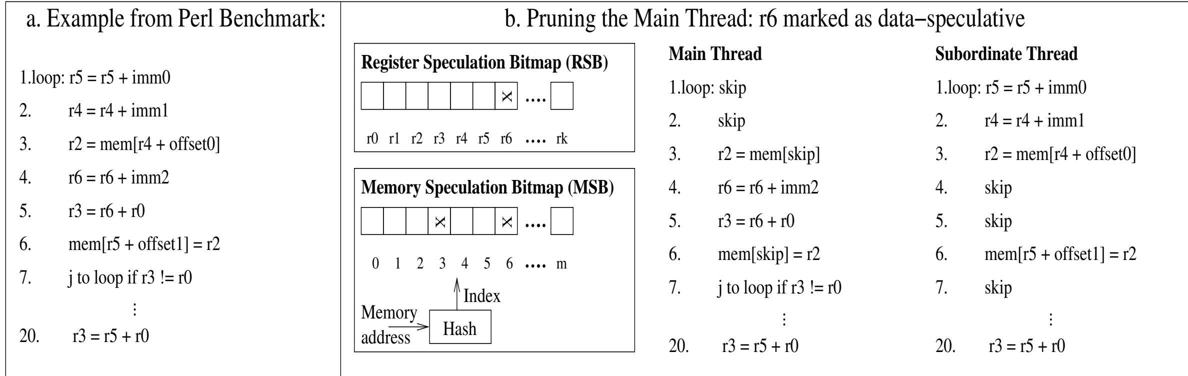

Figure 3.5: (a) Loop example from benchmark *perl*; (b) Example of reducing the number of executed instructions by the main thread.

its prediction to be taken and is determined to be highly predictable by the branch predictor. At this point the subordinate thread begins to skip the jump instruction along with its backward slice (instructions 4 and 5). This makes the subordinate thread run faster than the main thread. Registers r3 and r6 are marked in the RSB by the main thread as data-speculative, because their producer instructions were skipped by the subordinate thread. However, after the loop code, register r3 is updated by instruction 20 which uses non-data-speculative input registers (r5 and r0). As a result, its outcome is non-data-speculative and register r3 is unmarked in the RSB. Only during the loop iterations, r3 is marked in the RSB as data-speculative.

Registers r3 and r6 do not serve as input registers to any subsequent instructions executed by the subordinate thread. Therefore, the subordinate thread's outcomes of instructions 1 and 2 are non-data-speculative, and the main thread may consume them without executing their corresponding instructions. This is shown in

Figure 3.5b; note that the main thread skips instruction 1 and 2.

Note that the main thread does not skip store instructions because it has to maintain a correct L2 cache. The subordinate thread's dcache may become corrupted, because the subordinate thread may skip store instructions, and dirty blocks in its dcache can be displaced by old blocks from lower levels of memory that were not yet updated by the main thread. To ensure a correct L2 cache, all memory stores are executed by the main thread; however, the main thread can skip the address generation part. So, the memory address calculation part of instructions 3 and 6 is skipped by the main thread. When the main thread dispatches instruction 3 to the dynamic scheduler, it marks register r2 as data-speculative in the RSB. The main thread later unmarks it in the RSB if the value it read from memory matches the value it obtained from the subordinate thread. In this example, it is shown as unmarked. The memory access part of instruction 3 can be skipped as well by the main thread if the MSB bit at the hashed index of its address is not marked (it is not skipped in this example). Instruction 6 is a store instruction so it will be executed by the main thread and will mark the corresponding bit in the MSB.

A miss-speculation in the subordinate thread will occur in the last iteration of the loop of Figure 3.5. The subordinate thread will follow the same branch direction (taken). The main thread will follow the same direction as the subordinate thread and will execute the branch instruction, which will result in a branch miss-prediction. Before the main thread fully resolves that branch, it might have fetched and decoded from the wrong path. Any marking or unmarking in the RSB or MSB does not matter, as both bitmaps will be cleared when the subordinate thread is

re-started.

Pruning the Main Thread is not Affected by the Subordinate Thread Type: We presented a simple methodology with very little hardware added to help in pruning the main thread. We like to point out here that this is independent of the type of subordinate thread running. The subordinate thread maybe very speculative or moderately speculative. It can be formed dynamically or formed statically. If the subordinate thread does not use any form of speculation then the bitmaps are not needed, as all outcomes of the subordinate thread will be non-data-speculative; otherwise they are needed.

3.3 Communicating Subordinate Thread Results and Decoded Information to the Main Thread

In SST the subordinate thread forwards its results to the main thread as part of the increased cooperation between the main thread and the subordinate thread. Also, in SST the subordinate thread fetches all instructions and then decodes them. We let the subordinate thread forward the decoded information of all instructions it fetches to the main thread, even if it did not execute them. This saves the main thread from having to fetch and decode again what was already fetched and decoded by the subordinate thread. We used a *first-in-first-out (FIFO) queue* as the communication means between the subordinate thread and the main thread. The subordinate thread writes each instruction onto the FIFO queue when it commits

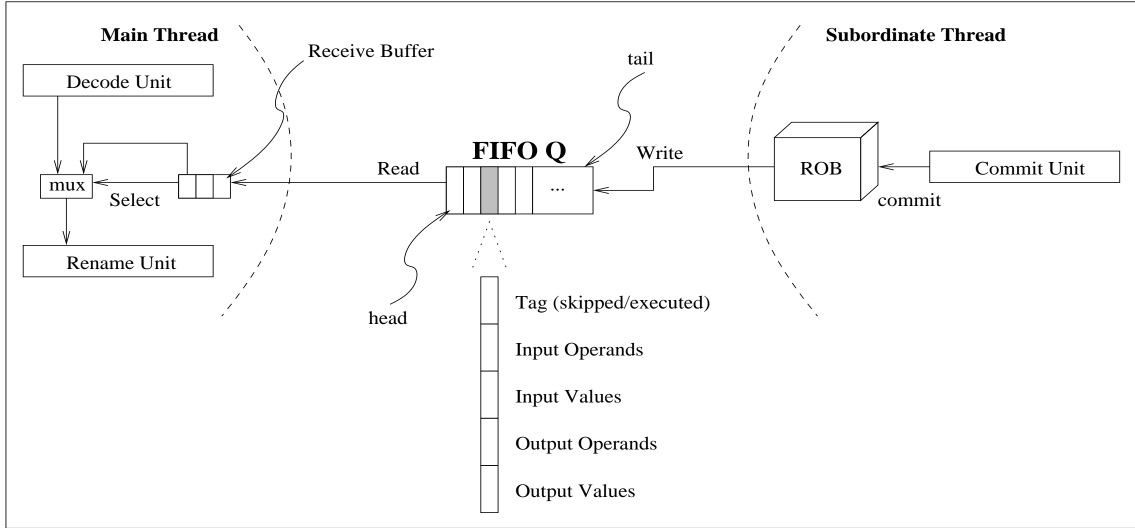

Figure 3.6: FIFO queue.

it. The main thread reads the entries of the FIFO queue during dispatch stage.

Operation of the FIFO Queue: As shown in Figure 3.6, the FIFO queue connects the main thread and the subordinate thread. When the subordinate thread commits an instruction, it writes its results (if it executed it) and its decoded information on one end of the FIFO queue, the tail. The main thread reads the entries of the FIFO queue from the other end, the head, and places the entries onto a receive buffer. Each entry in the FIFO queue contains a tag that indicates whether the subordinate thread executed the instruction or not. Also, each entry in the FIFO queue contains fields to store the decoded information (opcode, input and output operands) of the instruction as well as their values.

Main Thread Benefits from the FIFO Queue: The FIFO queue represents the medium in which all of the subordinate thread's work is stored for the main

thread consumption. Because of the simplicity of the FIFO queue, the instructions are placed by the subordinate thread in order and the main thread reads them in order, requiring no extra work for checking the order of the instructions, hence the main thread may not loose synchronization with the subordinate thread. Also, the main thread can read at its own pace (which is usually slower than the subordinate thread) from the FIFO queue. The main thread also uses the FIFO queue instead of its icache, for reading from it the decoded information of every instruction placed by the subordinate thread, and this saves a lot of fetch and decode cycles in the main thread. As shown in Figure 3.6, there is a multiplexer with a select signal that selects between the decoded instructions coming from the decode unit of the main thread versus the decoded instructions coming from the receive buffer. If the receive buffer is empty, then the select line is set to 0, and the decoded instructions coming from the decode unit pass through the multiplexer; otherwise the ones coming from the FIFO queue pass through the multiplexer. In the case when there is no running subordinate thread, then the receive buffer will be always empty and the decoded information will always be coming from the decode unit. The main thread may also benefit from the subordinate thread results that are non-data-speculative because it can read them from the FIFO queue, and consume them without having to execute their corresponding instructions. Finally, it requires no sophisticated comparisons for the main thread to integrate the subordinate thread results from the FIFO queue into its state (register file and memory).

Subordinate Thread Benefits from FIFO Queue: The subordinate thread also benefits from placing its results and decoded information onto the FIFO queue. In this way the main thread can monitor its control path, and can detect when it goes on a wrong path. Also, the FIFO queue increases the effective size of the subordinate thread instruction window, by allowing it to place all its outcomes on the FIFO queue once they are committed. This frees the subordinate thread ROB entries faster, allowing more in-flight instructions to be fetched into its ROB.

Drawbacks of Using a FIFO Queue: The FIFO queue is extra hardware that is placed outside the cores. Hence, it adds more complexity and communication latency between the threads. It takes several cycles for the subordinate thread to place its outcomes on the FIFO queue, and it takes additional cycles for the main thread to read those outcomes.

Alternatives to Using A FIFO Queue: There are alternatives to using a FIFO queue for communication between the main thread and the subordinate thread. One such alternative is using *shared memory*. In this scheme, the subordinate thread writes its results to shared memory and the main thread reads from shared memory. This requires the programmer to program the communication between the main thread and the subordinate thread via shared memory, which does not apply to our model because the SST subordinate thread is spawned and formed dynamically. Another alternative is to use *message-passing* via the on-chip interconnection network. It requires sending and receiving messages between the cores, which is not practical

because in the case of SST, there will be a continuous flow of messages going from the subordinate thread to the main thread and that may overload the interconnect. Also, if one message arrives before another, that may cause loss of synchronization between the main thread and the subordinate thread. For those reasons, we find the use of a FIFO queue to be more attractive, especially that it is a dedicated hardware buffer just to serve the main thread and the subordinate thread.

3.4 Putting it All Together: The SST Microarchitecture

We next present a hardware implementation of the SST scheme that we have proposed. It uses two cores present in a chip-multiprocessing (CMP) platform [1]. In addition to multiple sequencers, a CMP processor has multiple pipelines for processing multiple threads in parallel. Figure 3.7 shows a 2-core CMP enhanced to support our SST scheme. Each thread runs on a separate core containing a register file, an issue queue, a branch predictor, an ROB, an L1 dcache, and an icache. A second level cache (L2 cache) is shared among both threads, and can be updated only by the main thread. We added extra hardware for pruning both the main thread and the subordinate thread. The extra hardware included is the Register Speculation Bitmap (RSB) for the purpose of identifying non-data-speculative register values of the subordinate thread. Also, the Memory Speculation Bitmap (MSB) is included to help identify memory addresses that contain non-data-speculative values in the subordinate thread. The FIFO queue is included for communicating the subordinate thread results and decoded instructions to the main thread.

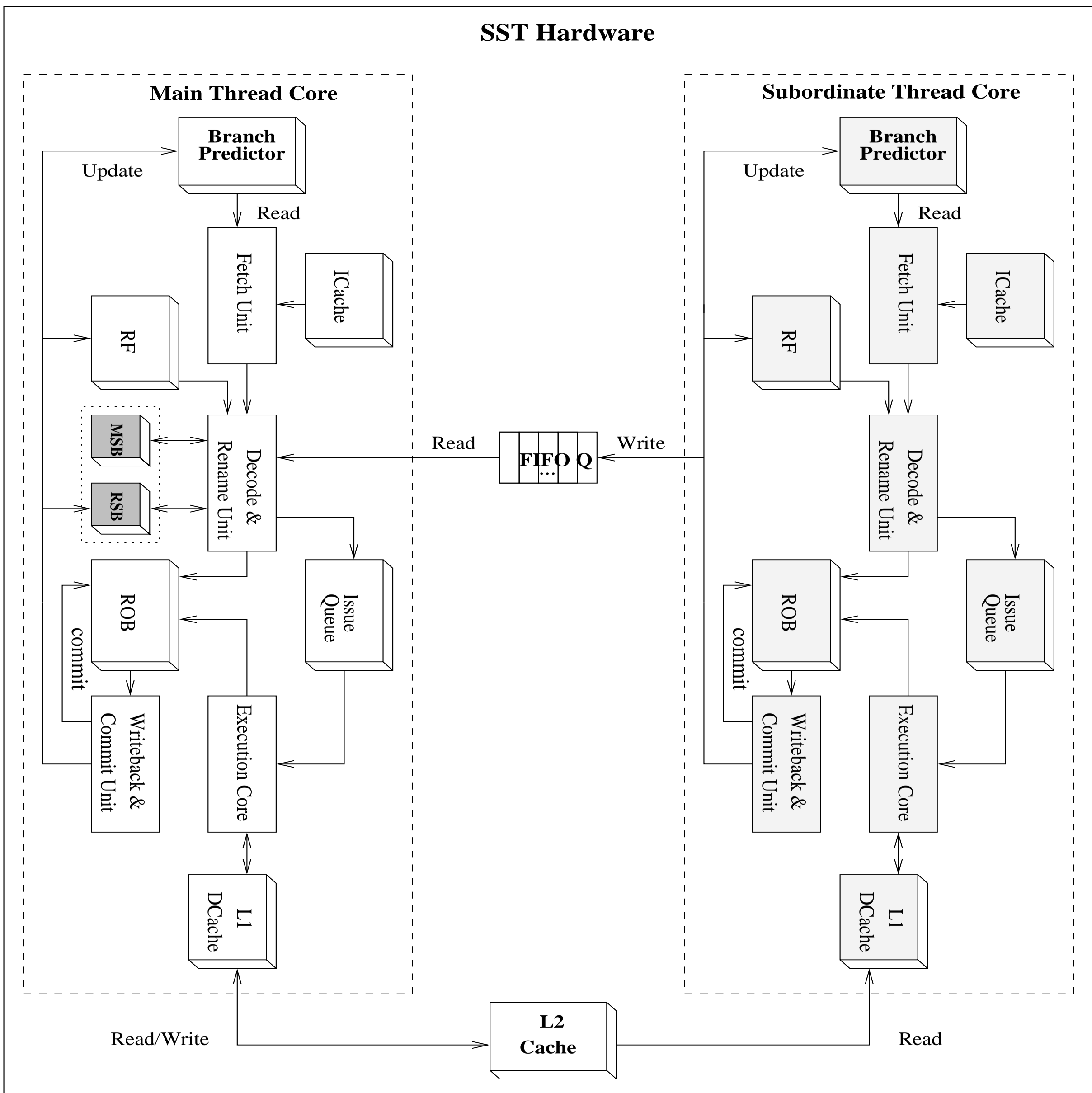

Figure 3.7: SST microarchitecture.

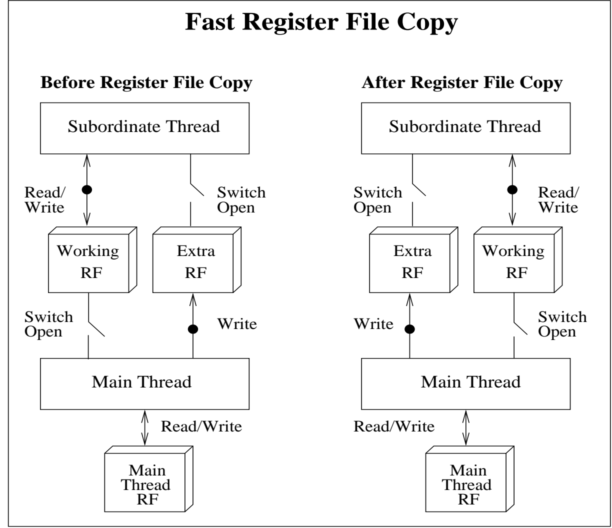

Figure 3.8: Fast recovery of the subordinate thread state.

3.4.1 Basic Operation

The subordinate thread starts with a full copy of the program, and is then distilled. It is spawned when the main thread is spawned and continues to run as long as the main thread runs. In our implementation of SST, the subordinate thread is the leader and the main thread follows. It is possible that the main thread may go ahead of the subordinate thread, such as when the subordinate thread recovers from miss-speculation. However, in our implementation of SST, we do not let the main thread advance with executing instructions if the FIFO queue is empty. This ensures that it never goes ahead of the subordinate thread, and hence, does not loose synchronization with the subordinate thread. The main thread restarts the subordinate thread when it goes on a wrong path and when a system call is encountered. In the case of a system call, the subordinate thread is restarted in kernel mode and continues to execute as a helper thread to the main thread. It does

not execute the I/O instructions; these are executed by the main thread. The RSB and MSB are read and updated by the main thread in the dispatch stage. The RSB can also be updated by the main thread in the writeback stage. The subordinate thread writes its outcomes into the FIFO queue in the commit stage and the main thread reads them in the decode (dispatch) stage.

Instructions Executed by the Main Thread: In our SST scheme, we try to eliminate the redundancy between the main thread and the subordinate thread. We achieve that by letting the main thread skip instructions that were correctly executed by the subordinate thread. The main thread has to execute though, all instructions identified as data-speculative in the subordinate thread as well as the ones skipped by the subordinate thread. The main thread must execute all store instructions, as well.

3.4.2 Memory System

The main thread and the subordinate thread share an L2 cache. The main thread has to maintain a correct memory system, and so it is allowed to read and write the L2 cache. A subordinate thread's dcache can be corrupt because it is speculative, and so it is not allowed to write to the shared L2 cache. Therefore, all memory writes to the L2 cache are done by the main thread even if the subordinate thread performed them correctly on its dcache.

3.4.3 Recovery of the Subordinate Thread from Miss-speculation

Because the subordinate thread is speculative, it often goes on a wrong path, as well as corrupts its state (register file and L1 dcache). The work done by the subordinate thread on wrong paths is useless. Also, if its state is mostly corrupt while it is on the correct path, then it will be doing useless work as well, as most of its input operands will have speculative (invalid) values. When the subordinate thread miss-speculates, it is better to re-start it with a fresh clean copy of the main thread correct state. This requires squashing the subordinate thread, and copying the program counter and correct register values from the main thread. The subordinate thread also invalidates all of its L1 dcache lines upon recovery. The main thread also clears the MSB and RSB bitmaps upon recovery of the subordinate thread.

Full versus Partial Recovery: While the subordinate thread is copying the main thread register file, the main thread cannot advance forward. We minimize the delays due to register file copy, by letting the subordinate thread copy the main thread register file only if most of its registers are corrupted (*full recovery*). Otherwise the subordinate thread does not have to copy the main thread register file (*partial recovery*). In partial recovery, the subordinate thread only recovers its correct path and invalidates the entries in its dcache. Partial recovery allows both the main thread and the subordinate thread to start executing much faster than in full recovery. Partial recovery can be applied when the subordinate thread went on a wrong path and only slightly corrupted its register file while on the correct path.

Fast Register File Copy: One simple way to reduce delays due to register copying is to include an extra register file (in addition to the register files kept by each thread). This is shown in Figure 3.8. The main thread is responsible for updating its own register file as well as the extra register file. When a subordinate thread is about to start (immediately after its spawning or after a miss-speculation recovery), it switches to the extra register file, which has the correct state. The extra register file now becomes the working register file of the subordinate thread. The register file used previously by the subordinate thread becomes the extra register file that will be updated by the main thread in the future. The use of the extra register file is similar to the use of *shadow registers* presented in [25] for doing compiler-based speculation (boosting).

Penalty for Re-starting the Subordinate Thread: When re-starting the subordinate thread, it takes a while for the first instruction result to be produced and buffered by the subordinate thread for the main thread consumption. That time is equal to at least the depth of the pipeline. During that time, the main thread must wait until the subordinate thread begins to produce results and write them onto the FIFO queue.

Chapter 4

Experimental Results of SST

In this chapter we present experimental results highlighting the performance gains obtained due to increased cooperation between the main thread and the subordinate thread in our SST scheme. In order to show the benefits of our scheme, we compare its performance with an already existing subordinate threading scheme, the slipstream processor [18], which does not let the main thread skip instructions. We also compare its performance against a second subordinate threading scheme, the Dual-Core Execution scheme (DCE) [36], which employs a similar subordinate thread to the one we use in SST, and does not let the main thread skip instructions. Our SST scheme achieves higher performance than the slipstream processor, with a much simpler hardware. It also achieves much higher performance than the DCE scheme with moderate additions of hardware, mainly the MSB and the RSB.

We developed our own cycle-accurate subordinate threading simulator from the SimpleScalar toolset [26]. Our simulator faithfully models an SST system running on a multi-core CMP, with a main thread and a subordinate thread, and their interconnections, as per the block diagram of Figure 3.7 in the previous chapter. The microarchitectural parameters we used are given in Table 4.1. The L1 dcache of a subordinate thread is invalidated on its recovery from the wrong paths. We used a single branch predictor for all cores, and the predictor is updated only by

the main thread.

Single Core Parameters	
L1 ICache	sz/assoc/repl/ln/lat=16KB/1way/LRU/64B/1cycle
L1 DCache	sz/assoc/repl/ln/lat=64KB/4way/LRU/64B/1cycle
L2 Cache (data+instrs.)	sz/assoc/repl/ln/lat=1024KB/8way/LRU/128B/6cycles
Main Memory Latency	50 cycles
Fetch/issue/retire	Bandwidth = 4/4/4
ROB/LdStQ/FetchQ	size = 32/16/8 entries
Branch Predictor	type = bimodal, size = 32K entries
Branch Penalty	3 cycles
SST-Specific Parameters	
MSB	64 bits
FIFO Queue	latency/bandwidth/size = 2 cycles/4 instrs./32 instrs.
Branch Threshold	Low speculation = 60, High speculation = 1
Sub. Thread Recovery	7+ cycles
Slipstream-Specific Parameters	
Sub. Thread Distill Unit	512 entries

Table 4.1: Microarchitectural Simulation Parameters For Smaller Cores

We used the SPEC_INT2000 benchmarks for this study. We used the SimPoint toolset [38, 39, 40] to identify representative simulation points. Each benchmark is simulated for 500 million instructions after fast-forwarding the number of instructions determined by SimPoint, which is around 1 billion for most benchmarks.

4.1 Performance Evaluation of SST Against Slipstream Processor

In this experiment, we evaluated 4 different configurations of our SST scheme to show the benefits of *symbiosis* — increased cooperation between the main thread and the subordinate thread — by allowing the main thread to consume the non-data-speculative results of the subordinate thread without executing their corresponding instructions.

In the first configuration of SST, the subordinate thread is not highly speculative and the main thread consumes results of the subordinate thread that do not involve any memory access (i.e., it does not consume any results of load instructions). The subordinate thread in the second configuration of SST is not highly speculative as well, but the main thread may consume all of the subordinate thread results including the ones corresponding to load instructions. In the third and fourth configurations of SST, the subordinate thread is highly speculative. The main thread in the third configuration consumes all results that are correctly produced by the subordinate thread except those of the load instructions, and in the fourth configuration, it consumes the results of the load instructions as well. The branch and memory thresholds shown in Table 4.1 indicate the level of speculation of the subordinate thread. For a not-too-speculative subordinate thread, a branch instruction is considered highly predictable and can be removed from the pipeline if its confidence counter reached 60, and for a highly speculative subordinate thread the branch instruction can be removed from the pipeline if its confidence counter reached 1.

4.1.1 Average IPC Improvement of SST

We first evaluated the IPC performance gains of SST against the slipstream processor [15, 18]. In slipstream processors, the A-stream (subordinate thread) runs a shorter program based on the removal of ineffectual instructions (highly predictable branches and their backward slices) while the R-stream (main thread) uses the A-stream results as predictions to make faster progress. Hence the R-stream (main thread) executes every instruction in order to validate its outcome against the outcome obtained from the A-stream (subordinate thread). In this experiment we distill the subordinate thread of SST just like in slipstream for a fair comparison. So, we use a table that stores saturating counters for highly predictable branches and their backward slices just as in slipstream. This table is updated by the main thread. For every fetched instruction, the subordinate thread checks its corresponding saturating counter to decide whether to skip or execute that instruction. The size of this table is 1024 entries. We will later show that if the SST subordinate thread is distilled in the manner we discussed in Section 3.1, then we can achieve much higher performance than the A-stream in slipstream (which uses the huge table).

Figure 4.1 presents the results obtained for the four configurations of SST against that of a slipstream processor. Each bar represents the average IPC performance improvement obtained from skipping instructions in the main thread for the four configurations of SST, versus the corresponding base slipstream scheme (the main thread does not skip instructions)[1].

[1] The base slipstream processor configurations we used have an average speedup of 7% and 14%

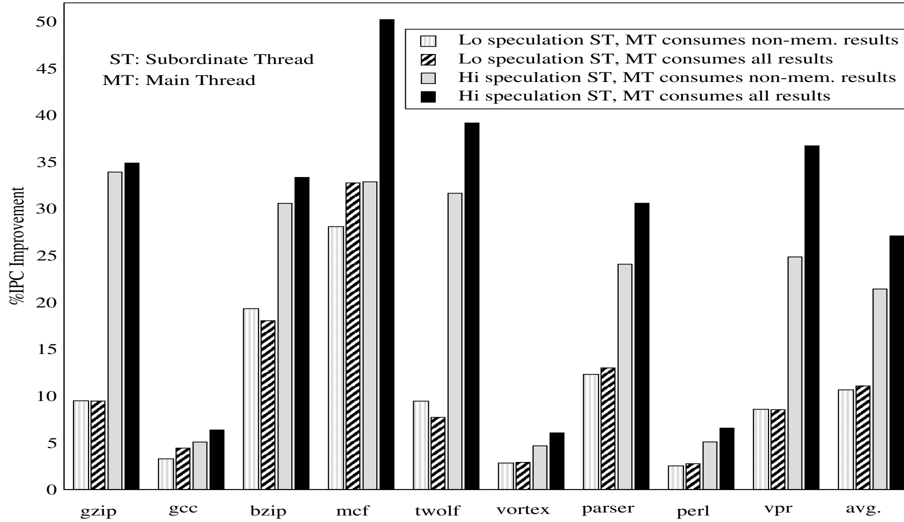

Figure 4.1: % IPC improvement achieved with symbiotic subordinate threading (SST) over the slipstream processor (main thread does not skip instructions). (a) SST with low speculation subordinate thread, and main thread does not skip load instructions; (b) SST with low speculation subordinate thread, and main thread skips load instructions; (c) SST with high speculation subordinate thread, and main thread does not skip load instructions; (d) SST with high speculation subordinate thread, and main thread skips load instructions

It is clear from Figure 4.1 that our SST performs well for all the benchmarks. The average performance improvement is 10%, 11%, 21%, and 27% for the four schemes. All benchmarks except *gcc*, *vortex*, and *perl* perform very well, especially with a highly speculative subordinate thread (bar 3 and bar 4). The lackluster performance for these 3 benchmarks is due to the subordinate thread incurring a

over a single-threaded (superscalar) processor for a configuration with moderate skipping in the subordinate thread (low speculation) and a configuration of aggressive skipping in the subordinate thread (high speculation), respectively.

large number of instruction cache misses.

In the bars of the average performance, there is a jump in performance from the first 2 bars to the last 2 bars. This indicates that with a subordinate thread that has fewer restrictions to advance, our SST scheme performs even better. We also noticed from the average performance bars that when memory accesses are skipped by the main thread the performance improves, especially when the subordinate thread aggressively skips instructions. When the subordinate thread does not do aggressive skipping, the performance does not improve much even when the main thread skips memory accesses. The performance numbers presented in Figure 4.1 are further analyzed in the following subsections, using additional statistics.

4.1.2 Instruction Distribution in The Main Thread

The first logical result to be drawn from the previous subsection is that the main thread runs faster, hence contributing to the overall performance improvement. It is running faster because it is skipping instructions whose outcomes were correctly produced by the subordinate thread. Figure 4.2 shows the distribution of the total instructions in the main thread for two of the SST schemes whose performance was shown in Figure 4.1 (first and third bars). Each bar in Figure 4.2 shows the division of skipped instructions in the main thread. The main thread performs all memory accesses but may skip the address computation part of memory instructions. A significant portion of the total instructions is skipped by the main thread in both schemes; hence the main thread runs faster. The first bar of Figure 4.2 corresponds

to an SST configuration with a high-speculation subordinate thread, and the second bar corresponds to an SST configuration with a low speculation subordinate thread. In the first scheme (first bar), fewer instructions are skipped by the main thread, because the subordinate thread skipped more aggressively. On the average, in the first case, approximately 45% of the instructions were executed by the subordinate thread and 55% of the instructions were executed by the main thread. In the second case, this division is approximately 60%-40%. The distribution of instructions among the threads and the performance (Figure 4.1) have a strong correlation. For example, for the first scheme in Figure 4.2, the division between the main thread and the subordinate thread for benchmark *parser* is roughly 50%-50%. In Figure 4.1 (third bar), the performance improvement for this scheme is 24%. For the second scheme, this division is roughly 30%-70%. Its performance improvement is about 13% (first bar of Figure 4.1). This tells us that a more equal distribution of instructions between the subordinate thread and the main thread for benchmark *parser* produces a more equal distribution of work among the threads and hence a higher performance. This is not the case for all benchmarks, however; some of them (such as *gcc*) will have a more equal distribution of work if the subordinate thread skips aggressively. On the average, by letting the main thread skip instructions, a more balanced distribution of work among the threads occurred, which resulted in better performance as shown in Figure 4.1.

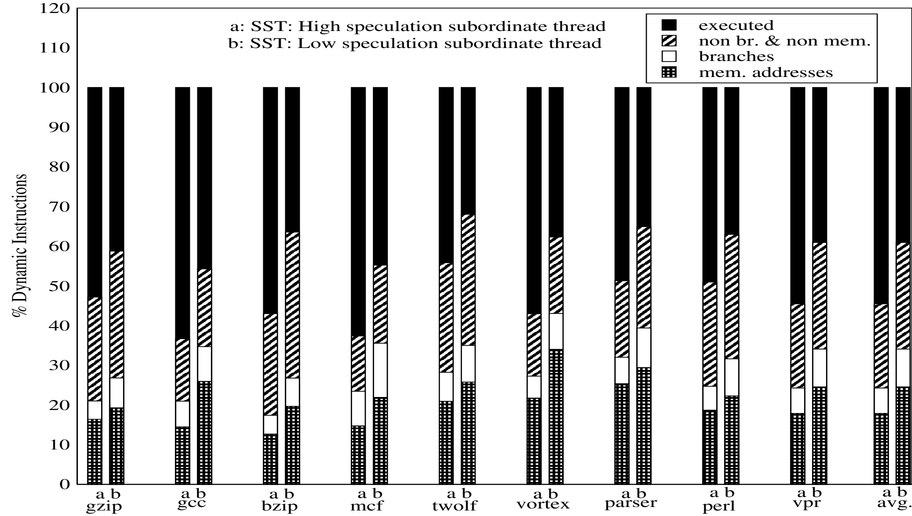

Figure 4.2: Instruction distribution in main thread for two schemes: (a) SST with high speculative subordinate thread; (b) SST with not too speculative subordinate thread

4.1.3 Less Work Done by the Subordinate Thread on Wrong Paths

We had argued earlier that if the main thread skips some instructions, it can detect subordinate thread miss-predictions earlier, thereby cutting down the time spent by the subordinate thread on wrong-path instructions. In Figure 4.3 we show a comparison of the distribution of work done by the subordinate thread for four schemes (4 bars per benchmark). Each bar shows the work done by the subordinate thread which is divided into three parts: percentage of instructions skipped, percentage of instructions executed on the correct path, and percentage of instructions executed on the wrong path. In the first two schemes (corresponding to the first two bars, respectively), the subordinate thread is highly speculative, and so it skips instructions aggressively. The main thread is not allowed to do any skipping in the first scheme (slipstream), and in the second scheme the main

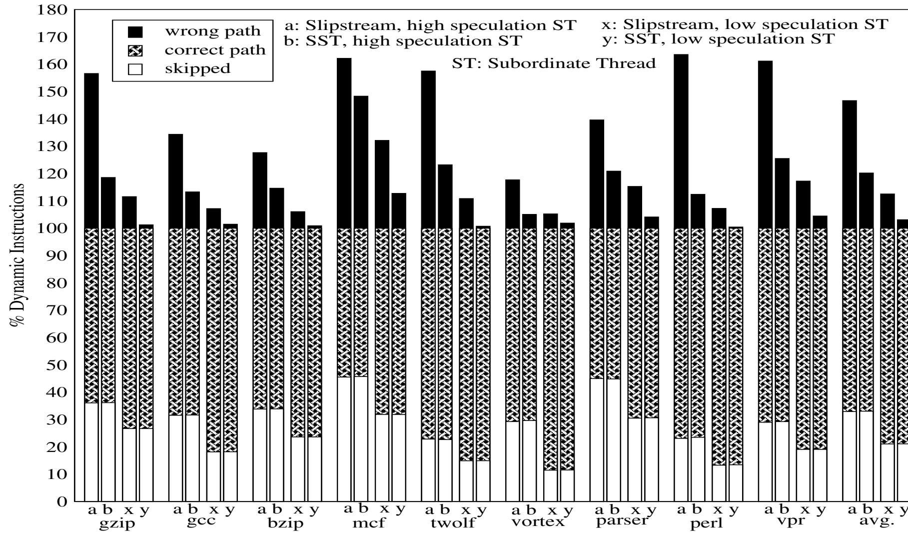

Figure 4.3: Work done by the subordinate thread on wrong paths for four schemes: (a) Slipstream with a highly speculative subordinate thread; (b) SST with a highly speculative subordinate thread; (c) Slipstream with a not too speculative subordinate thread; (d) SST with a not too speculative subordinate thread.

thread skips instructions (SST). In the third and fourth schemes (third and fourth bars, respectively), the subordinate thread is not too speculative. The third scheme corresponds to the slipstream processor and the fourth corresponds to SST.

It is clear from the first two sets of bars that the work done on the wrong path decreases significantly when the main thread skips instructions and advances faster. This is true for all the benchmarks, and agrees with our expectations. The same results are obtained for the last 2 sets of bars.

4.1.4 Performance Improvement with a Highly Speculative Subordinate Thread Versus a Not-Too-Speculative Subordinate Thread

In the previous subsection, we showed that the work done by the subordinate thread on wrong paths is significantly reduced in the SST configurations. It is higher though for a configuration with highly speculative subordinate thread compared to a configuration with a low-speculation subordinate thread as shown in Figure 4.3. However, the average performance for a configuration with the highly speculative subordinate thread is higher than that of the low-speculation subordinate thread, as shown in Figure 4.1. This is because the number of correct-path instructions executed by the subordinate thread for a configuration with the low-speculation subordinate thread (third and fourth bars of Figure 4.3) is much higher than that with a configuration with a highly speculative subordinate thread (first and second bars of Figure 4.3). This indicates that a low-speculation subordinate thread is much slower than a highly speculative subordinate thread. A highly speculative subordinate thread is able to perform a better job in hiding the long latency of critical memory instructions and branch miss-predictions, because it is faster (executing fewer non-critical instructions) and reaches those long latency instructions faster. Even though a highly speculative subordinate thread may end up skipping some critical instructions because it speculates aggressively, it still delivers more help to the main thread than a low-speculation subordinate thread. Note that if the subordinate thread is too aggressive in speculating and skipping instructions, such that all of its outcomes are incorrect, the main thread will end up executing most

of the instructions, rendering the subordinate thread useless. We will come across very speculative subordinate threads in Chapter 6.

4.1.5 Improvement in the Subordinate Thread L2 Cache Miss Rate

By analyzing all the benchmarks, we noticed that with symbiosis (main thread consuming results of the subordinate thread without executing their corresponding instructions) the L2 cache misses incurred by the subordinate thread decreased for almost all the benchmarks, while the number of L2 cache misses incurred by the main thread remained relatively unchanged for all benchmarks. This is shown in Figure 4.4. The first bar shows the L2 cache miss rate for a single thread scheme. The second and third bars show the distribution of L2 cache misses incurred among the subordinate thread and the main thread in the basic subordinate threading scheme (slipstream) and the SST scheme. The subordinate thread L2 cache misses are further divided into L2 cache misses that are useful to the main thread and L2 cache misses that are useless (i.e., do not provide any help to the main thread). It is clear from Figure 4.4 that with symbiosis (SST), the useless L2 misses decreased over all the benchmarks, especially for benchmarks *mcf*, *twolf*, *parser*, and *vpr*.

We like to point out that the decrease in L2 cache miss rate for the SST subordinate thread is because of the reduction in the speed gap between the main thread and the subordinate thread. When the main thread is running with a speed close to that of the subordinate thread, it is less likely that the main thread will throw pages out of the L2 cache that are needed by the subordinate thread in the

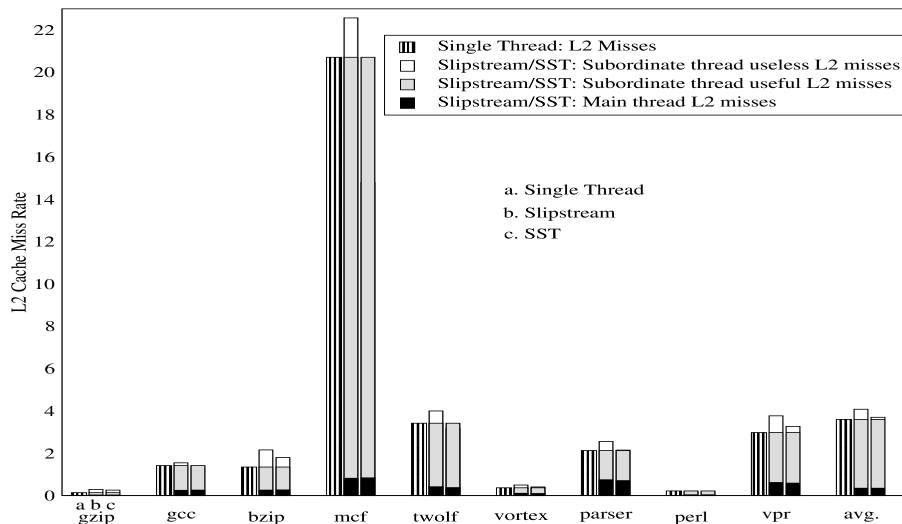

Figure 4.4: Distribution of average L2 cache misses obtained with: (a) Single thread; (b) Slipstream processor; and (c) SST.

near future. As a result, the subordinate thread L2 cache miss rate will improve. Also, with symbiosis, the main thread may consume the subordinate thread results of memory loads which reduces the number of times the main thread has to access memory. This again makes it less likely for the main thread to throw from the the L2 cache pages that are needed by the subordinate thread in the future. Finally, when the main thread runs faster, the memory updates made by the subordinate thread will be immediately done by the main thread. This reduces the speculative values used by the subordinate thread, which, in turn, reduces the number of memory accesses it performs with incorrect addresses, thereby reducing the number of useless L2 cache misses it may incur.

4.1.6 Improvement in the Main Thread L1 DCache Miss Rate

In order to understand the effect of memory symbiosis on the main thread L1 dcache misses in the SST scheme, we plotted the percentage of main thread total L1 dcache misses incurred when memory symbiosis is applied (white portion of first bar of Figure 4.5). By memory symbiosis, we mean letting the main thread consume the subordinate thread results of load instructions. It is clear from Figure 4.5 that the L1 dcache misses incurred by the main thread decreased with memory symbiosis, for all the benchmarks. On average, 8% of the L1 dcache misses incurred by the main thread were saved when memory symbiosis was applied (black portion of the first bar of Figure 4.5). The second bar of Figure 4.5 provides more statistics of the L1 dcache accesses done by the main thread when memory symbiosis is applied. A significant portion of the L1 dcache accesses done by the main thread is reduced (20% on average) when we apply memory symbiosis, as shown in the upper portion of the second bar of Figure 4.5. Together, the middle and lowest portions of the second bar in Figure 4.5 show the total L1 dcache accesses that are skipped by the main thread when memory symbiosis is applied. On average, almost 16% of the L1 dcache accesses skipped by the main thread with memory symbiosis would have caused an L1 dcache miss if the main thread did not skip them (lowest portion of the second bar in Figure 4.5).

Main Thread L2 Cache Misses: The main thread L2 cache misses do not show significant change with memory symbiosis, as shown in Figure 4.4. An insignificant decrease in the main thread L2 cache misses occurs with memory symbiosis though,

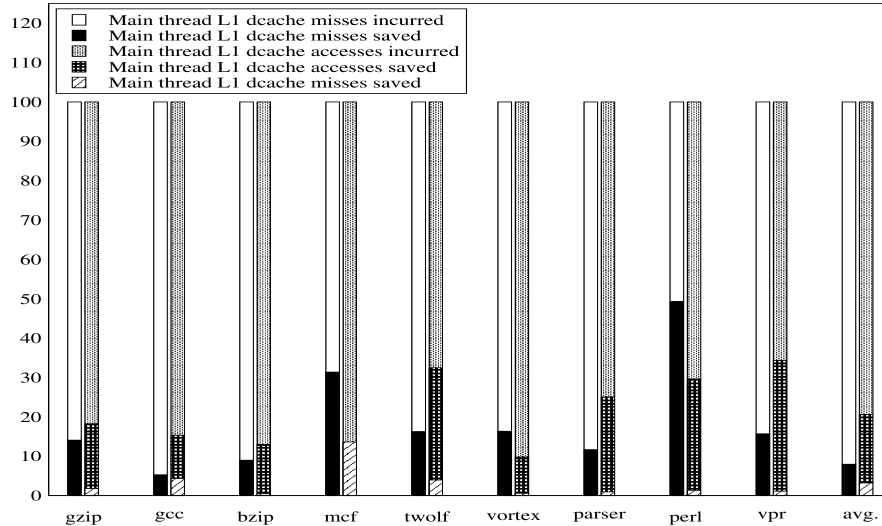

Figure 4.5: Main thread L1 dcache: (a) misses incurred and saved with SST when memory symbiosis is applied; and (b) accesses incurred and saved with SST when memory symbiosis is applied.

because of the overall decrease in the main thread memory accesses. However, a slight increase in the main thread L2 cache misses occur for benchmarks *mcf* and *vortex* because the main thread reaches the L2 cache misses faster before they are fully serviced by the subordinate thread.

4.1.7 Reduction in the Main Thread Branch Miss-predictions

In this subsection we show the advantage of our SST model over the slipstream model in reducing the branch miss-predictions incurred by the main thread. In SST, the main thread selectively consumes the subordinate thread branch outcomes only if they are non-data-speculative, and uses the predictions obtained from the branch predictor for all other branch instructions. This is in contrast to the slipstream model, in which the main thread blindly uses the subordinate thread outcomes

of branch instructions as predictions instead of the predictions obtained from the branch predictor. In Figure 4.6 we show the percentage of branch instructions that were miss-predicted in the main thread for three different processor models, a single thread model that uses the predictions of the branch predictor for all branch instructions (first bar), the slipstream model (second bar), and the SST model (third bar). As shown, the SST model performs the best with respect to reducing the branch miss-predictions incurred by the main thread, on average 58% less than the single thread model, while the slipstream model only reduced the branch miss-predictions of the main thread to an average of 40% less than the single thread model.

The subordinate thread may execute branches that have one or more data-speculative input operands, and this introduces more incorrect branch predictions in the main thread of the slipstream model. Those incorrect predictions may have been avoided if the subordinate thread followed the branch predictor predictions without executing the branch instruction. In other words, the predictions of the branch predictor are more accurate than the ones obtained by letting the subordinate thread execute the branch instruction with speculative input values. On the other hand, SST avoids introducing incorrect predictions into the main thread by not allowing the main thread to use all the subordinate thread branch outcomes as predictions. Rather, in SST, the subordinate thread outcomes of branch instructions that are executed using data-speculative input values are not trusted by the main thread. As a result, the main thread does not consume them from the subordinate thread, and instead follows the predictions of the branch predictor.

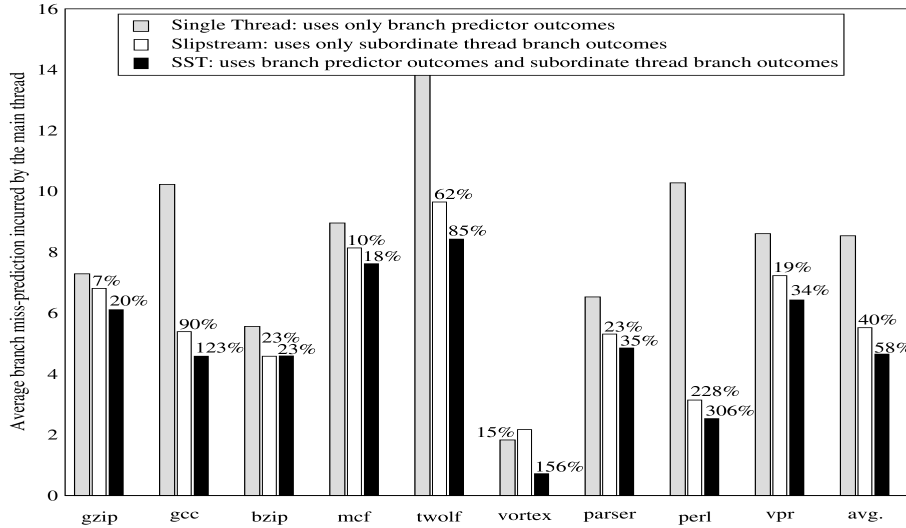

Figure 4.6: Main thread % of branch miss-predictions incurred when using: (a) The branch predictions obtained from a branch predictor for all branch instructions (single thread); (b) The branch predictions obtained from the subordinate thread for all branch instructions (slipstream); (c) The non-data-speculative branch outcomes of the subordinate thread, and the predictions obtained from the branch predictor for all other branch instructions (SST).

The SST model, however, is conservative because it considers all branch outcomes computed using speculative input values to be incorrect, which is not the case. Therefore, it wastes some opportunities in which the data-speculative branch outcomes of the subordinate thread are correct and may benefit the main thread by letting it avoid a branch miss-prediction penalty. This is not the case in the slipstream model where the main thread will benefit from all correct branch outcomes in the subordinate thread whether they are data-speculative or not. From the results shown in Figure 4.6, we can conclude that the data-speculative branch outcomes of the subordinate thread that are correct are fewer than the ones that

are incorrect, and hence the SST model wins.

Single Core Parameters	
Main Memory Latency	100+ cycles
ROB/LdStQ/FetchQ	size = 64/32/16 entries
Branch Penalty	16 cycles
SST-Specific Parameters	
Sub. Thread Re-start Penalty	16+ cycles
Memory Threshold	10 cycles

Table 4.2: Microarchitectural Parameters with Larger Cores

4.2 Performance Evaluation of SST Against DCE

In the previous section we highlighted the benefits of our SST model against the slipstream model. In this section we highlight the benefits of our SST model, using larger cores, with larger window size to be able to serve more in-flight instructions at the same time. Larger window sizes are especially suitable for long latency memory instructions because they can explore more in-flight instructions and hence can serve more than one L2 cache miss at the same time. We also use much larger L2 miss latency, at least a 100 cycles, and 16 cycles branch miss-prediction penalty. The new parameters are shown in Table 4.2.

We performed several experiments with varying L2 cache miss latencies. In all the experiments, we let the subordinate thread run ahead when it encounters an instruction that results in an L2 cache miss, by supplying an invalid value for

its output operand. A memory instruction is considered long latency if it reached the head of the ROB and blocked the subordinate thread pipeline for 10 cycles. This allows the subordinate thread to run even faster with a much wider instruction window. Hence, it reaches the long latency memory instructions faster than before. That makes it more suitable for prefetching. However, it may go on the wrong path much sooner.

We compare our SST results against a dual-core execution paradigm (DCE) [36], that was proposed to accelerate sequential programs. DCE is similar to our SST model and both of them share a similar high-level architecture: two processors connected via a FIFO queue. DCE consists of two superscalar cores, a front processor and a back processor. The front processor resembles the subordinate thread and the back processor resembles the main thread in our terms. The front processor executes instructions except for long latency cache misses, it instead produces an invalid value instead of blocking the pipeline, similar to runahead execution [19]. The front processor also forwards all its results to the back processor, which uses them as predictions similar to the slipstream processor.

4.2.1 IPC Improvement of SST without Memory Symbiosis (100 Cycles for Main Memory Access)

Figure 4.7 presents the IPC obtained for three schemes: a single thread scheme, a base subordinate threading scheme (DCE), and SST. We let the main thread in SST consume results of the subordinate thread only for non-memory instructions

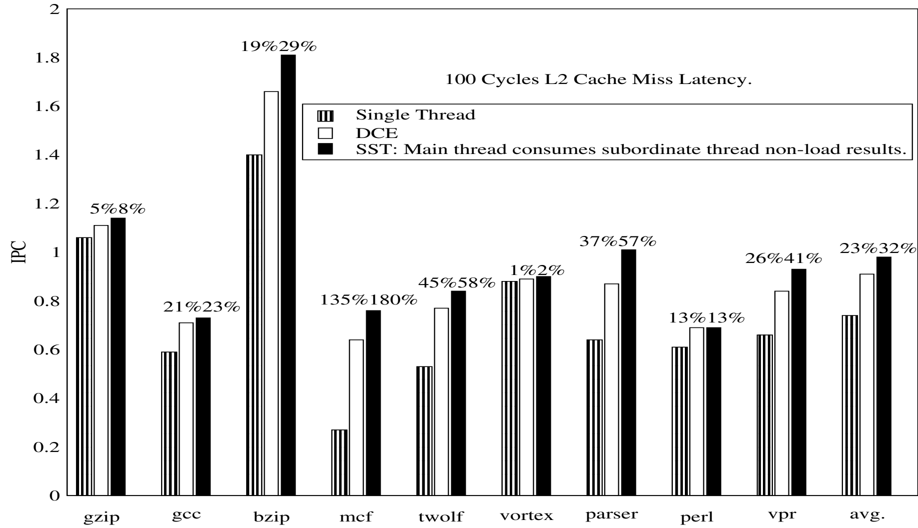

Figure 4.7: IPC obtained with memory latency 100 cycles for: (a) Single thread scheme; (b) DCE scheme; and (c) SST scheme (main thread consumes the results of the subordinate thread for only non-memory instructions.

(non-memory symbiosis). We also let the main memory latency be 100 cycles. There are three bars, corresponding to each of the three schemes. It is clear from Figure 4.7 that SST performs better than the other schemes. The average performance improvement of SST is 9% against the DCE scheme. For some benchmarks like *bzip*, *mcf*, *twolf*, *parser*, and *vpr*, there is a significant performance improvement over DCE.

4.2.2 IPC Improvement of SST with Memory Symbiosis (100 Cycles for Main Memory Access)

We performed another experiment in which we allowed the main thread to consume the subordinate thread results of load instructions as well (memory sym-

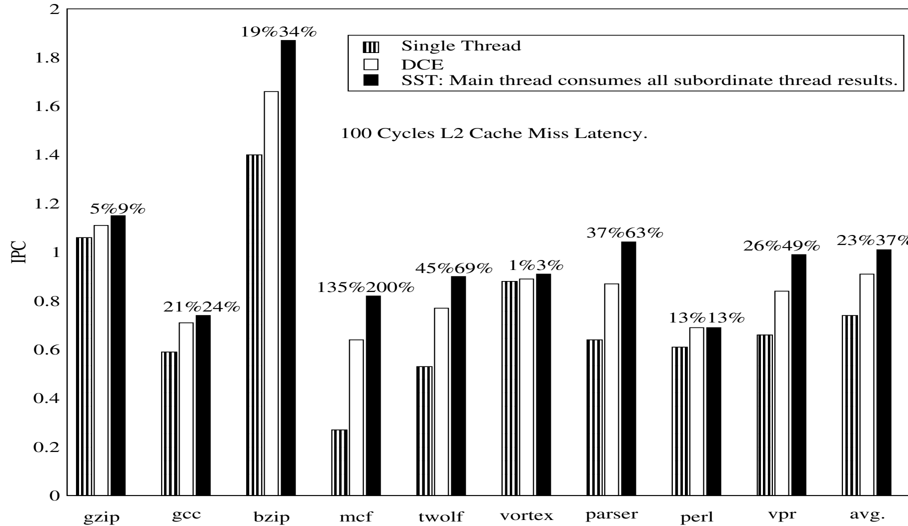

Figure 4.8: IPC obtained with memory latency 100 cycles for: (a) Single thread scheme; (b) DCE scheme; and (c) SST scheme (main thread consumes the results of the subordinate thread for all types of instructions).

biosis). We let the main memory latency be 100 cycles. Figure 4.8 shows the IPC obtained for the three schemes, a single thread scheme, the DCE scheme, and SST (first, second, and third bar, respectively). It is clear from Figure 4.8 that SST performs better than the other schemes. The average performance improvement of SST in this case is 14% against the DCE scheme. Again, there is a significant performance improvement for benchmarks *bzip*, *mcf*, *twolf*, *parser*, and *vpr*.

4.2.3 IPC Improvement of SST with Memory Symbiosis (300 Cycles for Main Memory Access)

We also evaluated our SST scheme with a 300 cycle L2 cache miss latency. The IPCs for the three schemes (single thread, DCE, and SST) are shown in Figure

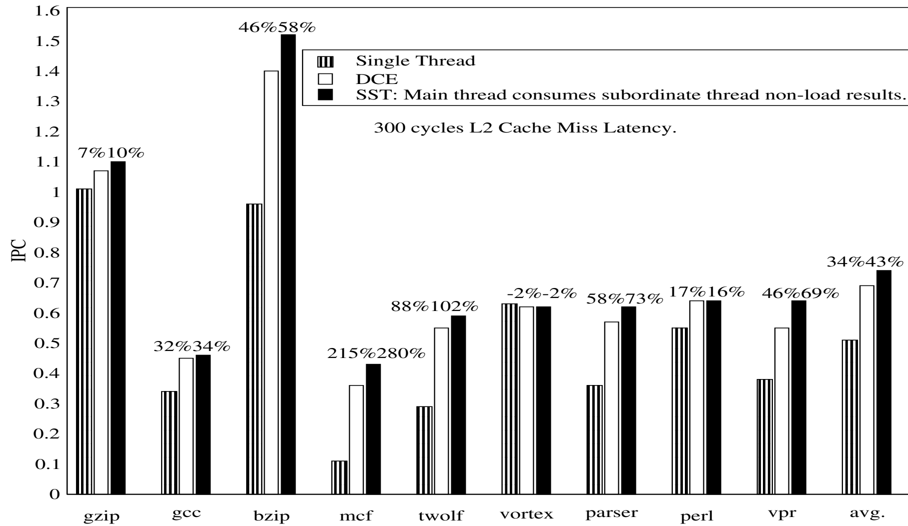

Figure 4.9: IPC obtained with memory latency 300 cycles for: (a) Single thread scheme; (b) DCE scheme; and (c) SST scheme (main thread consumes the results of the subordinate thread for only non-memory instructions.

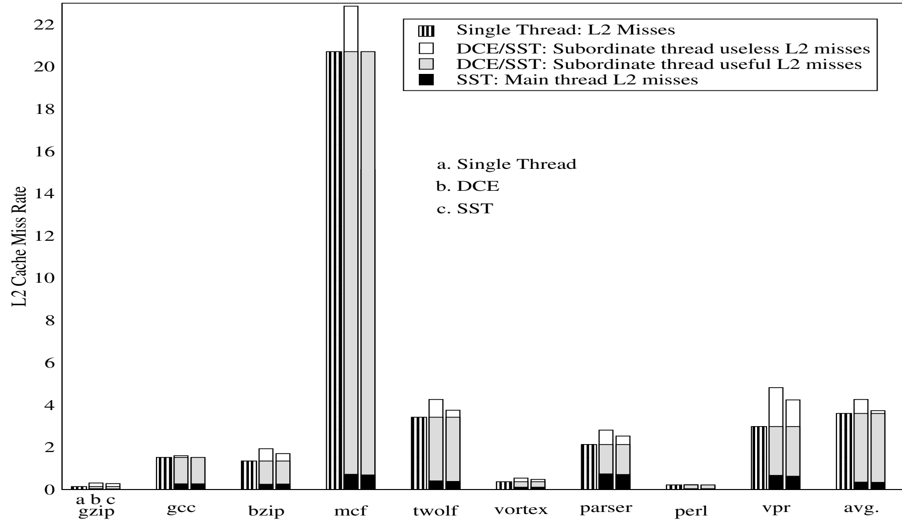

Figure 4.10: Distribution of average L2 cache misses obtained with memory latency 300 cycles for: (a) Single thread scheme; (b) DCE scheme; and (c) SST scheme (main thread consumes the results of the subordinate thread for only non-memory instructions).

4.9 (first, second and third bars respectively). Symbiosis was enabled for only non-memory instructions. The IPC for all three schemes across all benchmarks dropped further when the L2 cache miss penalty increased to 300 cycles as expected. It is clear from Figure 4.9 that SST still performs better than the single thread and the DCE schemes. The average performance improvement of SST is 9% against DCE.

4.2.4 Reduction in the Subordinate Thread L2 Cache Miss Rate

By analyzing all the benchmarks, we noticed that with symbiosis the L2 cache misses incurred by the subordinate thread decreased for almost all the benchmarks, while the number of L2 cache misses incurred by the main thread remained relatively unchanged for all benchmarks. This is shown in Figure 4.10. The first bar shows the number of L2 cache misses incurred in the single thread scheme. The second and third bars show the distribution of L2 cache misses incurred among the subordinate thread and the main thread in the basic subordinate threading scheme (DCE) and SST. The subordinate thread L2 cache misses are further divided into L2 cache misses that are useful to the main thread and L2 cache misses that are useless (i.e., do not provide any help to the main thread). It is clear from Figure 4.10 that with symbiosis, the useless L2 cache misses decreased over all the benchmarks, especially for benchmarks *bzip*, *mcf*, *twolf*, *parser*, and *vpr*. These results agree with what we obtained earlier in the previous section. However, the subordinate thread in the SST scheme incurred more useless L2 cache misses. This is explained by noting that the subordinate thread in SST run ahead when it encounters an L2 cache miss, and so

runs much faster than the subordinate thread in the previous section. This makes the speed gap between the main thread and the subordinate thread much larger, making it more likely for the subordinate thread to perform memory accesses with invalid addresses and hence generating useless L2 cache misses.

The subordinate thread L1 cache misses improved with SST just as in the previous section. Also, with SST the number of branch miss-predictions decreased in the main thread just like in the previous section.

Chapter 5

An Optimized Implementation of SST

In this chapter we present a new microarchitecture of SST that captures all of the features of the old SST design but is more efficient. We identify some inefficiencies with regard to distilling the subordinate thread and recovering it from the wrong path. In the new implementation of SST the subordinate thread is aware of its own speculation. This has several advantages. First, a speculative-aware subordinate thread can avoid executing instructions with data-speculative input values. This makes the subordinate thread faster, as it will execute fewer useless instructions. Also, this makes our SST more efficient because the main thread will execute those data-speculative instructions anyway, and so it would be redundant if the subordinate thread also executes them.

We also provide a simple recovery scheme for the subordinate thread when it goes on a wrong path that takes advantage of the dual-purpose core we provide in the new SST design to ensure a quick re-start for the subordinate thread after its recovery as well as eliminate the need for a shadow register file. In the new SST design, each core may play the role of a subordinate thread or a main thread, and both cores are coupled with a FIFO queue that operates in both directions. In this way, recovering the subordinate thread involves a simple switch mechanism to the role of each core (from a subordinate thread role to a main thread role and vice

versa) and to the direction of information flow on the FIFO queue.

In this chapter we only discuss the newly added parts and issues concerning the new design. Any other details are assumed to be identical to the old design and we refer the reader to Chapter 3. Finally, we present our results for the new SST design and compare it to the old SST design and to a subordinate threading scheme that does not employ symbiosis, the dual-core execution paradigm (DCE) [36].

5.1 A Partially Speculative-Aware Subordinate Thread

In the old design of SST, the subordinate thread has no account of the registers that contain data-speculative values in its register file or memory addresses that contain data-speculative values. As a result, it executes instructions that take, data-speculative values as input. It is highly probable that the subordinate thread will produce incorrect results for data-speculative instructions, thereby wasting its execution bandwidth on useless instructions, and limiting its instruction window size. With branch instructions it is even worse, because the subordinate thread may obtain a correct prediction from the branch predictor, and yet go on the wrong path because it executed the branch instruction with incorrect (data-speculative) input values. Finally, the main thread will anyway execute those instructions because they are data-speculative in the subordinate thread, and therefore it is redundant that the subordinate thread executes them. We introduce a simple mechanism for making the subordinate thread self aware of the speculations it makes. This aids it in making better decisions concerning which instructions to include and which

instructions to exclude, i.e., the distillation process. This also eliminates further redundant executions as well as speeds up the subordinate thread.

Distilling the Subordinate Thread in the Old SST Design: In the old design of SST, branch instructions that are highly predictable are identified using the saturating counters in the branch predictor. They are then marked and their backward slices that currently reside in the pipeline are also marked. They are all then converted to no-ops, freeing up all the resources they hold. Also, in the old design, long-latency memory instructions are removed from the pipeline as well when they arrive at the ROB head and block the pipeline. An invalid (speculative) value is supplied to their dependent instructions that currently reside in the pipeline, which, in turn, are also converted to no-ops. The old design, however, cannot identify any instructions further that are dependent on the ones removed. In other words, distillation occurs only for the window of instructions that happen to exist in the pipeline at the time when the long-latency memory instruction was identified or when the highly predictable branch instruction was identified.

Distilling the Subordinate Thread in the New SST Design: In the new design of SST, the subordinate thread includes an RSB and an MSB just like the main thread, to aid it in identifying registers that contain data-speculative values as well as memory locations that contain data-speculative values. Any instruction that is converted to a no-op or is identified as a long-latency instruction marks the corresponding bit of its output operand in the RSB as data-speculative. All

store instructions mark the corresponding bit of their hashed address in the MSB as data-speculative. In this way, the subordinate thread keeps track of data-speculative registers or memory locations. Later, if an instruction that uses a data-speculative value as its input arrives at the pipeline, it is automatically identified as a data-speculative instruction and is converted to a no-op. Hence, with the aid of the RSB and the MSB, the subordinate thread now can identify instructions that are dependent on the removed ones even if they arrive at the pipeline much later.

Operation of the Subordinate Thread RSB: The RSB is treated in the subordinate thread in the same way as in the main thread of the old and new designs of SST, but with minor differences. It is read and updated by the subordinate thread in the dispatch and writeback stages just as in the main thread. However, instructions that arrive at the ROB head and are identified as long-latency instructions are treated differently. Initially, those instructions are non-data-speculative and so they do not mark any bits in the RSB. When they arrive at the ROB head and become identified as long-latency instructions, the RSB bit corresponding to their output operand is marked as data-speculative in the writeback stage. Also, for all instructions currently residing in the pipeline that are dependent on the long-latency instruction, the RSB bit corresponding to their output operands is marked as data-speculative. The same also happens to highly predictable branches and their backward slices that currently reside in the pipeline.

Operation of the Subordinate Thread MSB: The MSB is read and updated by the subordinate thread in the dispatch and writeback stages just like the RSB. Initially, all of its bits are set to zeroes, indicating that all memory addresses in the subordinate thread contain non-data-speculative values. Unmarking the MSB bits occurs only when the subordinate thread recovers from a miss-speculation. Again, long-latency instructions and their dependency chains as well as highly predictable branches and their backward slices are treated as in the case with the RSB.

The RSB and MSB of the subordinate thread, however, do not reflect an accurate picture of the data-speculative register and memory locations. That's why the main thread in the new design must maintain another set of RSB and MSB.

Inaccuracy of the Subordinate Thread MSB: We identify a case that renders the subordinate thread MSB as less accurate than the main thread MSB (of the old and new designs). This case is of a store instruction whose address cannot be determined because its input operands are data-speculative, and so the corresponding bit of its hashed address cannot be marked in the MSB. Hence, the marked bits of the subordinate thread MSB only reflect a subset of the total bits that should be marked. The full set is represented by the marked bits in the main thread MSB.

Inaccuracy of the Subordinate Thread RSB: The inaccuracy of the subordinate thread MSB is reflected upon the RSB. If a load instruction that has a non-data-speculative address arrives, it may very well read from the address that was not marked in the subordinate thread MSB, and so it is not really non-data-

speculative. However, it will not mark the bit corresponding to its output register as data-speculative in the RSB, rendering the subordinate thread RSB inaccurate. Therefore, the marked bits of the subordinate thread RSB do not reflect the full speculative state of the register file.

Note that the slight inaccuracy of the subordinate thread RSB and MSB affects the number of instructions that will be distilled out of the subordinate thread. That may cause the subordinate thread to execute more instructions because it failed to identify some as data-speculative. That is tolerable as the SST scheme achieves considerable improvements, as will be shown in the result section.

Fully Speculative-Aware Subordinate Threads: Note that the case of a store instruction that has a speculative address can be handled in the subordinate thread by simply marking all the bits of its MSB as data-speculative. This makes the subordinate thread skip all subsequent load instructions, as well as their dependency chains. This conservative approach ensures that the subordinate thread is fully aware of its own speculation. However, the subordinate thread then may skip too many instructions, thereby becoming too speculative to be useful.

5.2 The Subordinate Thread Recovers from Miss-Speculation By Switching Roles with the Main Thread

Because each core in the new design of SST is symmetric, i.e., they both contain the same hardware, each may act as a main or subordinate thread. In the

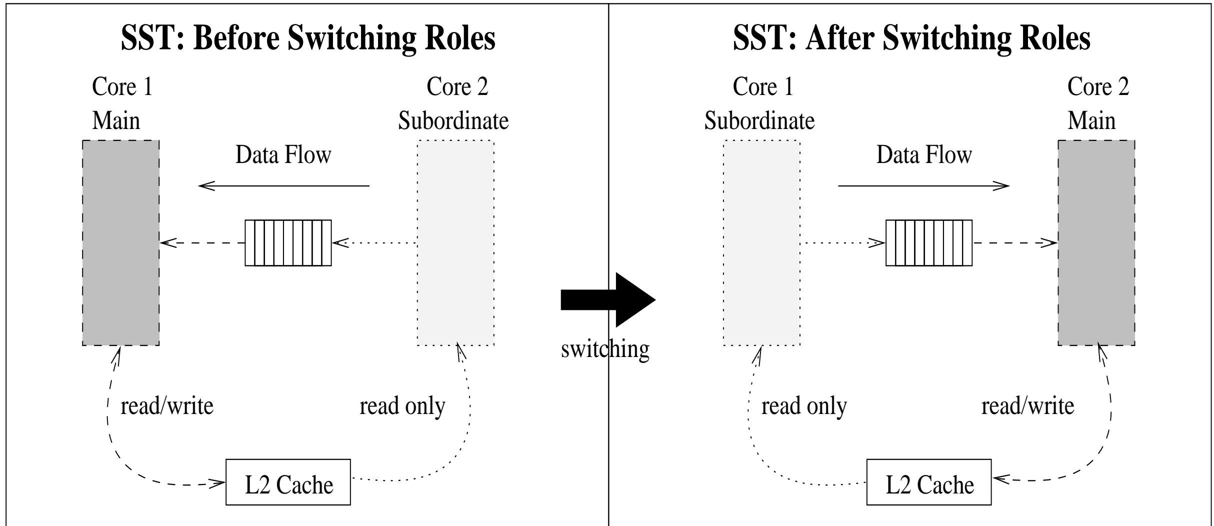

Figure 5.1: Subordinate thread and main thread switch roles after recovery of the subordinate thread from miss-speculation.

new design, we let the cores switch roles when recovering the subordinate thread from a miss-speculation, i.e., the main thread becomes the subordinate thread and the subordinate thread becomes the main thread. This aids in faster recovery of the subordinate thread from a miss-speculation, by eliminating the penalty associated with squashing and re-starting the subordinate thread (in the old design, when the subordinate thread re-starts after miss-speculation, it takes a number of cycles equal to the pipeline depth until it can produce the first result and place it on the FIFO queue for the main thread consumption, during which time the main thread is blocked, waiting for the subordinate thread results).

Switching Roles: When the main thread detects that the subordinate thread has gone on a wrong path it initiates recovery. This involves the subordinate thread

switching its register file with the shadow register file to have a clean copy of the register file and copying the program counter from the main thread. Also, all entries of the L1 dcache of the subordinate thread are invalidated, and then the subordinate thread is squashed. In the new design, once these steps are done, the main thread and the subordinate thread switch roles as shown in Figure 5.1. After switching, the subordinate thread has the correct memory and register file state (which belonged originally to the main thread) and can begin placing results onto the FIFO queue without any delays, as its pipeline is full. The flow of data on the FIFO queue is also switched, and so the main thread reads the results of the subordinate thread at the new end of the FIFO queue. Also, the accesses permission to the L2 cache are switched.

Using FIFO Queue Instead of Shadow Register File: In the new SST design we can eliminate the use of the shadow register file by using the FIFO queue instead. Because of switching roles, the FIFO queue can be used to transfer data in both directions. Hence, it can be used by the main thread to forward its register file values to the subordinate thread upon recovery. Once the main thread has written all of its register file values, it can switch its role to the subordinate thread role, and when the subordinate thread copies the register values from the FIFO queue, it can switch its role to the main thread role. The use of the FIFO queue to transfer the register values of the main thread to the subordinate thread upon recovery may introduce slight delays, but it has the advantage of eliminating the hardware associated with the shadow register file.

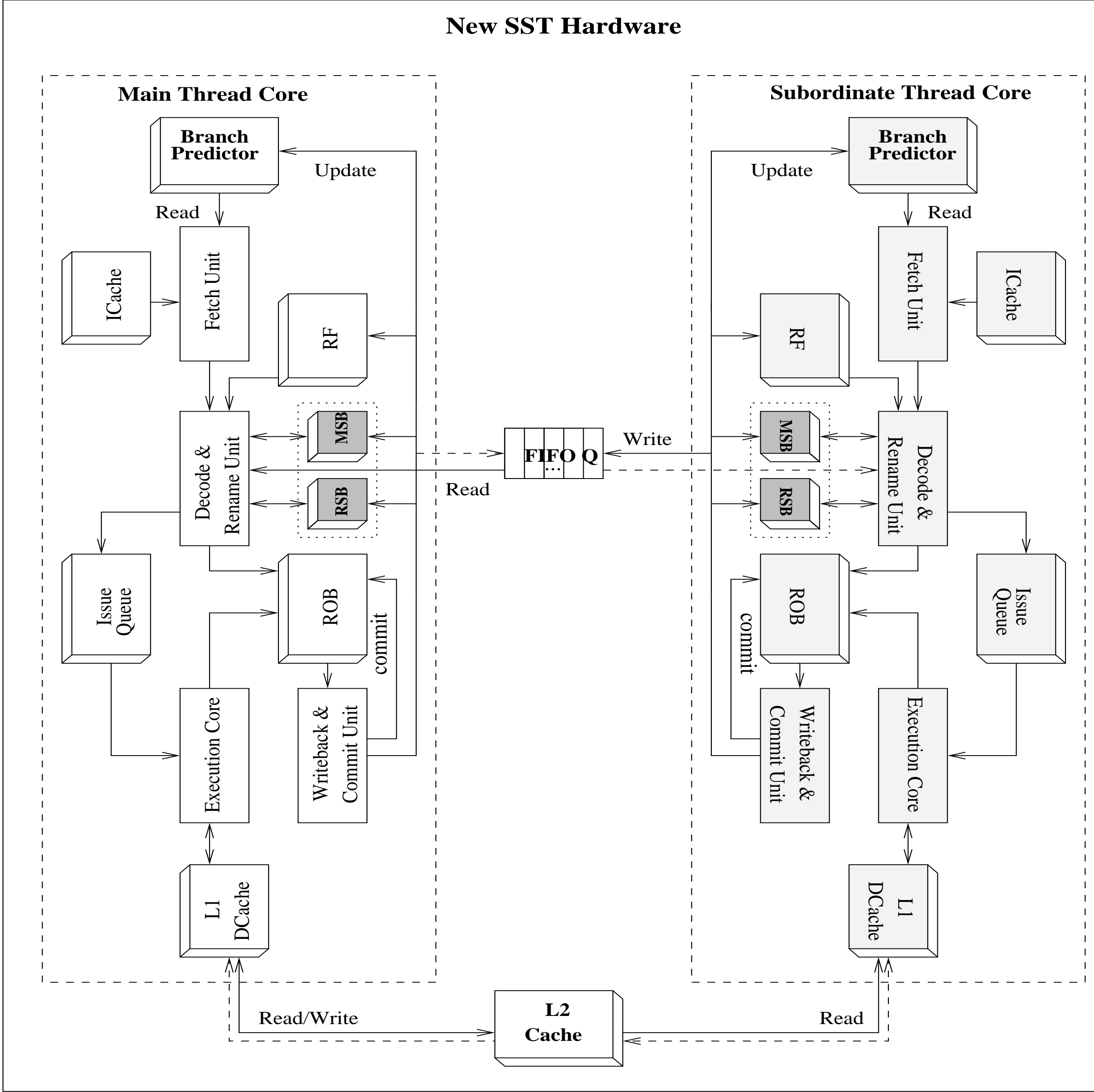

Figure 5.2: New SST Microarchitecture.

5.3 New SST Microarchitecture

Figure 5.2 shows the new SST microarchitecture. The hardware for identifying data-speculative outcomes of the subordinate thread is distributed among both the subordinate thread and the main thread in the new microarchitecture. The subordinate thread includes, an RSB to help it identify its own registers that contain non-data-speculative values, as well as an MSB to help it filter out memory addresses that are written by store instructions. The main thread includes an RSB and an MSB as in the older implementation to help it identify non-data-speculative outcomes of the subordinate thread. The RSB and MSB are still needed in the main thread because the subordinate thread RSB and MSB are not very accurate in the case of a partially speculative subordinate thread. In the case of a fully speculative subordinate thread, the main thread need not keep an RSB and an MSB. Note that the cores are symmetric, and so each core can act as a main thread as well as a subordinate thread. The data flow on the FIFO buffer is now in both directions. Other than these, both the old and new microarchitectures are identical.

5.4 Experimental Results

In this section we present the results we obtained by letting the subordinate thread keep track of its own speculations and recovering the subordinate thread by switching its role with the main thread. In order to study the new SST model, we developed a simulator that models the new SST scheme, which is an extension of the SST simulator developed earlier for the old SST design. The microarchitectural

parameters we used are shown in Table 5.1. The L1 dcache of a subordinate thread is invalidated on its recovery from the wrong paths. All cores use a single branch predictor, which is only updated by the main thread.

In our simulations, the subordinate thread treats long-latency memory instructions that reach the ROB head and block the pipeline as in runahead execution [19]; it supplies an invalid value and retires the blocking memory instruction before it is serviced. For a speculative-aware subordinate thread, we use the RSB and MSB to further distill it, and we disable them for a speculative-unaware subordinate thread.

In order to show the benefits of our new SST scheme, we compare its performance with the old SST scheme presented in the previous chapter. We also compare its performance against the DCE model (in which the main thread executes every instruction) [36]. We compare four different schemes together: (1) A subordinate threading scheme in which the subordinate thread is speculative-unaware, and the main thread does not skip any instruction (DCE), (2) its SST version, i.e., the main thread consumes the subordinate thread results without executing their corresponding instructions, (3) a subordinate threading scheme in which the subordinate thread is speculative-aware and the main thread does not skip any instruction (an extension of DCE), and (4) its SST version. Note that both the DCE

Single Core Parameters	
L1 ICache	sz/assoc/repl/ln/lat=16KB/1way/LRU/64B/1cycle
L1 DCache	sz/assoc/repl/ln/lat=64KB/4way/LRU/64B/1cycle
L2 Cache (data+instrs.)	sz/assoc/repl/ln/lat=1024KB/8way/LRU/128B/6cycles
Main Memory Latency	100 cycles
Fetch/issue/retire	Bandwidth = 4/4/4
ROB/LdStQ/FetchQ	size = 64 entries/32 entries/16 entries
Branch Predictor	type = bimodal, size = 32K entries
Branch Penalty	16 cycles
Superscalar-Specific Parameters	
Fetch/issue/retire	Bandwidth = 8/8/8
ROB/LdStQ	size = 128/64/32 entries
SST-Specific Parameters	
MSB	64 bits
FIFO Queue	latency/Bandwidth/size = 2 cycles/5 instrs./32 instrs.
Branch/Mem Thresholds	Branch Count = 1, Memory Cycles = 10
Sub. Thread Recovery	16 cycles

Table 5.1: Microarchitectural Simulation Parameters for Old & New SST

and SST schemes we use in the comparison use the same type of subordinate thread, i.e., distilled in the same way. This is to have a fair comparison when comparing the schemes with a speculative-unaware subordinate thread and with a speculative-aware subordinate thread.

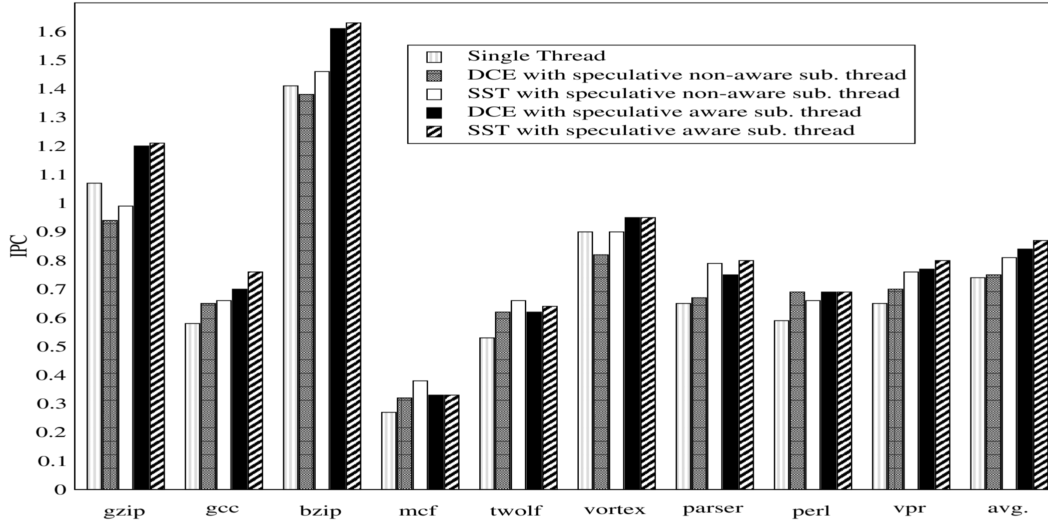

Figure 5.3: IPC for 5 schemes: (a) Single thread (a superscalar that combines two cores in one); (b) DCE with speculative unaware subordinate thread; (c) SST with speculative unaware subordinate thread; (d) DCE with speculative-aware subordinate thread; (e) SST with speculative-aware subordinate thread.

5.4.1 IPC Improvement

The IPCs we obtained for a single thread (superscalar with double the issue width of a single core, Table 5.1), a DCE scheme with speculative-unaware subordinate thread, an SST scheme with speculative-unaware subordinate thread, a DCE scheme with speculative-aware subordinate thread, and an SST scheme with speculative-aware subordinate thread, are presented in Figure 5.3. Each bar corresponds to one of the five schemes. The percentage improvement over a single thread for each of the four schemes is also shown in Figure 5.4. For almost all the benchmarks, the SST scheme and the DCE scheme with a speculative-aware subordinate thread (third and fourth bars) outperform the other two schemes except for

two benchmarks *mcf* and *twolf*. Those two benchmarks are memory bound and the speculative-unaware subordinate thread executes most of the data-speculative loads and stores, which yields correct results. In other words, the speculative-unaware subordinate thread performs value predictions for the memory addresses and this helps in case of benchmarks, *mcf* and *twolf*.

Note that the SST scheme of bar 2 outperforms the DCE scheme of bar 1 for almost all the benchmarks with the exception of benchmark *perl*. However, the performance of the SST scheme in bar 4 has become close to that of DCE in bar 3. This can be explained by the branch miss-predictions whose latencies the subordinate thread is able to hide. Recall that the subordinate thread in both schemes is speculative-aware, and so it only executes the branch instructions that are not data-speculative and skips all other branch instructions (and follows the prediction of the branch predictor). That makes the DCE and the SST schemes equivalent in terms of the number of subordinate thread correct branch outcomes consumed by the main thread. It also makes both schemes equivalent in terms of the number of branch miss-predictions incurred by the main thread because in both schemes the main thread will follow the prediction of the branch predictor for those branch instructions that were not computed by the subordinate thread. The DCE will follow the ones of the subordinate thread that have a prediction almost identical to that of the main thread branch predictor.

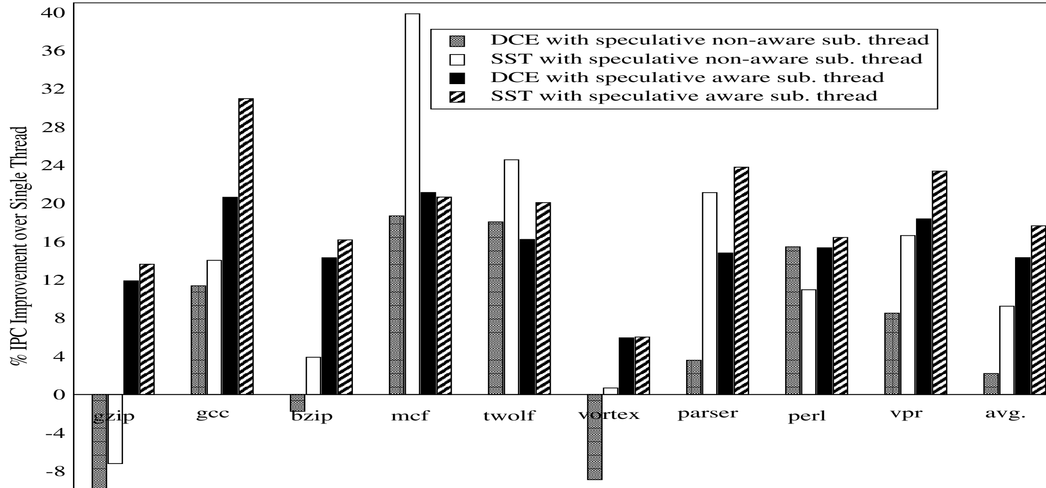

Figure 5.4: Percentage IPC improvement over a single thread (a superscalar that combines two cores in one) for four schemes: (a) DCE with speculative unaware subordinate thread; (b) SST with speculative unaware subordinate thread; (c) DCE with speculative-aware subordinate thread; (d) SST with speculative-aware subordinate thread.

5.4.2 Branch Miss-predictions in The Main Thread

From the above discussion we can say that the number of branch miss-predictions incurred by the main thread in the DCE and SST schemes that employ a speculative-aware subordinate thread must be roughly equal. We confirm this by plotting the percentage of branch miss-predictions of the main thread in Figure 5.5 for the same five schemes as above. The last two bars (bar 4 and bar 5) correspond to the DCE and SST schemes with a speculative-aware subordinate thread. Note that they are almost the same across all benchmarks, as expected. The percentage of branch miss-predictions incurred by the main thread for the DCE and SST schemes with a speculative-unaware subordinate thread is shown in bar 2 and 3, respectively. For

almost all the benchmarks, the DCE and SST schemes with a speculative-aware subordinate thread has fewer branch miss-predictions in the main thread than the other two schemes that employ a speculative-unaware subordinate thread, with the exception of benchmark *perl*. Finally, the DCE scheme with a speculative-unaware subordinate thread (bar 2) has the most number of branch miss-predictions in the main thread, which even exceeds that of the single thread. This is because the DCE scheme treats all of the branch outcomes of the subordinate thread as predictions instead of the predictions given by the branch predictor. This introduces extra branch miss-predictions into the main thread because the speculative-unaware subordinate thread executes almost all the speculative branch instructions and forwards their speculative results to the main thread, which trusts them and consumes them instead of the predictions given by the branch predictor. With symbiosis, most of those incorrect predictions are eliminated, as evident from the drop in the percentage of branch miss-predictions (bar 3).

5.4.3 Branch Miss-predictions in the Subordinate Thread

In Figure 5.6 we show the percentage of subordinate thread incorrect branch outcomes for four schemes: DCE with speculative-unaware subordinate thread, SST with speculative-unaware subordinate thread, DCE with speculative-aware subordinate thread, and SST with speculative-aware subordinate thread. The incorrect branch outcomes of the subordinate thread are either incorrect predictions of

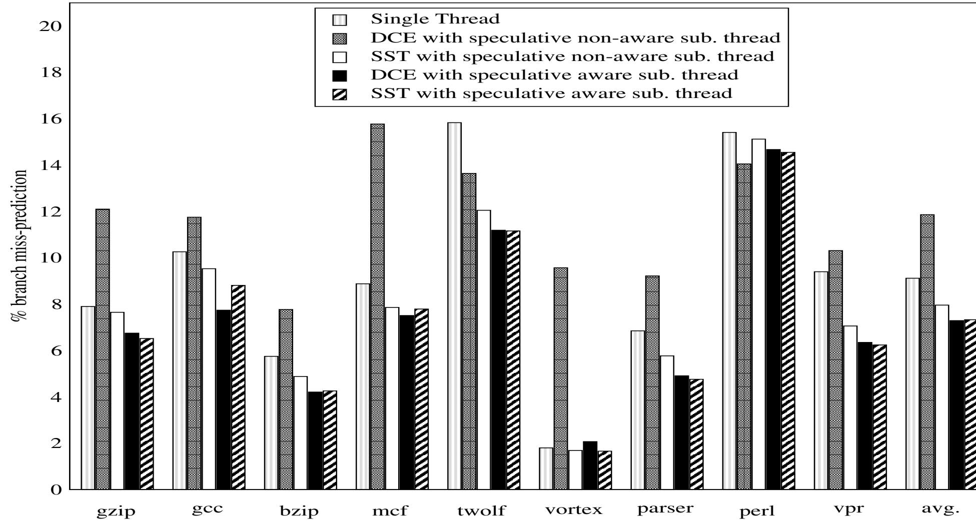

Figure 5.5: Percentage branch miss-predictions incurred by the main thread for five schemes: (a) Single thread (a superscalar that combines two cores in one); (b) DCE with speculative-unaware subordinate thread; (c) SST with speculative-unaware subordinate thread; (d) DCE with speculative-aware subordinate thread; (e) SST with speculative-aware subordinate thread.

the branch predictor that it followed without executing the corresponding branch instructions, or those computed by itself using data-speculative input values. In the DCE and SST schemes that use a speculative-unaware subordinate thread (bar 1 and bar 2, respectively), both type of incorrect branch outcomes exist, but in the DCE and SST schemes that use a speculative-aware subordinate thread, all incorrect branch outcomes correspond to only incorrect predictions obtained from the branch predictor. As shown in Figure 5.6, the speculative-aware subordinate thread has the least number of incorrect branch outcomes (bars 3 and 4) compared to the speculative-unaware subordinate thread (bars 1 and 2), with the exception of benchmark *perl*. For benchmark *perl*, using value prediction for the branch outcomes

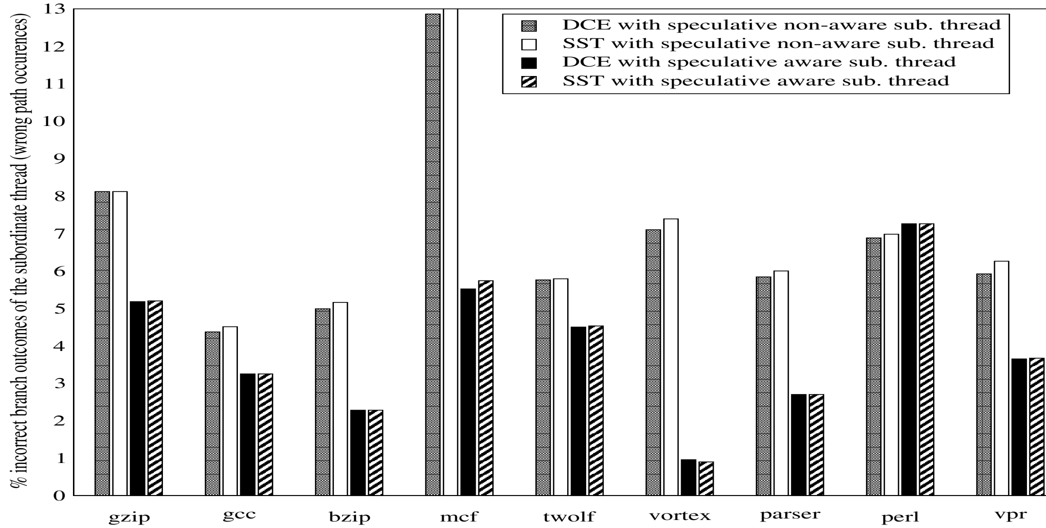

Figure 5.6: Percentage of incorrect branch outcomes of the subordinate thread for four schemes: (a) DCE with speculative-unaware subordinate thread; (b) SST with speculative-unaware subordinate thread; (c) DCE with speculative-aware subordinate thread; (d) SST with speculative-aware subordinate thread.

in the subordinate thread is more effective.

5.4.4 L2 Cache Miss Rate

We also show the impact of having a speculative-aware subordinate thread on the main thread L2 cache miss rate. It is clear from Figure 5.7 that the main thread L2 cache miss rate for the DCE and SST schemes with a speculative-aware subordinate thread (fourth and fifth bars) is much higher than that of the other schemes with a speculative-unaware subordinate thread (second and third bars). This confirms that data-predictions for calculating memory addresses in the speculative-unaware subordinate thread was relatively accurate, thereby making the subordinate thread

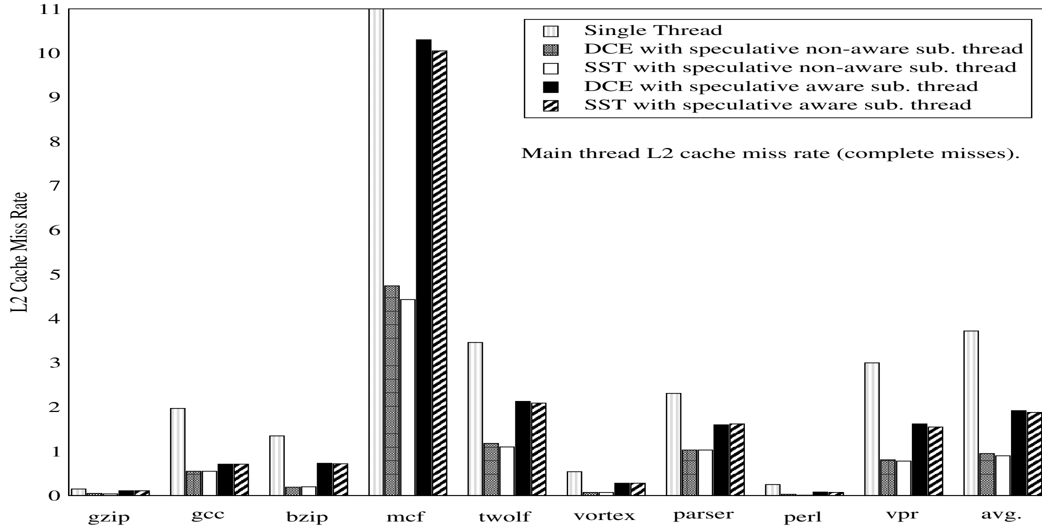

Figure 5.7: L2 cache miss rate (only complete misses) in the main thread for five schemes: (a) Single thread (a superscalar that combines two cores in one); (b) DCE with speculative-unaware subordinate thread; (c) SST with speculative-unaware subordinate thread; (d) DCE with speculative-aware subordinate thread; (e) SST with speculative-aware subordinate thread.

more effective in prefetching. On the other hand, the DCE and SST schemes with a speculative-aware subordinate thread did not benefit from the value predictions and simply skipped the data-speculative memory instructions.

5.4.5 Reduction in the Total Number of Executed Instructions

The speculative-aware subordinate thread skips the data-speculative instructions in addition to the highly predictable branches and long-latency instructions, while the speculative-unaware subordinate thread skips only the highly predictable branches and long-latency instructions. Therefore, the number of instructions executed by a speculative-aware subordinate thread is less than that of a speculative-

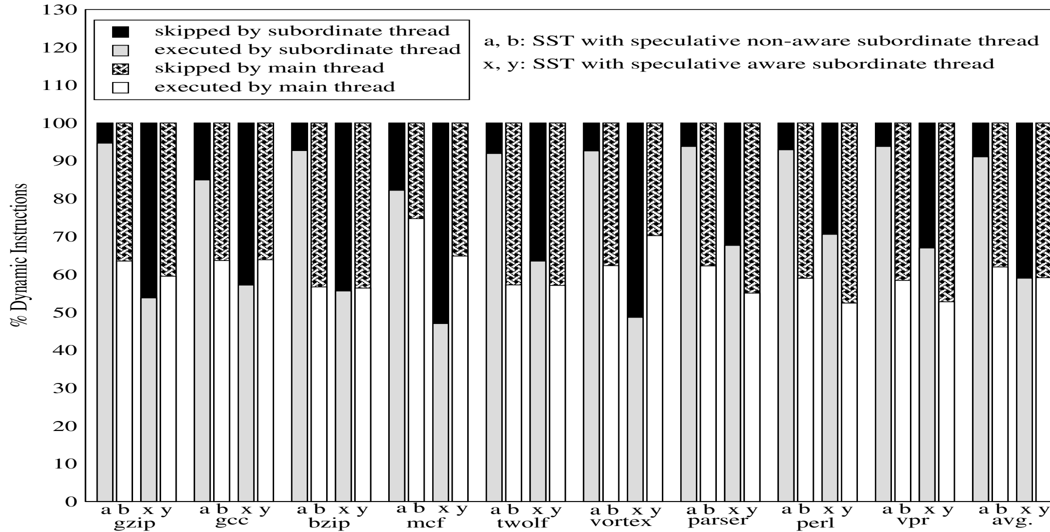

Figure 5.8: Distribution of skipped and executed instructions in the main thread and the subordinate thread for two schemes: (a) SST with speculative-unaware subordinate thread; (b) SST with speculative-aware subordinate thread.

unaware subordinate thread. However, the main thread consumes relatively the same number of results from either threads because each thread must have executed the same number of non-data-speculative instructions. This implies that the total number of instructions executed in an SST scheme with a speculative-aware subordinate thread must be lower than that of an SST scheme with a speculative-unaware subordinate thread. In order to confirm that, we consider the number of instructions executed in the main thread and the subordinate thread for two schemes: SST with speculative-unaware subordinate thread and SST with speculative-aware subordinate thread (the first two bars and the last two bars in Figure 5.8, respectively). The first and third bars show the number of instructions executed by the subordinate thread versus the number of instructions it skipped for both schemes. It is clear that the speculative-aware subordinate thread executed much fewer instructions than the

speculative-unaware subordinate thread. The second and fourth bars show the number of instructions the main thread executed versus the ones it skipped for the two SST schemes. On average, the main thread executed relatively the same number of instructions for both schemes. We can therefore conclude that the SST scheme with a speculative-aware subordinate thread executes fewer instructions than the SST scheme with a speculative-unaware subordinate thread.

The main thread in SST with a speculative-aware subordinate thread executed slightly fewer instructions (bar 4) than the SST with a speculative-unaware subordinate thread (bar 2). The speculative-unaware subordinate thread is slower than the speculative-aware subordinate thread because it executes more instructions, and so it is not as effective as the speculative-aware subordinate thread in hiding the branch miss-prediction latency. Hence, it produces slightly fewer results that can be consumed by the main thread without executing their corresponding instructions. This is true for most of the benchmarks, with the exception of benchmark *vortex*.

Chapter 6

HSST: Hierarchical Symbiotic Subordinate Threading

Subordinate threads that execute fewer instructions may advance very rapidly, but maybe highly speculative and go along wrong paths quite frequently as well as produce incorrect results. On the other hand, subordinate threads that execute more instructions may produce many more correct results, but may not be fast enough to hide the latencies of long-latency instructions. Therefore, we investigate a scheme that utilizes both the speed of highly speculative subordinate threads and the confidence of not-too-speculative subordinate threads.

In this chapter we propose *hierarchical symbiotic subordinate threading (HSST)* to achieve both of these goals, by incorporating a hierarchy of subordinate threads. That is, the main thread along with the subordinate threads form a hierarchy, in terms of their composition as well as forwarding of results. Each subordinate thread contains a subset of the instructions of its *parent thread*, the thread that is immediately above it in the hierarchy. Therefore, it is faster and can explore the future earlier than its parent but is more speculative than its parent. The subordinate thread outcomes are not thrown away; rather they are forwarded to their parents using techniques similar to previously proposed approaches (Chapter 3). With this arrangement, each thread benefits from the threads below it in the hierarchy.

HSST is a light-weight architectural framework that extends the SST scheme to using several subordinate threads instead of a single subordinate thread. It makes use of otherwise idle cores in a CMP to improve the performance of individual threads running on the active cores. It incurs only minor hardware changes. We developed a cycle-accurate multi-core simulator to verify its performance. We evaluated HSST against SST. Our experimental results show that a HSST configuration with two subordinate threads improves the average performance by 16% over SST.

6.1 A Motivating Example

We begin with a motivating example to illustrate the tradeoff between a fast and highly speculative subordinate thread that can stride ahead to explore the future versus a more conservative subordinate thread that does not speculate so often, and so is slow, with a limited run-ahead capability. The example we present is for two SST models, main-subA model and main-subB model. The subordinate threads subA and subB have different degrees of speculation. SubA is less speculative than subB. We will describe the benefit of each with regard to its ability to perform data pre-fetching in the L2 cache.

Consider the main thread code snippet in Figure 6.1a. The code snippets subA and subB are shown in Figure 6.1b and Figure 6.1c, respectively. Assume all three threads begin execution at instruction 0 and iterate only once in the loop. Cache blocks of instructions 3 and 8 still have 20 cycles to arrive at the L1 dcaches in all three threads. SubA and subB execute a subset of the instructions executed by

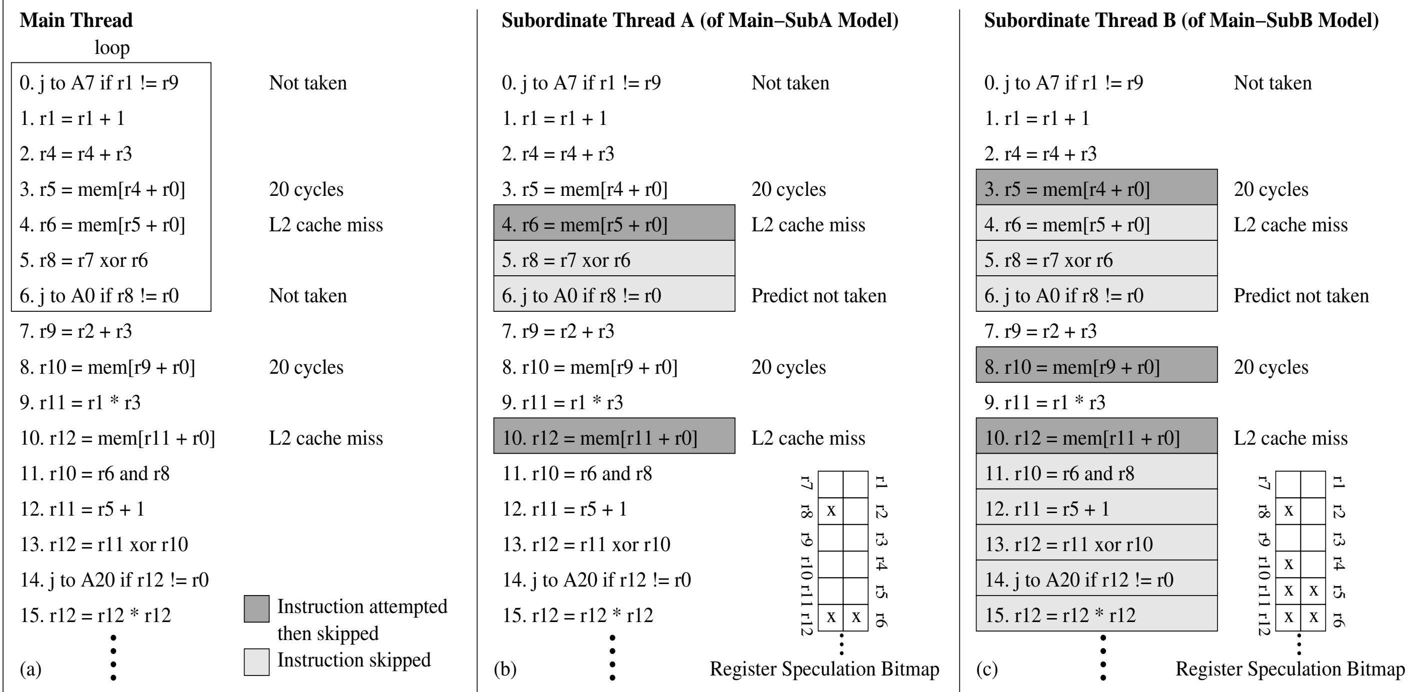

Figure 6.1: Example from benchmark *perl* showing the code snippet for: (a) Main thread; (b) Subordinate thread of main-subA model; and (c) Subordinate thread of main-subB model.

the main thread. Instructions skipped by the individual threads are highlighted in grey. Both subordinate threads skip long-latency instructions that block the pipeline and their dependency chains, similar to that in run-ahead execution [19]. Dependency chains of skipped instructions are identified by marking the output registers of skipped instructions as data-speculative in the RSB. SubA identifies an instruction to be a long-latency instruction if it reaches the head of the ROB and stalls the pipeline for 20 cycles. SubB will wait for 10 cycles only. Both subordinate threads may skip branch instructions, and follow their predicted outcomes. SubA attempts to execute instructions 4 and 10, and then removes them out of the pipeline because they miss in the L2 cache. It skips 5 and 6 because they are dependent on 4. SubB skips all instructions except 0, 1, 2, 7, and 9. It attempts to execute instructions 3, 8, and 10, and then skips them because it concludes that they are long-latency instructions. It skips the remaining instructions because they are dependent on instructions 3, 8, and 10.

Comparison between SubA and SubB: SubB executes only a subset of the instructions executed by subA, and so it runs faster and can explore the future earlier than subA. However, subB corrupts its state (register file and memory) much faster than subA, and so it may not find any independent instructions to execute in the future. The number of bits marked as data-speculative in subA's RSB is half those marked in subB's RSB, as shown in Figure 6.1b and Figure 6.1c, respectively. Therefore, the correct results produced by subB are fewer than those produced by subA. Also, subB is more likely to go on the wrong path than subA, as it skips more

High Versus Low Speculation Subordinate Threads

High Speculation Pros:

–Faster (executes fewer instructions)
–Larger instruction window

Low Speculation Cons:

–Slower (executes more instructions)
–Smaller instruction window

High Speculation Cons:

–Corrupts its state faster
–Goes on the wrong path more often
–Aggressive, misses opportunities
–Produces fewer correct results
–Huge speed gap with the main thread

Low Speculation Pros:

–State is sllightly corrupted
–Goes on the wrong path less often
–Less aggressive, exploits opportunities
–Produces a lot of correct results
–Slightly faster than the main thread

Figure 6.2: Pros and cons of high and low speculation subordinate threads.

branch instructions. Note that because subB is aggressive in skipping instructions, it did not attempt to execute instruction 4 although it is an L2 cache miss. On the other hand, subA's state is less corrupt, but it had to block longer than subA, waiting for instructions 3 and 8 to complete, before reaching the L2 cache miss instructions (4 and 10). Finally, the speed gap between the main thread and subB is much larger than that between the main thread and subA. This implies that subA will spend less time on the wrong path than subB and is less likely to throw pages out of the L2 cache that are needed by the main thread, than subB. We summarize the pros and cons of highly speculative subordinate threads such as subB as well as low-speculative subordinate threads such as subA in Figure 6.2. HSST exploits the advantages of both types of subordinate threads while avoiding their disadvantages.

6.2 Implementation Details of HSST

We concluded the previous section by presenting the trade-offs of a fast and more speculative subordinate thread that can run ahead to explore the future versus a slow and less speculative subordinate thread with limited run-ahead capability. The basic idea of combining the advantages of both schemes and eliminating the disadvantages is to organize multiple subordinate threads as a cache-like hierarchy. Each subordinate thread in the hierarchy is a subset of its *parent thread* (main or subordinate). Therefore, it is more speculative and faster than its parent, and can run ahead of its parent (Figure 6.3a).

We next present an implementation of HSST with a main thread and multiple subordinate threads on a chip-multiprocessing (CMP) platform [37]. In addition to multiple sequencers, a CMP processor has multiple pipelines for processing multiple threads in parallel. Figure 6.3b shows a high level design of the CMP that supports HSST. The top of the hierarchy is the main thread, followed by subA, then subB, and so on. The L2 cache is shared among all threads and is updated only by the main thread. A subordinate thread's L1 dcache can be corrupt because it is speculative, and so it is not allowed to write to the shared L2 cache.

There are several concerns for supporting HSST on a CMP platform. We address those concerns with minimum additional hardware. The first concern is spawning subordinate threads on idle cores. This is handled by the *thread controller (TC)*. The second concern is distilling each subordinate thread such that it executes only a subset of the instructions executed by its parent thread. Our subordinate

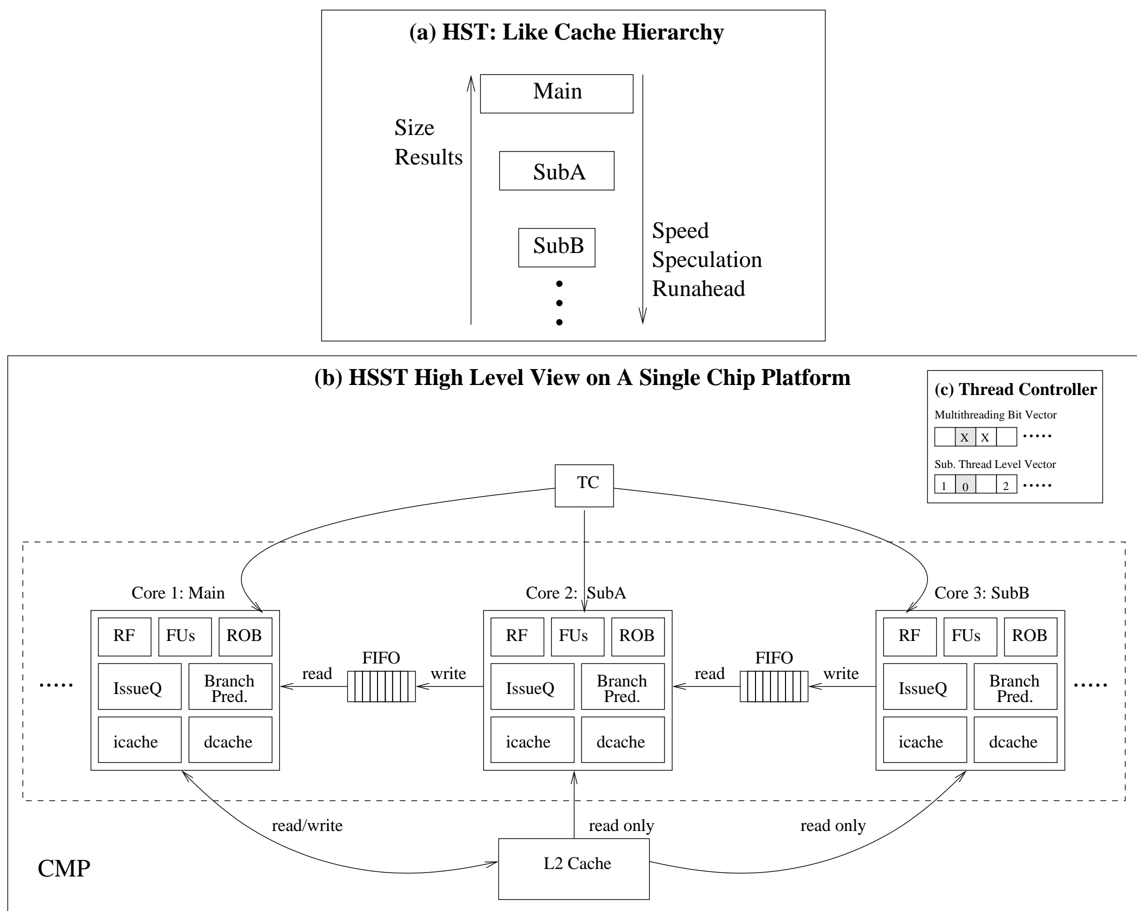

Figure 6.3: HSST High Level Microarchitecture: (a) HSST similar to a cache hierarchy; (b) HSST block diagram; and (c) Components of Thread Controller (TC).

thread is pruned dynamically, which requires support at the core level. Figure 6.4 shows the detailed microarchitecture design of HSST. Each core is modified to support HSST. The HSST core maintains three simple bitmaps, the RSB, the MSB and the *level vector (LV)* for pruning the subordinate thread. In the case of partially speculative-aware subordinate threads, each core must include an extra RSB and an extra MSB to be able to identify its child's results that are non-data-speculative. Third, results generated by a subordinate thread are forwarded to its parent thread via a *first-in-first-out (FIFO) queue* that connects each subordinate thread to its parent thread. Note that in Figure 6.4 we show only two subordinate threads, subA and subB in addition to the main thread although, HSST can support any number of subordinate threads. SubA follows the main thread in the hierarchy, followed by subB.

6.2.1 Spawning Subordinate Threads

Our HSST implementation is purely at the hardware level and maintains the flexibility to support multithreaded applications. The subordinate thread mode is used only when there are free cores. This is determined by the TC. The TC maintains a bit for every core that indicates if that core is idle or is busy (running a thread from a multithreaded application). Those bits are stored in the multithreaded vector, as illustrated in Figure 6.3c. When there are free cores, the TC may spawn subordinate threads on them. The TC assigns a level to each spawned subordinate thread and stores the level associated with each subordinate thread in the level

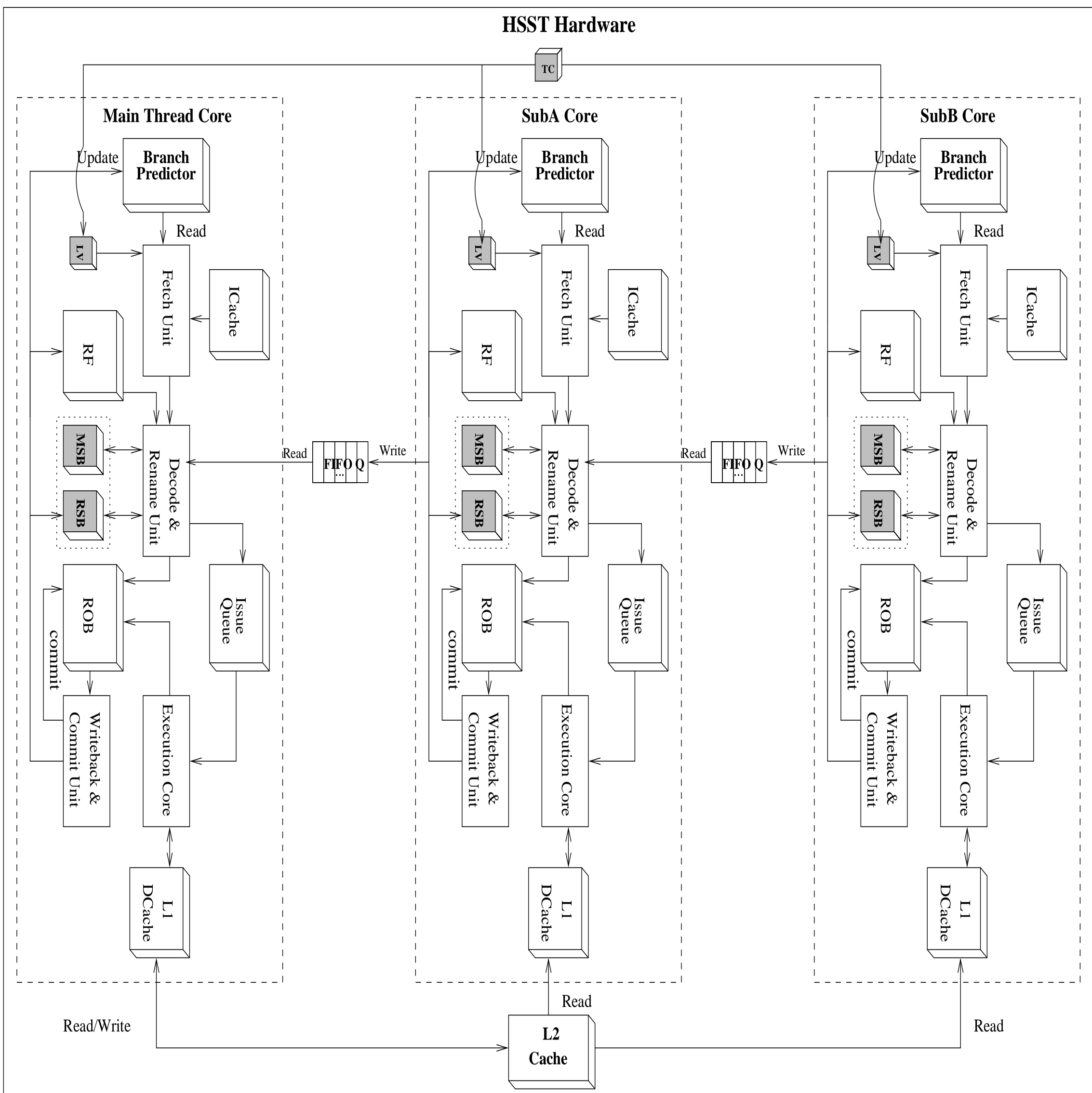

Figure 6.4: HSST detailed microarchitecture design.

vector. In Figure 6.3c, there are two free cores, and so the TC spawned a subordinate thread on each free core for helping one of the running threads (the shaded box). Each spawned subordinate thread is assigned a level that is stored in the level vector. The main thread is assigned level 0 (shaded box). The TC assigns levels in ascending order. Every time the TC assigns a subordinate thread to a free core, it sets the appropriate bit in the core's level vector (LV). The core needs to store the subordinate thread level, as it is required for dynamically pruning the subordinate thread, as will be seen next.

Note that each core has two modes, a subordinate thread mode and a regular mode. The core realizes it is in the subordinate thread mode if its LV is not empty. Our implementation so far is for having subordinate threads that help only a single main thread. We can extend it to several main threads each having its own subordinate threads. In this case the TC will have to maintain a level vector for each main thread to store in it which cores run subordinate threads for that particular main thread and at what level each subordinate thread is running.

6.2.2 Distilling the Subordinate Thread

Each subordinate thread is distilled dynamically in order to be able to run ahead of its parent thread. The level bit set in the core running the subordinate thread determines the aggressiveness of the subordinate thread in skipping instructions. A level one subordinate thread is the least aggressive, as it is the level right below the main thread. A level 2 is more aggressive, and so on. A subordinate thread

may skip past long-latency instructions as well as highly predictable branches and their backward slices. It may also skip the dependency chains of the skipped instructions, and so it maintains information about its register file state and its memory state in order to be able to identify the dependency chains of skipped instructions. It maintains this information using the MSB and the RSB.

Long-Latency Instructions: A subordinate thread decides that a memory instruction is a long-latency instruction if it arrives at the ROB head and stalls the pipeline for a number of cycles determined by the subordinate thread level. For example, the first level subordinate thread may use n cycles. The second level subordinate thread may use $n/2$ cycles, the third level may use $n/4$ cycles, and so on. The subordinate thread will toss the long-latency instruction out of the pipeline and free all its resources.

Highly Predictable Branches: Highly predictable branches can be determined from the saturating counters stored in the branch predictor for every branch instruction. A branch instruction is considered to be highly predictable if its saturating counter reaches a certain threshold determined by the subordinate thread level. A lower threshold of the saturating counter is required as we go down the hierarchy. The subordinate thread may toss out instructions that form the backward slice of the branch if they are still in the pipeline. Note that we give priority to memory instructions. If a memory instruction forms a backward slice of a highly predictable branch, it is not tossed out of the pipeline until it arrives at the ROB head and

attempts the memory access. This is to ensure that if it was an L2 miss then it is at least attempted.

Dependency Chains of Skipped Instructions: Each subordinate thread maintains an RSB to help it filter out instructions that attempt to use speculative register values. Also, each subordinate thread maintains an MSB to help it in identifying LOAD instructions that read data-speculative values and hence can be skipped by the subordinate thread.

6.2.3 Result Integration

Each child thread forwards all its instructions and results to its parent thread and marks the non-data-speculative ones. The child thread retires its results at the tail of the FIFO buffer that connects it to its parent. The parent thread reads at the head of the FIFO buffer all forwarded results. The parent consumes the non-data-speculative ones without executing their corresponding instructions. The parent thread also does not fetch and decode instructions because it receives them from its child thread. Note that the parent thread cannot proceed with executing any instruction unless its child thread has already placed it on the FIFO queue. This is to ensure that the parent thread always follows its child thread and that they are both synchronized.

6.2.4 Recovering the Subordinate Thread Corrupted State

When the subordinate thread state is corrupt or when it goes on a wrong path, it becomes useless and the best thing is to restart the subordinate thread and recover its state. Recovering the subordinate thread state requires copying the more accurate state of its parent. The subordinate thread may copy the register file of its parent and invalidate its L1 dcache. If its parent thread is a subordinate thread, then it needs also to copy the RSB and MSB of its parent. When a parent thread detects that its child thread had gone on a wrong path, it announces to the TC its need to recover its children and the TC restarts all threads below it. All restarted subordinate threads will need to copy the state of the parent thread that initiated the recovery.

6.3 Experimental Results

In order to evaluate our proposed HSST scheme, we extended our SST simulator to support multiple subordinate threads that are arranged in a hierarchy. Our simulator faithfully models an HSST system running on a multi-core CMP, with a main thread, two subordinate threads, and their interconnections, as per the block diagram of Figure 6.4. The microarchitectural parameters we used are shown in Table 6.1. The L1 dcache of a subordinate thread is invalidated on its recovery from the wrong paths. All cores use a single branch predictor, which is updated only by the main thread.

Single Core Parameters	
L1 ICache	sz/assoc/repl/ln/lat=16KB/1way/LRU/64B/1cycle
L1 DCache	sz/assoc/repl/ln/lat=64KB/4way/LRU/64B/1cycle
L2 Cache (data+instrs.)	sz/assoc/repl/ln/lat=1024KB/8way/LRU/128B/6cycles
Main Memory Latency	18 cycles (results presented for 100 cycles as well)
Fetch/issue/retire	bandwidth = 4/4/4
ROB/LdStQ/FetchQ	size = 64/32/16 entries
Branch Predictor	type = bimodal, size = 32K entries, penalty 7 cycles+
HSST-Specific Parameters	
MSB	64 bits
FIFO Queue	latency/bandwidth/sz = 2 cycles/5 instrs./32 instrs.
Branch Threshold	conf. count for subA/subB/subC/subD = 16/8/5/4
Memory Threshold	wait cycles for subA/subB/subC/subD = 50/25/16/12
Sub. Thread Recovery	20 cycles

Table 6.1: HSST Microarchitectural Parameters

We evaluated our HSST scheme against two SST schemes, main-subA scheme and main-subB scheme, each having a main thread and a single subordinate thread. In the main-subA scheme, the subordinate thread is subA and in the main-subB scheme, it is subB. We let HSST use two subordinate threads, subA and subB. SubA follows the main thread in the hierarchy and subB is at the lowest level of the hierarchy. We achieved significant performance improvement with HSST against the SST schemes. HSST may use any number of subordinate threads. We tried more

Superscalar-Specific Parameters	
Fetch/issue/retire	bandwidth = 16/16/16
ROB/LdStQ/FetchQ	size = 256/128/64 entries
Branch Penalty	7 cycles (results presented for 16 cycles as well)
Main Memory Latency	18 cycles (results presented for 100 cycles as well)

Table 6.2: Superscalar Microarchitectural Parameters

than two subordinate threads as well and did not see significant further performance improvement. The latency thresholds for skipping memory and branch instructions in each subordinate thread are shown in Table 6.1. Our comparisons are against a base line superscalar processor with three times the issue width of a single core and three times its capacity for in-flight instructions.

6.3.1 Performance Improvement

Figure 6.5 presents the IPC obtained with result integration enabled for four schemes: single thread base scheme, main-subA scheme, main-subB scheme and HSST. There are four bars per benchmark, corresponding to each scheme. The single thread scheme is a superscalar processor whose microarchitectural parameters are shown in Table 6.2. For all four schemes we use a branch penalty of 7 cycles and a memory latency of 18 cycles. For the SST schemes and the HSST scheme, the subordinate thread incurs minimum penalty on recovery because we used a shadow register file as in Chapter 3. Also, the branch threshold for highly predictable

branches for subA is 2 and for subB is 1. We also present another set of results for the same four schemes in Figure 6.6, but we let the memory latency be 100 cycles, the branch miss-prediction penalty be 16 cycles, the subordinate thread recovery penalty be 20 cycles, the branch threshold for subA be 16, the branch threshold for subB be 8, the subordinate thread be speculative-aware and only skip branch instructions.

It is clear from Figures 6.5 and 6.6 that HSST performs better than main-subA and main-subB for all the benchmarks. The average performance improvement of the three symbiotic schemes over the single thread base scheme is shown on top of the average bars, 38% for HSST, 22% for main-subA, and 19% for main-subB in Figure 6.5 and in the same order in Figure 6.6, 21%, 13% and 14%. The first set of results shown in Figure 6.5 indicate that with a much smaller pipeline per core, a more accurate branch predictor and smaller memory latencies, the HSST scheme performs better, than in a much larger pipeline with long memory latencies and a less accurate branch predictor. This implies that our scheme exploits parallelism when it is available better than the superscalar processor with 3 times the issue width and 3 times the size of each core. The performance improvement of HSST is due to result integration, which hides the branch penalties and pre-fetching the L2 cache misses, as will be seen in the following subsections.

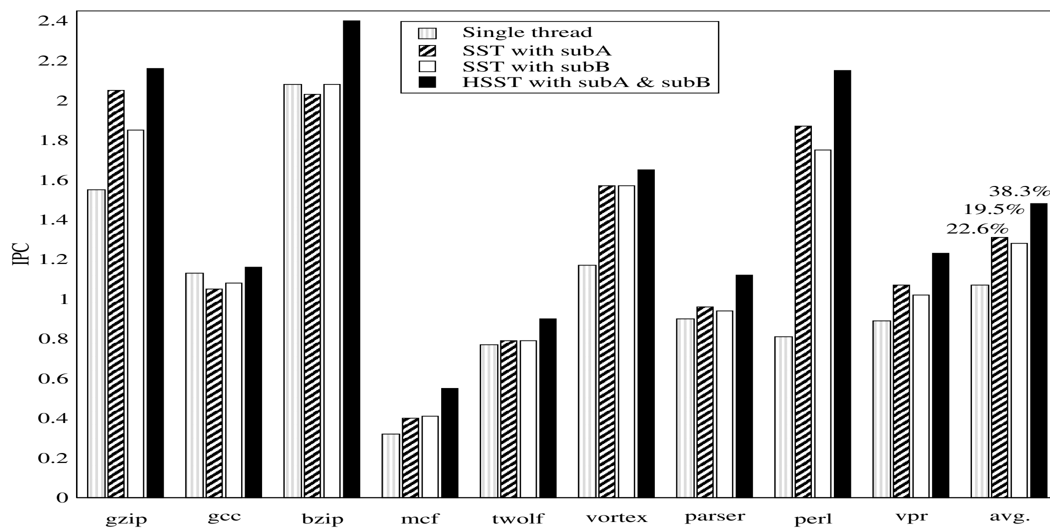

Figure 6.5: IPC obtained for four schemes: (a) Single thread (superscalar); (b) SST with subA (main-subA); (c) SST with subB (main-subB); and (d) HSST with both subA and subB.

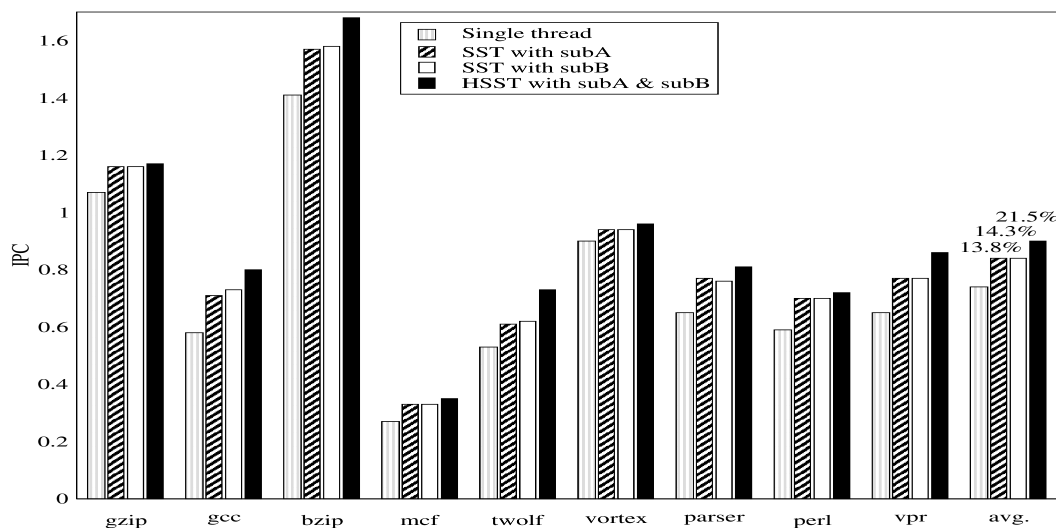

Figure 6.6: IPC obtained for four schemes: (a) Single thread (superscalar); (b) SST with subA (main-subA); (c) SST with subB (main-subB); and (d) HSST with both subA and subB.

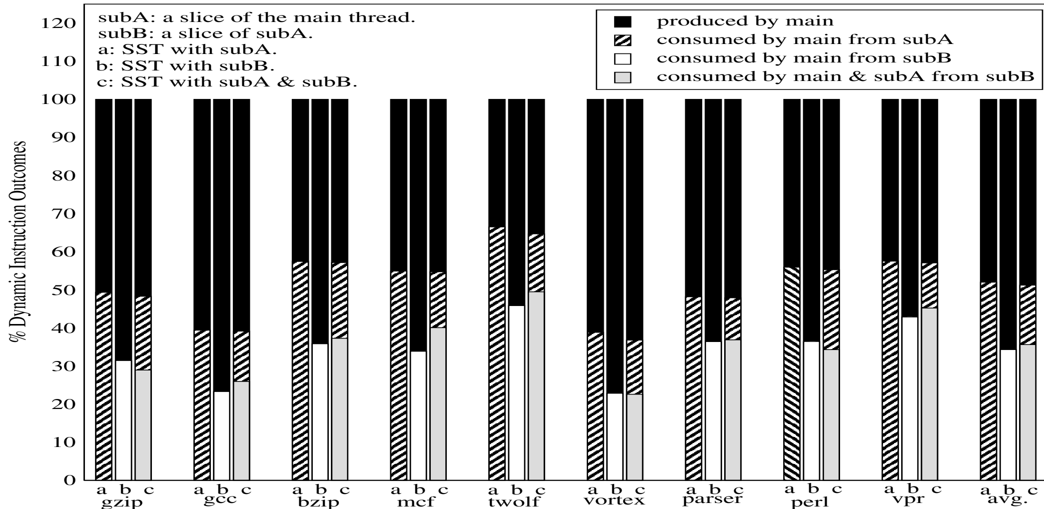

Figure 6.7: Distribution of instruction outcomes in main thread for three schemes: (a) SST with subA (main-subA); (b) SST with subB (main-subB); and (c) HSST with subA and subB.

6.3.2 Advantages of Result Integration

In HSST, results correctly produced by a child thread are consumed by its parent thread without executing their corresponding instructions. This exploits the available parallelism and hides the branch miss-prediction latencies, in addition to reducing the redundant and useless computations done by each thread.

Exploiting Parallelism: Result integration allows independent computations done by a child thread to overlap with other independent computations done by its parent thread. This speeds up the parent thread, which in turn will speed up its own parent and so on until the highest level in the hierarchy (main thread). We looked at the distribution of instruction outcomes in the main thread for the three schemes (main-subA, main-subB, and HSST). This data is presented in Figure 6.7. The

three bars show the distribution of instruction outcomes in the main thread for main-subA, main-subB, and HSST, respectively. In all three bars, the distribution of instruction outcomes shows the percentage of outcomes produced by the main thread (instructions executed by the main thread) and the percentage of instruction results consumed by the main thread from the subordinate threads without re-executing them.

In HSST the main thread executed roughly the same number of instructions as in the main-subA scheme (first bar), and yet the IPC of HSST is higher than that of main-subA (Figures 6.5 and 6.6). The IPC is governed by the main thread, and so, it must have gained speed. The reason for this speed is that the main thread received the outcomes of subA sooner. That means that subA must have gained speed. We know that a considerable amount of the outcomes that subA was supposed to produce were actually produced in parallel by subB. Therefore, subA did not have to execute all the instructions that it was suppose to execute. Rather, it consumed the outcomes of a big percentage of those instructions from subB and passed them along with the outcomes it produced to the main thread.

Efficient Execution: Result integration reduces the number of instructions each thread has to do, by eliminating redundant computations. Redundant computations are computations that a parent thread need not do because its child thread has already done them. This allows each thread to use the available resources more efficiently. In other words, result integration makes efficient use of the small hardware structures available in each core while delivering high speed.

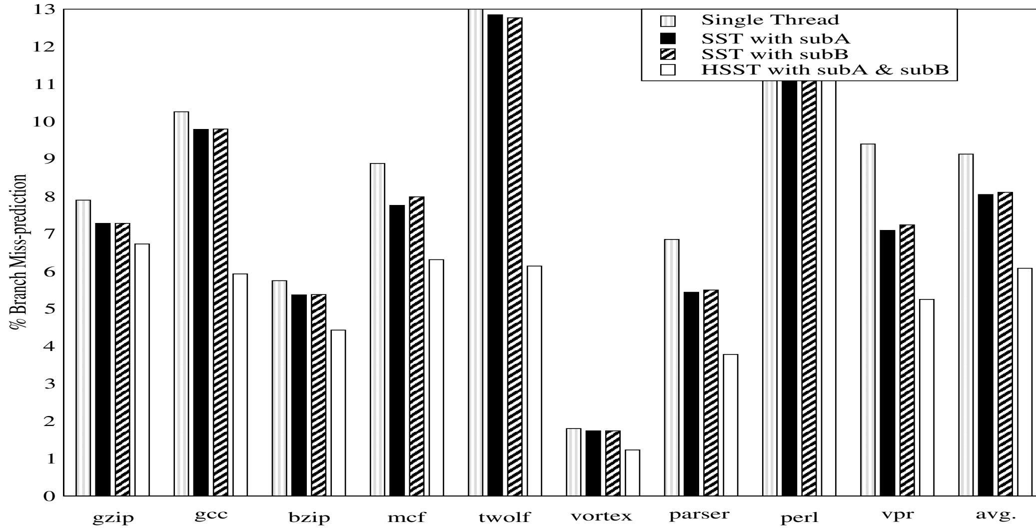

Figure 6.8: Average branch miss-predictions in main thread for four schemes: (a) Single thread (superscalar); (b) SST with subA (main-subA); (c) SST with subB (main-subB); and (d) HSST with both subA and subB.

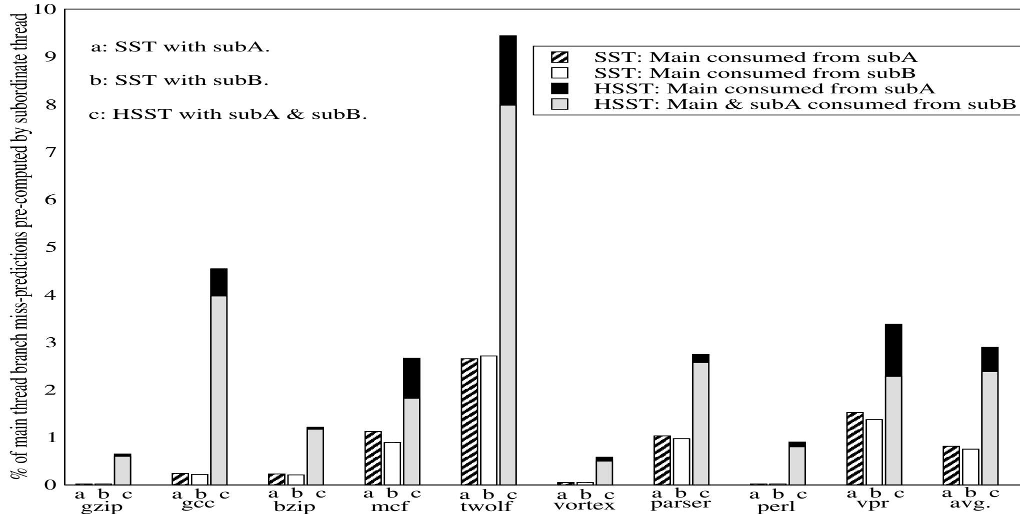

Figure 6.9: Percentage of branch instructions that were a miss-prediction and the main thread obtained their correct outcomes from the subordinate thread, for three schemes: (a) SST with subA (main-subA); (b) SST with subB (main-subB); and (c) HSST with both subA and subB.

Hiding Branch Penalty: The miss-prediction penalty of a miss-predicted branch instruction can be incurred by several threads up the hierarchy. With result integration, all threads above the thread that first incurred the penalty do not have to incur it if that thread executed the branch non-speculatively. In HSST, subA consumes the non-speculative results of subB without executing their corresponding instructions. Those results include branch instructions that may have been miss-predicted but were pre-executed by subB, thereby hiding their miss-prediction penalty. Moreover, the main thread consumes branch outcomes that subA consumed from subB, in addition to the branch outcomes calculated by subA. We plotted the percentage of times the main thread went on the wrong path for the following four models: single thread (superscalar), main-subA, main-subB, and HSST (Figure 6.8). The main thread in HSST incurred fewer branch miss-predictions (third bar) than the main-subA scheme (first bar) and the main-subB scheme (second bar). The advantage of HSST over the SST schemes with regard to the number of branch miss-predictions can be explained by the amount of correct branch results subA obtained from subB. We plot in Figure 6.9 the average number of branch outcomes that are miss-predicted in the main thread, but the subordinate thread was able to hide their miss-prediction penalty, for the three schemes, main-subA, main-subB, and HSST. In HSST subB was able to pre-execute for subA a considerable amount of its miss-predicted branches (grey portion of the third bar in Figure 6.9). Further, subA in turn forwarded all its results along with subB's results to the main thread. Note that because subA was sped up by subB, subA was able to hide even more branch penalties, thereby benefiting the HSST main thread.

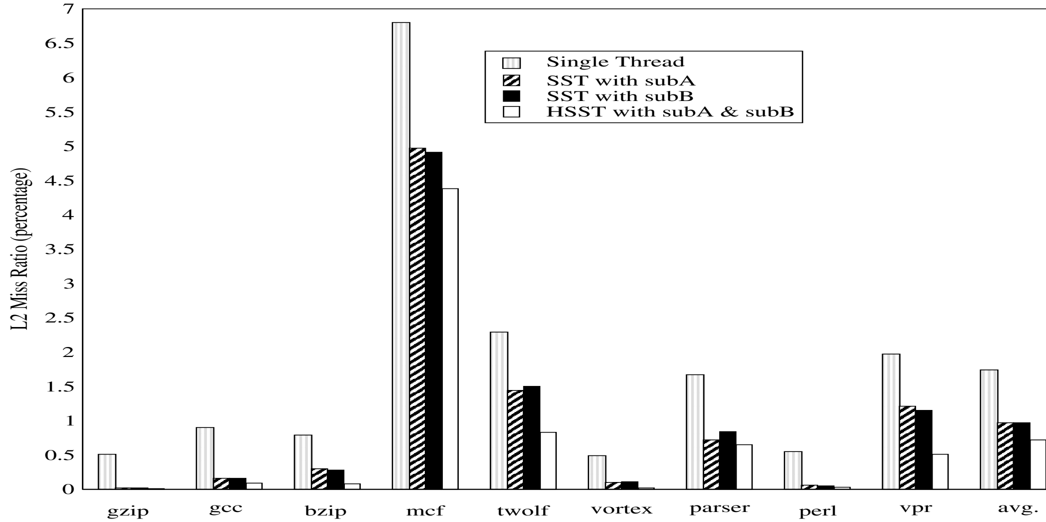

Figure 6.10: L2 cache miss ratio in main thread for four schemes: (a) Single thread; (b) SST with subA (main-subA); (c) SST with subB (main-subB); and (d) HSST with both subA and subB.

6.3.3 Improvement in L2 Cache Miss Ratio

Figure 6.10 shows the main thread L2 cache miss ratio for four schemes, single thread (superscalar) scheme, main-subA, main-subB, and HSST. On average, the L2 cache miss ratio reduced further with HSST (fourth bar). This result implies that in HSST, subB is helping subA with the L2 cache misses. This in turn speeds up subA, which is also going to bring in some blocks into the L2 cache before the main thread needs them. These results imply that more effective L2 cache pre-fetching is occurring in HSST than in the main-subA and the main-subB models.

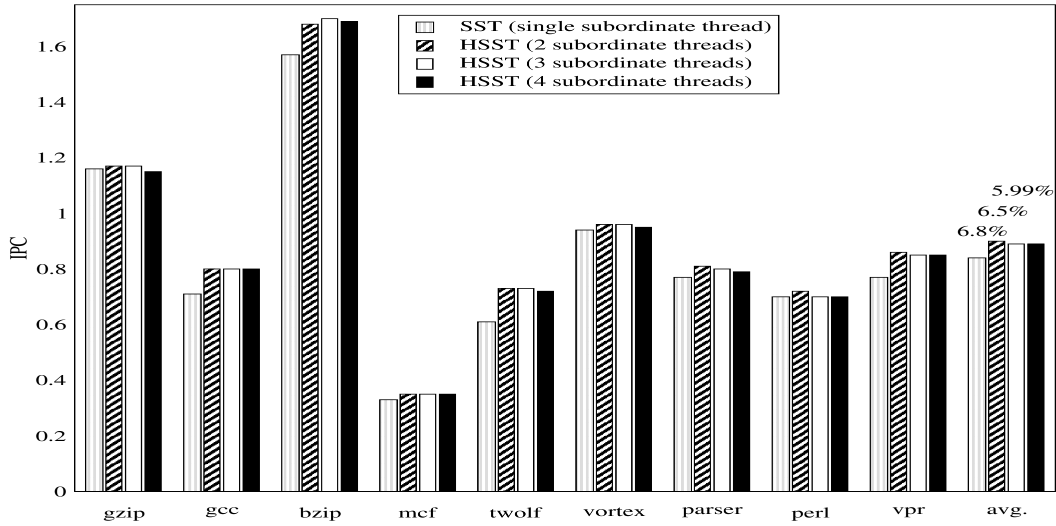

Figure 6.11: IPC obtained for four schemes: (a) SST with a single subordinate thread; (b) HSST with two subordinate threads; (c) HSST with three subordinate threads; and (d) HSST with four subordinate threads.

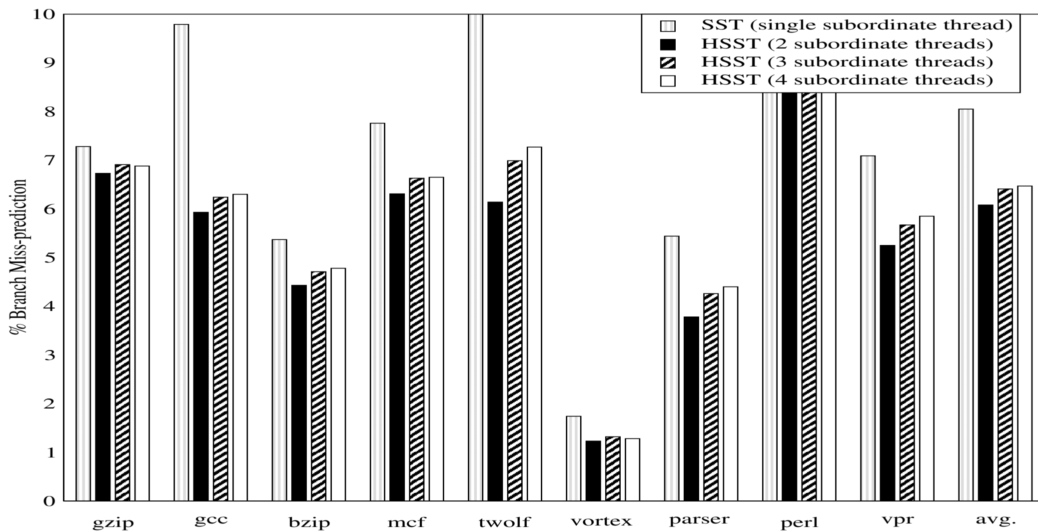

Figure 6.12: Average branch miss-predictions in main thread for four schemes: (a) SST with a single subordinate thread; (b) HSST with two subordinate threads; (c) HSST with three subordinate threads; and (d) HSST with four subordinate threads.

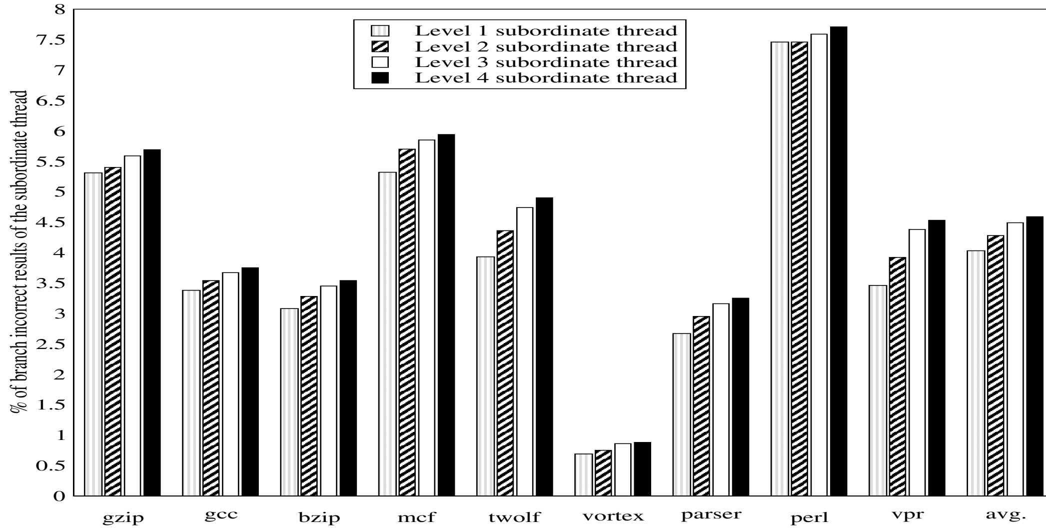

Figure 6.13: Average incorrect branch results of four subordinate threads with different levels of speculation: (a) Subordinate thread at speculation level 1 (subA); (b) Subordinate thread at speculation level 2 (subB); (c) Subordinate thread at speculation level 3 (subC); and (d) Subordinate thread at speculation level 4 (subD).

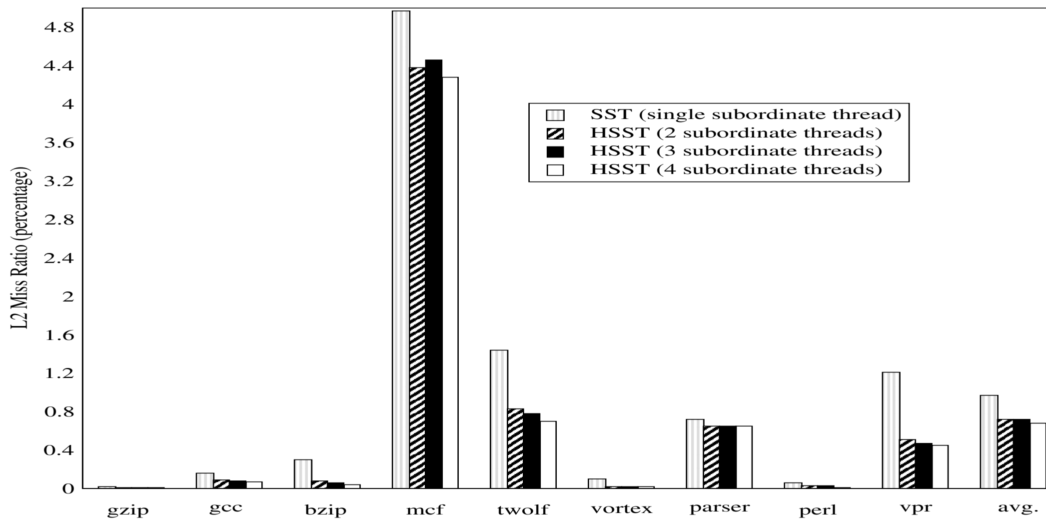

Figure 6.14: L2 cache miss ratio in main thread for four schemes: (a) SST with a single subordinate thread; (b) HSST with two subordinate threads; (c) HSST with three subordinate threads; and (d) HSST with four subordinate threads.

6.3.4 Experimenting with More than Two Subordinate Threads

We performed another experiment in which we used more levels of speculations in the hierarchy, i.e., more subordinate threads in order to evaluate the effect of adding more subordinate threads. We evaluated three subordinate threads and four subordinate threads. Level 1 subordinate thread (subA) is the subordinate thread right below the main thread, followed by level 2 subordinate thread (subB), then level 3 subordinate thread (subC), and finally level 4 subordinate thread (subD). Level 4 subordinate thread is the most speculative subordinate thread, its branch threshold is 4 and its memory threshold is 12 cycles while level 1 subordinate thread is the least speculative with a branch threshold of 16 and a memory threshold of 50 cycles. The thresholds of each level are shown in Table 6.1.

The IPC we obtained for four schemes (SST with a single subordinate thread, HSST with two subordinate threads, HSST with three subordinate threads, and HSST with four subordinate threads) is shown in Figure 6.11. The single subordinate thread we used with SST is the first subordinate thread right after the main thread in all the three HSST schemes. The IPC across all the benchmarks does not show any improvement beyond that obtained with two subordinate threads. With three and four subordinate threads (3rd and 4th bars), the IPC either does not change or decreases. We obtained more statistics to understand the lackluster performance shown when increasing the number of subordinate threads beyond two.

We plot the main thread average branch miss-prediction for the same four schemes in Figure 6.12. The main thread incurs the most number of branch miss-

predictions in the SST scheme (first bar). It incurs the least number of branch miss-predictions when we use two subordinate threads (second bar). When we use two and three subordinate threads, the number of branch miss-predictions increases (third and fourth bars, respectively). This is because the subordinate thread at the bottom of the hierarchy becomes more and more speculative and hence goes more on the wrong path. Its parent thread, therefore, has to squash it more often and cannot proceed until its child is recovered. This slows down both the child and the parent, making both threads ineffective in hiding the branch miss-prediction latency. We show in Figure 6.13 the number of times each subordinate thread was squashed because it went on the wrong path. The number of times a subordinate thread gets squashed increases as we go from the first bar (least speculative subordinate thread) to the fourth bar (most speculative subordinate thread).

Finally, Figure 6.14 shows the L2 cache miss rate of the main thread for the four. There is no significant decrease in the L2 cache miss rate shown in the last two bars (HSST with 3 and 4 subordinate threads). However, there is a significant decrease in the L2 cache miss rate in the second bar (HSST with two subordinate threads) compared to the SST scheme (first bar). From our discussion, we conclude that HSST may not provide significant performance improvement beyond two subordinate threads due to the increased penalties of squashing the highly speculative subordinate threads that offset the insignificant decrease in the main thread L2 cache miss rate.

Chapter 7

Related Work

Although our SST work originated mainly from subordinate threading, the end product does have some similarities with previously proposed ideas. We shall compare and contrast our ideas to these existing ideas.

7.1 SST and Run-ahead execution

In run-ahead execution [11, 19] only a single thread is running at any time. When the main thread is unable to make progress because the instruction window is blocked due to a long latency cache miss, the state of the processor is checkpointed and it switches to the run-ahead mode. In the run-ahead mode, the blocking instruction is removed from the window by supplying it with an invalid value. In this way, the processor can continue to fetch, execute, and pseudo retire instructions, without updating the architectural state. Instructions that follow, if dependent on the blocking instruction, are also removed from the window. When the blocking instruction completes, the processor returns to the 'normal' mode and restores the checkpointed state. All instructions executed in the run-ahead mode will be fetched and executed again during normal mode. The run-ahead mode uses the same hardware context as the main thread with some extra hardware. The benefit of run-ahead comes from letting the processor fetch and execute more instructions

than the instruction window normally permits with the hope of reaching subsequent long-latency cache misses earlier so as to process them in parallel with the blocking instruction that first initiated the run-ahead mode. However, the cost of transitioning from the run-ahead mode to the normal mode involves a pipeline squash, which is equal to a branch miss-prediction penalty. Early return from the run-ahead mode to the normal mode may hide such latency, but it limits the distance of effective run-ahead execution [19].

We identify several limitations in run-ahead execution, that are overcome in our SST scheme. The main difference with our execution model is that, run-ahead execution uses a single hardware context for both the normal and run-ahead modes, while SST uses a separate hardware context for the main thread and the subordinate thread(s). Hence, all threads in SST run in parallel whereas in run-ahead execution, only a single mode is running at any point of time, either normal mode or speculative run-ahead mode. This allows speculative execution in SST to last much longer than in run-ahead execution, in addition to eliminating checkpointing and mode transition. In run-ahead execution, speculative execution in the run-ahead mode stops once the processor returns to the normal mode even if such speculative execution is on the correct path and generates correct pre-fetch addresses. This affects the aggressiveness of run-ahead execution. Second, each cache miss in a chain of dependent cache misses will cause the run-ahead processor to enter the run-ahead mode. If only a few instructions exist between such misses, the processor will pre-execute the same set of future instructions multiple times [55], thereby wasting processor resources. SST eliminates all these limitations seamlessly and

achieves higher performance. Third, in SST the subordinate thread exploits more opportunities to have a wider window and to run faster, skipping highly predictable branches and their backward slices in addition to processing the blocking instructions in the same manner as run-ahead execution. Fourth, in SST, the main thread monitors the subordinate thread state to obtain the maximum benefit out of the subordinate thread, but in run-ahead execution this monitoring does not exist, and so once the run-ahead mode deviates from the wrong path or corrupts its state, it becomes useless. Finally, the use of re-using results in run-ahead [20] execution is limited, because the run-ahead mode runs a very short time with correct values and then it corrupts its state or deviates away from the correct path, thereby producing useless incorrect outcomes. On the other hand, because the main thread monitors the subordinate thread path in SST, it is able to make it run more effectively on the correct path with correct state, thereby contributing more correct results to the main thread.

In continual flow pipelines (CFP) [56], long latency instructions such as cache misses and their dependent instructions (called slice instructions) are drained out of the issue queue and register file by using invalid values as fetched data, similar to run-ahead execution. Unlike run-ahead execution, the slice instructions are not thrown out of the pipeline; rather they are stored in a slice processing unit and the subsequent independent instructions continue their execution speculatively. When the blocking cache miss completes, the slice instructions re-enter the execution pipeline and commit the speculative results. In this way, the work during run-ahead execution is not discarded and there is no need to re-fetch and re-execute those in-

structions. To maintain such speculative data, however, CFP requires coarse-grain retirement and a large centralized load/store queue (a hierarchical store queue is proposed to reduce its latency criticality [57, 56] and a new improvement is proposed in [58]). Compared to CFP, SST eliminates such large centralized structures and builds upon much simpler processor cores (e.g., smaller register files). The fast branch resolution at the subordinate thread (due to its simpler, shallower pipeline) reduces the cost of most branch miss-predictions. Since SST does not need any centralized rename-map-table checkpoints, it also eliminates the complexity for estimating branch prediction confidence and creating checkpoints only for low-confidence branches, as needed in CFP.

7.2 SST and Leader/Follower Architectures

In this section we compare our model against subordinate threading architectures that exhibit the leader/follower aspect of our model. In SST, the subordinate thread (leader) and the main thread (follower) together with the FIFO communication queue form a very large instruction window for single-thread out-of-order execution. Coupling two (or more) relatively simple processors to form a large instruction window for out-of-order processing was originated in multiscalar processors [61], and SST provides a complexity-effective way to construct such a window while eliminating elaborate inter-thread (or inter-task) register/memory communication.

Decoupled Architectures: Running a program on two processors, one leading and the other following, finds its roots in decoupled architectures [21, 68], which par-

tition the program into two partitions — a memory access partition and an execute partition — each of which executes in parallel. The Execute Processor performs all computations and the Access Processor performs all accesses to the data memory. The access processor performs the data fetch ahead of demand by the execute processor, thereby hiding the memory access latency. The primary difference with our processor is that decoupling is part of the instruction set architecture (requiring more sophisticated compilation), whereas our execution model is purely at the microarchitecture level. Also, our scheme does not classify instructions as memory access instructions and execute instructions. In SST, the subordinate thread(s) not only pre-fetches the data but also provides a highly accurate instruction stream by fixing branch miss-predictions for the main thread. Moreover, all this is accomplished without the difficult task of partitioning the program.

Slipstream processors: Slipstream processors [15, 18] are leader/follower architectures proposed to accelerate sequential programs. They are similar to SST and share a similar high-level architecture: two processors connected through a FIFO communication buffer. However, SST and slipstream processors achieve their performance improvements in quite different ways. In slipstream processors, the A-stream runs a shorter program based on the removal of ineffectual instructions while the R-stream uses the A-stream results as predictions to make faster progress. The A-stream is a relatively slower leader since long latency cache misses still block its pipeline unless they are detected ineffectual and removed from the A-stream. The R-stream is a relatively slower follower as well, because it must execute every

instruction executed by the A-stream even if the A-stream executed it correctly. On the other hand, in SST the main thread consumes the subordinate thread non-speculative results without executing their corresponding instructions (instead of using them as predictions). This allows the main thread to become a faster follower. Also, in SST, the subordinate thread is a much faster leader as it operates on a virtually 'ideal' L2 cache as well as skip highly predictable branches and their backward slices; hence, its effective instruction window is much bigger than that of slipstream.

Dual-core execution model: The dual-core execution paradigm (DCE) [36] is another leader/follower architecture proposed to accelerate sequential programs. It is similar to SST and shares a similar high-level architecture: two processors connected through a FIFO communication buffer. It consists of two superscalar cores, a front processor (leader) and a back processor (follower). The front processor resembles the A-stream and the back processor resembles the R-stream in slipstream terms. The front processor, however, executes all instructions except for long-latency cache misses. For a long-latency cache miss, it instead produces an invalid value instead of blocking the pipeline similar to run-ahead execution. Other than that, everything else in DCE is the same as in slipstream. Our SST model differs from DCE in the same way as it differs from slipstream. Also, the subordinate thread in SST is a faster leader than DCE's front processor, because it not only operates on a virtually 'ideal' L2 cache, but it also skips highly predictable branches and their backward slices. Therefore, its effective instruction window is larger than that of

DCE.

Flea-Flicker Model: "Flea-Flicker" two pass pipelining [41] is closest to SST in terms of integrating run-ahead execution and leader/follower architectures. In the Flea-Flicker design, two pipelines (A-pipe and B-pipe) are introduced and coupled with a queue. The A-pipe executes all instructions without stalling. Instructions with one or more unready source operands skip the A-pipe and are stored in the coupling queue. The B-pipe executes instructions deferred in the A-pipe and incorporates the A-pipe results. Compared to this work, SST is based on out-of-order execution, thereby having higher latency hiding. More importantly, flea-flicker tries to reuse the work of the A-pipe by introducing a lot of complexity overheads (e.g., the centralized memory order bookkeeping and the coupling result store in flea-flicker), while SST uses the simple RSB and MSB bitmaps to identify results that the main thread can reuse from the subordinate thread. The elimination of such centralized structures is the reason why SST is a much more scalable and complexity effective design.

Dual-core speculative multithreading: Srikanth et. al [42] proposed a minimal dual-core speculative multithreading model (SpMT) that achieves significant performance improvement for single-threaded applications. In this model, one core executes the speculative threads (leader), while the other executes non-speculatively (follower). In this scheme, the results of instructions that are executed by the speculative threads and not affected by data dependence violations are buffered and later

committed by the non-speculative thread without re-executing them. The non-speculative thread may spawn speculative threads whenever a spawn point arrives and only one speculative thread may run at any point of time. There are three types of speculative threads; run-ahead speculative thread that is spawned when a cache miss latency is encountered, a procedure thread that is forked when the procedure call is encountered, and a loop speculative thread that is forked when a backward slice of a branch instruction is encountered several times. The main difference with SST is that SST uses a single speculative thread and it has no forking or spawn points.

Master/Slave speculative parallelization: In master/slave speculative parallelization (MSSP) [43, 44], there are two types of threads, a compiler generated single master thread (subordinate thread) and a slave (main thread) that is parallelized into multiple tasks. The slave threads use the outcomes of the master as predictions. The HSST version of SST differs from master-slave in dividing the subordinate thread and not the main thread into several more speculative subordinate threads, and in the hierarchical organization of the subordinate threads.

Pre-execution/Pre-computation architectures: In pre-execution/pre-computation architectures [59, 12, 3, 4, 60, 9], a pre-execution/pre-computation thread is constructed using either hardware or the compiler and leads the main thread to provide timely pre-fetches or computed branch outcomes (for miss-predicted branches). In a multithreaded architecture, however, pre-execution threads and the main thread

compete for a shared instruction window and a cache miss in any thread will block its execution and potentially affect other threads through resource competition. In future execution [45], an otherwise idle core on a chip multiprocessor pre-executes future loop iterations using value prediction to perform cache pre-fetching for the main thread.

7.3 SST and Result Reuse

Deterministically reducing the number of executed instructions by means of Dynamic Instruction Reuse was proposed in [16, 17]. The main idea is to keep copies of recent instruction results (along with their operand values) so that future dynamic instances of the same instruction can use the same value, if they have the same input values as the buffered ones. The key differences with our scheme are that it works with a single thread of control, and buffers instruction results for far longer periods of time. Moreover, our scheme does not compare entire register values as in [16, 17]; rather it uses the RSB to identify registers that are data-speculative from those that are not.

The use of result integration in data-driven multithreading: Speculative data-driven multithreading (DDMT) [4] forks subordinate threads that are decided statically. With the use of a technique called register integration [13], the main thread is able to allow the main thread to directly use results computed in the data driven threads (DDTs). Integration exploits the fact that both the main thread and the DDT place results in a shared physical register file (in an SMT (simultaneous multi-

threading) implementation). Using a modification to register renaming, integration allows the main thread to recognize and claim DDT results. The main way SST differs from DDMT is in the way the main thread integrates results from the subordinate thread. Because of its generality, it can handle subordinate threads with a lot of data and control speculations. Pruning the main thread in SST is based on the architected register specifiers, and so it is independent of register renaming. SST can therefore work on both SMT and CMP (Chip Multiprocessor) platforms. The subordinate thread in SST runs as long as the main thread runs, while a DDT is spawned when needed, and vanishes after it performs its task.

7.4 SST and Clustered Architecutures

In clustered architectures [22, 23, 24], the processor resources are split into two or more clusters. Each cluster is simpler, faster, and consumes less power than a monolithic architecture. Instructions are generally dispatched to clusters based on data dependencies in order to localize dependencies within a cluster and to reduce communication among clusters. The main difference with our execution model is that clustered processors have a single thread of control (and therefore a single fetch unit), whereas SST has two or more threads of control.

Chapter 8

Future Work

Our SST/HSST has some limitations. In this chapter we identify some of those drawbacks and provide insight into our intended solutions. We also discuss some ways of extending our SST work in the future.

8.1 Making the Fastest Thread the Leader

Our SST and HSST schemes are leader-follower architectures because the parent thread never goes ahead of its child thread. The assumption is that, the child thread is more speculative than its parent thread and runs faster, and therefore it is always ahead of its parent thread. However, there are occasions when the parent thread can go ahead of the child thread because each of them generates accesses to different memory locations. For instance, a parent thread and its child thread may access a block of memory that resides in the parent's L1 dcache but is not in the child's L1 dcache and not in the shared L2 cache. In this case the child will block for sometime until it realizes that the corresponding memory instruction is a long latency instruction and retires it early before the block arrives from main memory. In SST and HSST however, we do not allow the parent thread to go ahead of its child thread, and so the parent thread will also block until it receives the speculative result of the memory instruction from its child. If the parent thread has gone ahead

and executed the memory instruction, it would have delivered the result to its own parent much faster.

We intend to explore SST and HSST schemes that do not define the leader to be the child thread and the follower to be the parent thread. Rather, the leader will be the fastest thread (whether parent or child). In other words, instead of holding the parent thread until its child thread forwards its results, we will permit the parent thread to run ahead of its child thread. This has several implications: First, speed will be governed by the fastest thread at any time, i.e, if at some time the parent thread can be faster than the child thread, then it makes sense to make the parent thread the leader. Second, the parent and the child may loose synchronization. Therefore, mechanisms must be provided to ensure that they are re-synchronized. Third, the parent thread must therefore be independent of its child thread, and so it must fetch and decode its own instructions when it goes ahead of its child thread.

8.2 Hybrid HSST Processor

In SST or HSST the subordinate threads are more general and they are all distilled in the same manner but with different levels of speculations. We intend to try specialized subordinate threads such that each is concerned about a different type of critical latency. For instance, the lowest level subordinate thread can be only concerned about pre-fetching the instruction cache for all the other levels as well as fetching and decoding all instructions. This can be done by making it only fetch, decode, and retire all instructions, and so the only latency it would incur is the icache

miss latency. We can use another type of subordinate thread that is concerned only about pre-fetching the L2 cache and skips all branches and their backward slices. We can use a third type of subordinate thread that is concerned only about resolving hard-to-predict branch instructions. We can then put all threads in a hierarchy with the main thread. It would make sense to put the most speculative thread (the one that performs icache pre-fetching) at the bottom of the hierarchy, and on top of it the one that pre-fetches the L2 cache and then the one that performs branch pre-computation and at the top of the hierarchy, the main thread. That would make an HSST scheme with a hybrid collection of subordinate threads. We can take that further by adjusting the level of speculation of each thread based on the running application.

8.3 Exploiting Program Behavior Changes Using Dual Thread Execution Models

One way of utilizing the additional processing cores in a multi-core environment is to run subordinate threads on them so as to speed up the execution of critical instructions. Another option is to spawn speculative threads on them so as to exploit thread level parallelism. We performed a study on our SST scheme which is a subordinate threading scheme and a speculative multithreading technique, the trace processor [69].

In the trace processor, the compiler or hardware partitions a sequential program into *speculative threads*, and the processor executes multiple traces in parallel,

with the help of multiple processing cores. Processing cores are arranged as a circular queue, in which only the head processing core is allowed to commit its instructions. All other processing cores cannot commit instructions until they become the head. A speculative thread is a contiguous sequence of dynamic instructions, called a trace. A trace is spawned before control reaches that trace, and before knowing if its execution is required or not. The use of traces allows aggressive exploitation of thread-level parallelism from programs that are inherently sequential.

Our study shows that some applications benefit more from the SST scheme, and others benefit more from the speculative multi-threading approach (the average performance was slightly higher for the SST approach). More importantly, our results also show that many of the applications cannot be strictly categorized as favoring either speculative multi-threading or decoupled execution as SST. Rather, most applications alternate between the two categories during different phases of their execution. We plan to identify characteristics of code regions that make them more suitable to be run using one approach or the other. We also plan to evaluate the potential for a hybrid processor that can switch execution modes between a trace processor mode (exploiting thread-level parallelism) and SST mode (exploiting decoupled execution).

8.4 Division of Work

Our SST and HSST models try to divide the instructions to be executed among the different threads such that if one thread produces the correct result of a

particular instruction, then all the threads above it in the hierarchy should consume that result without executing its corresponding instruction. In that sense, we are dividing the instructions to be executed among the threads. However, good division of instructions among the threads does not always result in good division of work. For better utilization of the hardware, it is more important to eliminate redundant computations and have a more equal distribution of work among the threads.

We plan to study the division of work among the threads more carefully, such that we can identify more accurately the level of speculation of each thread. An ideal division of work would make each thread in a dual-core perform only 50% of the required work in parallel. If we use a second subordinate thread, then each of the three threads should perform only 33.3% of the work in parallel, and so on. Unfortunately, the world is not that ideal for several reasons: First, the application may not be easily divided among the threads equally because it may not contain enough parallelism. Second, the huge memory wall as well as branch miss-prediction penalties are another obstacle to achieving a more equal work distribution among the cores (threads). We intend to identify at run-time the ideal division of work of an application among the available cores. We also intend to identify the ideal number of threads to use for a particular application.

8.5 Power Studies

We also intend to do some power studies of our SST and HSST schemes against already existing schemes such as the slipstream processor and the DCE scheme

[18, 36]. We expect that the SST scheme will yield the lowest power consumption because it tries to reduce the redundant computations, and so it executes fewer instructions than both the slipstream processor and the DCE scheme.

8.6 Simulation Work

Although we performed most of our studies using a simulator that we developed based on the SimpleScalar toolkit, we realize that it is not very modular, and so it is difficult to extend or modify it. We intend to revisit the design of our simulator to make it more modular such that it is easier to manipulate and use for more studies.

Our current simulator does not allow us to do power studies. In order to do our intended power studies, we plan to extend our simulator to make it simulate some power models. This will require an extensive development work.

Also, as we add more cores to our simulator, the simulation time increases. Simulating our HSST scheme with three subordinate threads, takes on average 8 hours for a single benchmark, for only 500 million instructions. We realize that it is crucial to our research to make the simulation time faster.

Finally, we intend to experiment with the floating-point benchmarks also. We expect that they will yield much higher performance than the integer benchmarks because our schemes exploit the available parallelism, which is more present in floating-point benchmarks.

Chapter 9

Summary and Conclusions

In keeping with the natural trend towards integration, current and future microprocessors are embracing the prosperity of single-chip multi-core processors. Although multi-core processors deliver significantly improved system throughput, single-thread performance is not addressed, and is negatively affected. This is because, multi-core architectures integrate simpler and smaller cores to achieve high throughput and meet a low power budget while high single-thread performance requires larger and more complex cores that have a wide instruction window to sustain a vast amount of instructions while serving long-latency memory instructions in parallel.

In this dissertation, we presented Symbiotic Subordinate Threading (SST), a novel processor architecture that utilizes idle cores on a single chip multiprocessor for improving single-thread performance while maintaining the flexibility to support multithreaded applications. We demonstrated that our SST scheme achieves high performance with minor hardware changes over dual-core processors such as the slipstream processor and the DCE processor. Its performance ranges from 7% to 45% over slipstream processor and 2% to 20% over DCE for integer benchmarks. We showed that SST can directly integrate the correct results of the subordinate thread into the main thread state without executing their corresponding instruc-

tions in the main thread. Result integration benefits our SST scheme in two ways. First, the subordinate thread incorrect results are not used as predictions in the main thread, hence eliminating the introduction of incorrect predictions in the main thread. Second, the number of instructions executed by the main thread is significantly reduced, resulting in a faster main thread and a more efficient use of the hardware resources. In other words, instructions are implicitly divided among the main thread and the subordinate thread, which allows more than one instruction to be serviced in parallel. Also, with a faster main thread, the subordinate thread spends less time on wrong path instructions, and hence it is less corruptive and performs its tasks more efficiently.

We also presented another implementation of SST in which the subordinate thread is informed of its own speculative state and uses this information to avoid executing more instructions that are likely to produce speculative incorrect results. We demonstrated that this new design of SST minimizes the number of times the subordinate thread deviates from the correct path, hence reducing the number of times the subordinate thread has to recover from the wrong path, i.e., less squash and re-start penalties. Also, a speculative aware subordinate thread is faster and more efficient, because it executes only the instructions that are more likely to produce useful results. However, we noticed that a speculative-aware subordinate thread neglects value prediction as an effective means for predicting memory addresses. As a result, the new SST scheme did not deliver good performance for benchmarks that benefited from value prediction such as *mcf* and *twolf*. The average performance improvement of the new SST scheme that employs a speculative-aware

subordinate thread over the old SST scheme is 9%, and ranges from -20% to 20%. We also noticed that the DCE scheme with a speculative-aware subordinate thread performed almost as well as the SST scheme with a speculative-aware subordinate thread. This is because the DCE scheme suffers in general from incorrect branch outcomes produced by the subordinate thread and consumed by the main thread as predictions; however, with a speculative-aware subordinate thread, the number of incorrect branch outcomes of the subordinate thread reduced significantly. The new SST scheme still has the advantage of executing fewer instructions than the DCE scheme.

Another extension of our SST scheme is the Hierarchical Symbiotic Subordinate Threading (HSST), in which a subordinate thread is allowed to have its own subordinate thread. The HSST scheme, brings together the advantages of high speculative subordinate threads with the advantages of low speculative subordinate threads while reducing the impact of their drawbacks. Highly speculative subordinate threads are faster with a larger instruction window to explore but they go more often on the wrong path. On the other hand, low speculative subordinate threads execute more instructions, and so they produce more correct results, however they are slower and have a limited instruction window to explore. We compared our HSST scheme with two subordinate threads (subA and subB) against two SST schemes, each employing one of the subordinate threads. SubA is less speculative than subB and follows the main thread in the hierarchy. SubB acts as a subordinate thread for subA. We report an average performance improvement of 16% over the SST configuration with subA and 18% over the SST configuration with subB. We also

presented results for HSST scheme with two, three, and four subordinate threads. Our results indicate that as the number of subordinate threads increases, the penalties associated with squashing and recovering the subordinate thread increase. Also, the benefit of the subordinate thread with regard to cache pre-fetching and branch pre-computation decreases as it becomes more speculative. An HSST configuration with three and four subordinate threads did not improve the performance beyond two subordinate threads.

The hardware requirements for SST and all its extensions are moderate. For pruning the main thread we only require the bitmaps (RSB and MSB) and the associated logic for maintaining them. For pruning the subordinate thread we require additional bitmaps (RSB and MSB) and their associated logic. The FIFO queue is needed for the communication between the subordinate thread and the main thread. The FIFO queue is perhaps the largest piece of hardware we add. However, the FIFO queue is a simple piece of hardware that allows forwarding the results of the subordinate thread to the main thread, and facilitates integration and improves the performance. Also, the FIFO queue is a much smaller and less complex piece of hardware than doubling the size of each core.

Finally, we plan to continue the work on SST. We believe that more enhancements can be made to SST to improve its performance.

BIBLIOGRAPHY

[1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. -Y. Chang, *The Case for a Single-Chip Multiprocessor* (Proc. 7th International Symposium on Architectural Support for Programming Languages and Operating Systems, 1996) .

[2] Lawrence Spracklen and Santosh G. Abraham, *Chip Multithreading, Opportunities and Challenges* (Proc. 11th International Symposium on High Performance Computer Architecture, 2005) .

[3] C. -K. Luk, *Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors* (Proc. 28th International Symposium on Computer Architecture, June 2001) .

[4] A. Roth and G. S. Sohi, *Speculative Data-Driven Multithreading* (Proc. 7th International Symposium on High Performance Computer Architecture (HPCA-7), 2001) .

[5] A. Roth, A. Moshovos, and G. S. Sohi, *Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation* (Proc. 13th Annual International Conference on Supercomputing, June 1999), pages 356-364.

[6] J. Pierce and T. Mudge, *Wrong-Path Instruction Prefetching* (Proc. 27th Annual IEEE/ACM International Symposium on Microarchitecture, 1994) .

[7] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, *Dynamic Speculative Precomputation* (Proc. 34th Annual IEEE/ACM International Symposium on Microarchitecture, 2001) .

[8] S. S. W. Liao, P. H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen, *Post-Pass Binary Adaptation for Software-Based Speculative Precomputation* (Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2002) .

[9] C. Zilles and G. S. Sohi, *Execution-Based-Prediction Using Speculative Slices* (Proc. 28th International Symposium on Computer Architecture, 2001) .

[10] M. Annavaram, J. Patel, and E. Davidson, *Data Prefetching by Dependence Graph Precomputation* (Proc. 28th International Symposium on Computer Architecture, June 2001) .

[11] J. Dundas and T. Mudge, *Improving Data Cache Performance by Pre-executing Instructions Under A Cache Miss* (Proc. International Conference on Supercomputing, July 1997), pages 68-75.

[12] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y. -F. lee, D. Lavery, and J. P. Shen, *Speculative Precomputation: Long-range Prefetching of Delinquent Loads* (Proc. 28th International Symposium on Computer Architecture, June 2001) .

[13] A. Roth and G. S. Sohi, *Register Integration: A Simple and Efficient Implementation of Squash Reuse* (Proc. 33 Annual IEEE/ACM International Symposium on Microarchitecture, 2000) .

[14] T. Aamodt, P. Marcuello, P. Chow, P. Hammarlund, and H. Wang, *Prescient Instruction Prefetch* (Proc. MTEAC-6, November 2002) .

[15] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, *Slipstream processors: improving both performance and fault tolerance* (Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000), pages 257-268.

[16] A. Sodani and G. S. Sohi, *Dynamic Instruction Reuse* (Proc. 24th International Symposium on Computer Architecture, June 1997) .

[17] R. Bodik, R. Gupta, and M. L. Soffa, *Load-Reuse Analysis: Design and Evaluation* (Conf. on PLDI-99, Atlanta, Georgia, May 1999) .

[18] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, *A Study of Slipstream Processors* (Proc. 33rd annual IEEE/ACM international symposium on Microarchitecture, December 2000) .

[19] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, *Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors* (Proc. 36th Annual IEEE/ACM International Symposium on Microarchitecture, December 2003) .

[20] O. Mutlu, H. Kim, J. Stark, and Y. Patt, *On Reusing the Results of Pre-Executed Instructions in a Runahead Execution Processor* (Computer Architecture Letters, Vol. 4, January 2005) .

[21] L. Kurian, P. T. Hulina, and L. D. Coraor, *Memory Latency Effects in Decoupled Architectures with a Single Data Memory Module* (Proc. 19th International Symposium on Computer Architecture, 1992), pages 236-245.

[22] R. Canal, J. M. Parcerisa, and Antonio Gonzalez, *Dynamic Cluster Assignment Mechanisms* (Proc. 6th International Symposium on High Performance Computer Architecture, 2000) .

[23] J. Keller, *The 21264: A Superscalar Alpha Processor with Out-of-Order Execution* (Microprocessor Forum, October 1996) .

[24] S. Palacharla, N. J. Jouppi, and J. E. Smith, *Complexity-Effective Superscalar Processors* (Proc. 24th International Symposium on Computer Architecture, 1997), pages 206-218.

[25] M. D. Smith, M. Horowitz, and M. S. Lam, *Efficient Superscalar Performance Through Boosting* (Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992) .

[26] D. Burger, T. M. Austin, and S. Bennett, *Evaluating Future Microprocessors: The Simplescalar Tool Set* (University of Wisconsin Madison, July 1996), CS TR-1308.

[27] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt, *Difficult-Path Branch Prediction Using Subordinate Microthreads* (Proc. 29th International Symposium on Computer Architecture, May 2002) .

[28] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, *Simultaneous Subordinate Microthreading (ssmt)* (Proc. 26th International Symposium on Computer Architecture, May 1999) .

[29] A. Roth and S. Sohi, *A Quantitative Framework for Automated Pre-Execution Thread Selection* (Proc. 35th International Symposium on Microarchitecture, 2002), pages 430-441.

[30] D. M. Tullsen, S. Eggers, and H. M. Levy, *Simultaneous Multithreading: Maximizing On-Chip Parallelism* (Proc. 22th International Symposium on Computer Architecture, 1995) .

[31] J. Rattner, *Multi-core to the masses* (Proc. 14th International Conference on Parallel Architectures and Compilation Techniques, 2005) .

[32] D. Kim and D. Yeung, *Design and Evaluation of Compiler Algorithms for Pre-Execution* (Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002), pages 159-170.

[33] D. Kim and D. Yeung, *A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code* (ACM Transactions on Computer Systems, August 2004), pages 326-379.

[34] D. Kim, S. S. Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen, *Physical Experimentation with Prefetching Helper Threads on Intel's Hyperthreaded Processors* (Proc. IEEE 2nd International Symposium on Code Generation and Optimization, March 2004), pages 27-38.

[35] R. Mameesh and M. Franklin, *Symbiotic Subordinate Threading* (Proc. ICCD-23, 2005) .

[36] Huiyang Zhou, *Dual-core execution: Building a Highly Scalable Single-thread instruction window* (Proc. 14th International Conference on Parallel Architectures and Compilation Techniques, 2005) .

[37] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. -Y. Chang, *The Case for a Single-Chip Multiprocessor* (Proc. 7th International Symposium on Architectural Support for Programming Languages and Operating Systems, 1996) .

[38] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, *Automatically Parallelizing Large Scale Program Behavior* (Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems, 2002) .

[39] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, *Using SimPoint for Accurate and Efficient Simulation* (Proc. SIGMETRICS, June 2003) .

[40] G. Hamerly, E. Perelman, and B. Calder, *How to Use SimPoint to Pick Simulation Points* (Proc. SIGMETRICS, 2004) .

[41] R. Barnes, E. Nustrom, J. Sias, S. Patel, N. Navarro, and W. Hwu , *Beating in-order stalls with flea-flicker two pass pipelining* (IEEE Transactions on Computers Vol. 55 No. 1, 2006) .

[42] S. Srinivasan, H. Akkary, T. Holman, and K. Lai, *A Minimal dual-core speculative multithreading architecture* (Proc. ICCD-22, 2004) .

[43] Craig Zilles and G. Sohi, *Master/Slave Speculative Parallelization* (Proc. 35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002) .

[44] Craig Zilles, *Master/Slave Speculative Parallelization and Approximate Code* (PhD Thesis, Univeristy of Wisconsin, 2002) .

[45] Ilya Ganusov and Martin Burtschur, *Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors* (Proc. 14th International Conference on Parallel Architectures and Compilation Techniques, 2005) .

[46] Ilya Ganusov and Martin Burtschur, *Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading* (Proc. 15th International Conference on Parallel Architectures and Compilation Techniques, 2006) .

[47] C. Moore, *POWER4 System Microarchitecture* (Proc. Microprocessor Forum, 2006) .

[48] R. Kalla, B. Sinharoy, and J. Tendler, *IBM POWER5 chip: a dual-core multithreaded processor* (Proc. 37th Annual IEEE/ACM International Symposium on Microarchitecture, 2004), pages 40-47.

[49] P. Kongetira, *A 32-way Multithreaded SPARC Processor* (Proc. Hot Chips 16, http://www.hotchips.org/archive/, 2004) .

[50] Advanced Micro Devices, *AMD Demonstrates Dual Core Leadership* (http://www.amd.com, 2004) .

[51] T. Maruyama, *SPARC64 VI: Fujitsu's Next Generation Processor* (Proc. Microprocessor Forum, 2003) .

[52] C. McNairy and R. Bhatia, *Montecito - the Next Product in the Itanium Processor Family* (Proc. Hot Chips 16, http://www.hotchips.org/archive,2004) .

[53] K. Farkas, N. Jouppi, and P. Chow, *Register File Considerations in Dynamically Scheduled Processors* (Proc. of 2nd International Symposium on High-Performance Computer Architecture, 1996), pages 40-51.

[54] James Burns and jean-Luc Gaudiot, *Area and System Clock Effects on SMT/CMP Throughput* (IEEE Transactions on Computers, Vol. 54, No. 2, February 2006) .

[55] O. Mutlu, H. Kim, and Y. Patt, *Techniques for efficient processing in runahead execution engines* (Proc. 32nd International Symposium on Computer Architecture, 2005) .

[56] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, *Continual Flow Pipelines* (Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 2004) .

[57] H. Akkary, R. Rajwar, and S. Srinivasan, *Checkpoint processing and recovery: towards scalable large instruction window processors* (Proc. 36th International Symposium on Microarchitecture, 2003) .

[58] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai, *Scalable load and store processing in latency tolerant processors* (Proc. 32nd International Symposium on Computer Architecture, 2005) .

[59] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, *Dynamically Allocating Processor Resources Between Nearby and Distant ILP* (Proc. 28th International Symposium on Computer Architecture, 2001) .

[60] P. H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, *Memory Latency-Tolerance Approaches for Itanium Processors: Out-of-Order Execution vs. Speculative Precompuation* (Proc. of the 8th International Symposium on High Performance Computer Architecture, 2002) .

[61] M. Franklin, *The Multiscalar Archtecture* (PhD Thesis, University of Wisconsin, Madison, December 1993) .

[62] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg, *Slipstream Execution Mode for CMP-Based Multiprocessors* (Proc. 9th International Symposium on High-Performance Computer Architecture, February 2003) .

[63] H. Akkary and M. A. Driscoll, *A Dynamic Multithreading Processor* (Proc. 31st International Symposium on Microarchitecture, 1998), pages 226-236.

[64] M. Franklin and G. S. Sohi, *The Expandable Split Window Paradigm for Exploiting Fine-grain Parallelism* (Proc. International Conference on Supercomputing, 1997) .

[65] C. B. Zilles, J. S. emer, and G. S. Sohi, *The Use of Multithreading for Exception Handling* (Proc. 32nd International Symposium on Microarchitecture, 1999), pages 219-229.

[66] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, *Transient-Fault Recovery Using Simultaneous Multithreading* (Proc. 29th Annual International Symposium on Computer Architecture, May 2002), pages 87-98.

[67] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, *Detailed Design and Evaluation of Redundant Multithreading Alternatives* (Proc. International Symposium on Computer Architecture, May 2002), pages 99-110.

[68] James E. Smith, *Decoupled Access/Execute Compuer Architectures* (Proc. 9th International Symposium on Computer Architecture, 1982) .

[69] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, *Trace Processors* (Proc. 30th Annual Symposium on Microarchitecture, 1997) .

[70] S. Vajapeyam and T. Mitra, *Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code sequences* (Proc. 24th International Symposium on Computer Architecture, 1997) .

[71] Q. Jacobson, E. Rotenberg, and J. E. Smith, *Path-based Next Trace Prediction* (Proc. 30th International Symposium on Microarchitecture, 1997) .

[72] Kahle, J., *The Cell Processor Architecture* (Proc. 38th Annual IEEE/ACM International Symposium on Microarchitecture, 2005) .

[73] Dongkeun Kim, *Compiler-Based Pre-Execution* (Department of Electrical and Computer Engineering, University of Maryland in College Park, 2004) .