ABSTRACT

Title of dissertation: COMPILER-BASED PRE-EXECUTION

Dongkeun Kim, Doctor of Philosophy, 2004

Dissertation directed by: Professor Donald Yeung
Department of Electrical and Computer Engineering

Pre-execution is a novel latency-tolerance technique where one or more *helper threads* run in front of the main computation and trigger long-latency delinquent events early so that the main thread makes forward progress without experiencing stalls. The most important issue in pre-execution is how to construct effective helper threads that quickly get ahead and compute the delinquent events accurately. Since the manual construction of helper threads is error-prone and cumbersome for a programmer, automation of such an onerous task is inevitable for pre-execution to be widely used for a variety of real-world workloads.

In this thesis, we study *compiler-based pre-execution* to construct prefetching helper threads using a source-level compiler. We first introduce various compiler algorithms to optimize the helper threads; *program slicing* removes noncritical code unnecessary to compute the delinquent loads, *prefetch conversion* reduces blocking in the helper threads by converting delinquent loads into nonblocking prefetches, and *loop parallelization* speculatively parallelizes the targeted code region so that more memory accesses are overlapped simultaneously. In addition to these algorithms to expedite the helper threads, we also propose several important algorithms to select the right loops for pre-execution regions and pick up the best thread initiation scheme to invoke helper threads. We implement all these algorithms in the Stanford University Intermediate Format (SUIF) compiler infrastructure to automatically generate effective helper threads at the program source level. Furthermore, we replace the external tools to perform program slicing and offline profiling in our most aggressive compiler framework with static algorithms to reduce the complexity of compiler implementation. We conduct thorough evaluation of the compiler-generated helper threads using a

simulator that models the research SMT processor. Our experimental results show compiler-based pre-execution effectively eliminates the cache misses and improves the performance of a program.

In order to verify whether prefetching helper threads provide wall-clock speedup even in real silicon, we apply compiler-based pre-execution in a real physical system with the Intel Pentium 4 processor with Hyper-Threading Technology. To generate helper threads, we use the pre-execution optimization module in the Intel research compiler infrastructure and propose three helper threading scenarios to invoke and synchronize the helper threads. Our physical experimentation results prove prefetching helper threads indeed improve the performance of selected benchmarks. Moreover, to achieve even more speedup in real silicon, we observe several issues need to be addressed a priori. Unlike the research SMT processor where most processor resources are shared or replicated, some critical hardware structures in the hyper-threaded processor are hard-partitioned in the multithreading mode. Therefore, the resource contention is more intricate, and thus helper threads must be invoked very judiciously. In addition, the program behavior dynamically changes during execution and the helper threads should adapt to it to maximize the benefit from pre-execution. Hence we implement user-level library routines to monitor the dynamic program behavior with little overhead and show the potential of having a runtime mechanism to dynamically throttle helper threads. Furthermore, in order to activate and deactivate the helper threads at a very fine granularity, having light-weight thread synchronization mechanisms is very crucial.

Finally, we apply compiler-based pre-execution to multiprogrammed workloads. When introducing helper threads in a multiprogramming environment, multiple main threads compete with each other to acquire enough hardware contexts to launch helper threads. In order to address such a resource contention problem, we propose a mechanism to arbitrate the main threads. Our simulation-based experiment shows pre-execution also helps to boost the throughput of a multiprogrammed workload by reducing the latencies in the individual applications. Moreover, when the helper thread occupancy of each main thread in the workload is not too high, multiple main threads effectively share the hardware contexts for helper threads and utilize the processor resources in the SMT processor.

COMPILER-BASED PRE-EXECUTION

by

Dongkeun Kim

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
2004

ACKNOWLEDGMENTS

There are many people who helped me with making this thesis possible. First of all, I really thank my advisor, Dr. Donald Yeung. It has been always my pleasure to discuss problems and findings with him for the past four years. I also enjoyed the time when I took his ENEE446 class during the first semester in my graduate study and I worked as a TA for his class the next semester. His devotion to research and advising has inspired me throughout the entire period of my study in the graduate program.

I also thank my dad and mom, who have been always my biggest supporters and counsellors throughout my entire life. Especially, I will never forget the day when I left my country to begin the graduate study in US. Everything on my way ahead was uncertain and I had complicated feelings mixed with excitement and nervousness. Without their consistent love and caring, it would have been impossible to go through every step to arrive at this moment. I also thank my lovely sisters, Jueun and Juhyun; I admit I have not been a good brother so far, but I promise I will be from now on.

During my stay in US for the past five years, I was so lucky to have many good people around me in both Maryland and California. I thank my lab-mates in the SCAL of the University of Maryland for their help and sharing good memory: Seungryul, Gautham, Deepak, Hameed, Sumit, Wanli, Xuanhua, Ammer, Sada, Zahran, Anashua, Vinod, Aneesh, Jayanth, Brinda, Renju, Barat, and all the other members in our group. I also appreciate the members of my PhD defense committee: Dr. Manoj Franklin, Dr. Bruce Jacob, Dr. Rajeev Barua, and Dr. Chau-wen Tseng. I would like to acknowledge the Korean graduate students at Maryland and their family: Woochul, Joonhyuk, Seungjong, Yeongsun, Younggu, Kyechong, Jusub, Bongwon, Hyunmo, Bohyung, Jihwang, Yooah, Junpyo, and many others. Also, my internship experience at Intel became more enjoyable and valuable thanks to Shih-wei, Perry, Jamison, Hong, John, Xinmin, Murali, Ryan, Satish, Dan, Gerolf, Youngsoo, Daehyun, Margarita, Nancy, Chris, Chaoying, John Young, Claire, David, Kihoon, Sunghoon, Haejin, and all my colleagues and friends in the San Francisco Bay area.

My friends in both US and Korea have been always supportive and they have encouraged me to keep moving forward: Jiwon, Mikyung, Jaehyuk, Junkwan, Seongjun, Youngho, Dongook, Youngjun, Daeki, Seungkyung, Junghyun, Haejin, Hyoje, Jonghwa, Hyomin, Hongseok, Jihyun, Jonghee, Heesun, Insuk, Junho, Saekyung, Chulsung, Woojay, Daewoo, Dongeun, Taemin, Hyungchan, Jun, Seungyong, Jungsoo, Sangwoo, and all other friends of mine.

Especially, I would like to thank Dr. Hyunook Kim. He has been a good friend, devoted mentor, and hero to me, and I have always admired his passion for research. Also, I pray for God to bless his family, Hyunmi and Hanna.

And, I apologize to those I have inadvertently left out.

Lastly, I thank God for giving me opportunity to know him at the last phase of my graduate study and showing me his endless love.

TABLE OF CONTENTS

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Motivation 1

1.2 Contributions 7

1.3 Roadmap 10

2 Background 12

2.1 Helper Threading 12

2.2 Multithreading 14

2.2.1 Multiprogramming 14

2.2.2 Parallel Processing 15

2.2.3 Pre-Execution 16

2.3 Data Prefetching 17

2.3.1 Prediction-Based Prefetching 18

2.3.2 Execution-Based Prefetching 19

3 Overview of Compiler Algorithms for Pre-Execution 21

3.1 Identifying Targets 21

3.1.1 Identification of Delinquent Loads 21

3.1.2 Identification of Target Code Regions 24

3.1.3 Code Cloning 25

3.2 Improving Effectiveness of Pre-Execution 26

3.2.1 Program Slicing 27

3.2.2 Prefetch Conversion 27

3.2.3 Loop Parallelization 28

3.2.4 Synchronization 29

3.3 Minimizing Ineffectiveness of Pre-Execution 30

3.4 Preserving Program Correctness 32

3.4.1 Removing Side Effects 33

3.4.2 Handling Exceptions 33

4 SUIF-Based Compiler Frameworks for Pre-Execution 34

4.1 Compiler Algorithms for SUIF-Based Aggressive Framework 34

4.1.1 Identifying Targets 35

4.1.2 Improving Effectiveness of Pre-Execution 39

4.1.3 Minimizing Ineffectiveness of Pre-Execution 54

4.1.4 Preserving Program Correctness 55

4.2 Compiler Algorithms for SUIF-Based Reduced Frameworks 57

4.2.1 Eliminating Program Slicing 57

4.2.2 Prefetch Conversion without Program Slicing 59

4.2.3 Eliminating Cache-Miss Profiles 60

4.2.4 Eliminating Loop Profiles 62

4.3 Implementation 63

4.3.1 Summary of Compiler Frameworks 64

4.3.2 Aggressive Prototype Compiler 65

4.3.3 Reduced Prototype Compilers 66

4.3.4 Live-In Passing 67

4.3.5 Code Generation 68

4.3.6 ISA Support 70

5 Evaluation in a Simulation-Based Environment 73

5.1 Experimental Methodology 73

5.1.1 Simulator Configuration 73

5.1.2 Benchmarks 75

5.2 Results of Aggressive Compiler 76

5.2.1 Characterization of Helper Threads 77

5.2.2 Performance Results 81

5.2.3 Contributions of Algorithms 84

5.2.4 Comparison of Thread Initiation Schemes 87

5.2.5 Comparison of HW/SW Synchronization Mechanisms 89

5.2.6 Accuracy of Helper Threads 91

5.3 Results of Reduced Compilers 94

5.3.1 Impact of C Compiler's Code Removal 95

5.3.2 Impact of SUIF-Based Prefetch Converter 98

5.3.3 Impact of Eliminating Cache-Miss Profiles 98

5.3.4 Impact of Eliminating Loop Profiles 100

5.4 Summary 101

6 Pre-Execution Optimization Module in the Intel Research Compiler Infrastructure 103

6.1 Unique Aspects of a Physical System 104

6.1.1 Hardware Resource Management 104

6.1.2 Thread Synchronization Mechanism and Cost 105

6.1.3 Available Hardware Contexts 107

6.2 Compiler Algorithms for the Intel Compiler 107

6.2.1 Identifying Targets 107

6.2.2 Improving Effectiveness of Pre-Execution 110

6.2.3 Minimizing Ineffectiveness of Pre-Execution 115

6.2.4 Preserving Program Correctness 116

6.3 EmonLite: User-Level Library Routines for Runtime Performance Monitoring . . 116

6.3.1 EmonLite vs. Intel VTune Performance Analyzer 117

6.3.2 Components of EmonLite 118

6.3.3 Implementation Issues 118
6.3.4 Example Usage: Chronology of L2 Miss Events 119
6.4 Implementation 120
6.4.1 Prototype Compiler 120
6.4.2 Live-in Passing 121
6.4.3 Code Generation 121

7 Evaluation in a Physical System with the Intel's Hyper-Threaded Processor 123
7.1 Experimental Evaluation 123
7.1.1 System Configuration 123
7.1.2 Benchmarks 124
7.1.3 Baseline 126
7.1.4 Evaluation of Static Thread Initiation Schemes 127
7.1.5 Evaluation of Dynamic Throttling of Helper Threads 137
7.1.6 Sensitivity of Program's Dynamic Behavior 139
7.2 Key Observations 140
7.2.1 Impediments to Speedup 141
7.2.2 Essential Mechanisms 142

8 Evaluation of Compiler-Based Pre-Execution in a Multiprogramming Environment 143
8.1 Multiprogramming Simulation Methodology 143
8.2 Evaluation Results 145

9 Related Work 149
9.1 Multithreading 149
9.1.1 Speculative Multithreading 150
9.1.2 Helper Threading 151
9.1.3 Single-Chip Multithreading Processors 153
9.2 Data Prefetching 155

9.2.1 Prediction-Based Prefetching 155

9.2.2 Execution-Based Prefetching 156

9.3 Pre-Execution 158

9.3.1 Compiler-Based Pre-Execution 159

9.3.2 Linker-Based Pre-Execution 160

9.3.3 Hardware-Based Pre-Execution 162

9.3.4 Physical Experimentation with Prefetching Helper Threads 163

10 Conclusion 165

10.1 Summary of Contributions and Implications of the Research 165

10.2 Future Direction 167

Bibliography 169

LIST OF TABLES

4.1 Compilers used to generate the helper thread code for our experiments on the SMT processor simulator. The compilers differ in how they remove the unnecessary code, select the prefetch conversion candidates, identify the delinquent load instructions, and estimate the inner-most loop work. 64

5.1 SMT processor simulator configuration. 74

5.2 Benchmark characteristics. 76

5.3 Static characterization of the helper thread code generated by our aggressive compiler. . 77

5.4 Dynamic characterization of the helper thread code generated by our aggressive compiler. 79

6.1 Hardware resource management in the Intel's hyper-threaded processor. 104

7.1 Physical system configuration. 124

7.2 Benchmark characteristics. 125

7.3 Statistics for the StripMining scheme. 128

7.4 Percentage SD statistics. Profiles are collected without launching helper threads. 137

8.1 Benchmarks for the multiprogramming experiments. The TDF values for each benchmark are listed in the third row, and the helper thread initiation schemes employed in each benchmark are listed in the last row. Benchmark groupings are indicated in the first row: Group 1 benchmarks have high-TDF values, Group 2 benchmarks have medium-TDF values, and Group 3 benchmarks have low-TDF values. 145

LIST OF FIGURES

1.1 Four approaches to automatically extract helper threads: a) compiler-based extraction, b) linker-based extraction, c) dynamic optimizer-based extraction, and d) hardware-based extraction. Techniques differ in when and how the code extraction is performed. 4

4.1 Algorithm to compute the set of pre-execution regions, P. Φ denotes the empty set. To estimate the loop work, we rely on offline profiles for loop-trip counts. 37

4.2 VPR code example to demonstrate our program slicing. Labels "1" – "4" indicate the cache-missing memory references selected for the slicing criteria. Labels "5" and "6" indicate the memory references that will be eliminated by store removal. Labels "7" and "8" indicate loops that bound the scope of slicing. Labels "S1," "S2," "S3," and "S4" show the slice result for the selected memory references. 42

4.3 Code generated by our aggressive compiler for the `get_non_updateable_bb()` function from Figure 4.2. a) Pre-execution code after program slicing in which an `asm` macro is added for code pinning (labeled "1"). b) Pre-execution code after program slicing and prefetch conversion (labeled "2"). Bold-face code denotes the cache-missing memory references. . 46

4.4 Three helper thread initiation schemes: a) Serial, b) DoAll, and c) DoAcross. Solid lines denote the main thread, dotted lines denote the helper threads, arrows denote the thread spawning, and numeric labels denote the loop iteration counts. 48

4.5 Algorithm to decide the thread initiation scheme for pre-execution regions. 51

4.6 Code generated by our compiler without program slicing for the `get_non_updateable_bb()` function from Figure 4.2. An `asm` macro is added for code pinning (labeled "1"). 58

4.7 Algorithm to compute the set of pre-execution regions, P, without loop-trip count profiles. Φ denotes the empty set. 63

4.8 Design of our most aggressive prototype compiler. This compiler uses Unravel to perform program slicing and offline profiles to drive optimizations. Arrows denote the interactions between the compiler modules. 65

4.9 Major components in our least aggressive prototype compiler. This compiler uses dead code elimination to remove the unnecessary code and static analysis to identify the delinquent loads and select the pre-execution regions. Arrows denote the interactions between the compiler modules. 66

4.10 Code generated by the aggressive compiler to implement a) the Serial scheme for the AMMP benchmark and b) the DoAll scheme for the VPR benchmark. Code to initiate pre-execution and synchronize the helper threads appears in bold face. 71

4.11 Code generated by the aggressive compiler to implement the DoAcross scheme for the TWOLF benchmark. Code to initiate pre-execution and synchronize the helper threads appears in bold face. 72

5.1 Normalized execution time broken down into the “Busy Time,” “Overhead,” and “Mem Stall” components. The “Baseline” and “Pre-exec” bars show the performance without and with pre-execution, respectively. 81

5.2 Cache-miss coverage broken down into five categories: uncovered cache misses occurring outside the pre-execution regions (“Non Region”), fully covered cache misses by pre-execution (“Full”), partially covered cache misses by pre-execution (“Partial”), remaining cache misses that hit in the L2 cache (“L2-Hit”), and remaining cache misses satisfied from main memory (“Mem”). 83

5.3 Impact of individual algorithms on overall performance in the aggressive compiler. The “Parallel” bars perform pre-execution with speculative loop parallelization only; the “Slicing” bars perform pre-execution with program slicing and speculative loop parallelization, but without prefetch conversion; and the “Pre-exec” bars perform pre-execution with all optimizations including prefetch conversion. 86

5.4 Comparing the helper thread initiation schemes. The “Baseline” bars report the execution time without pre-execution. The remaining bars report the execution time when the Serial, DoAll, and DoAcross schemes are applied, respectively. The thread initiation schemes selected by our compiler appear in bold face. 88

5.5 Impact of architectural support for synchronization. The "SW Semaphore" and "HW Semaphore" bars report the execution time with the software and hardware counting semaphores, respectively. All bars are normalized to the "Baseline" bars from Figure 5.1. 90

5.6 Breakdown of the pre-execution regions into 3 major categories: originally parallel, parallel after program slicing and store removal, and serial. The serial category is further broken down into regions using the SERIAL scheme, regions speculatively parallelized affecting control flow only, and regions that are incorrectly parallelized. 93

5.7 Normalized execution time broken down into the "Busy Time," "Overhead," and "Mem Stall" components. Groups of bars labeled "Compiler A" to "Compiler E" report performance for the 5 compilers from Table 4.1. 94

5.8 Comparison of dynamic instruction counts between Compilers A and B. The "Main Thread" bars report the dynamic instruction counts for the main thread. The "Compiler A" and "Compiler B" bars report the dynamic instruction counts for the helper threads generated by Compilers A and B, respectively. Each bar is broken down into the "Compute" and "Overhead" components. 96

6.1 Example of the EMONLITE code instrumentation: `price_out_impl()` of MCF. 119

6.2 Chronology of the L2 miss events at a fine granularity: `price_out_impl()` of MCF . . . 120

6.3 Coarse-grain phase behavior of the L2 miss events: `price_out_impl()` of MCF 120

7.1 VTune profiles: cycles and L2 misses associated with the identified delinquent loads . . . 126

7.2 Normalized execution time without thread pinning. The execution time without helper threads, but with thread pinning, is our baseline for the actual performance measurements. 127

7.3 Speedup of the static thread initiation schemes. From the left to right bars, we label each bar as "SO" for the SERIAL scheme with the OS API, "SH" for the SERIAL scheme with the hardware mechanism, "MO" for the STRIPMINING scheme with the OS API, and "MH" for the STRIPMINING scheme with the hardware mechanism. The speedup is computed by normalizing the execution time of each configuration to that of the baseline. 129

7.4 Speed of the cache-miss kernels for various situations. For these experiments, we use the SERIAL scheme with the hardware thread synchronization mechanism. 132

7.5 Cache-miss coverage using the SERIAL scheme with the hardware thread synchronization mechanism. 134

7.6 Dynamic behavior of the performance events in BZIP2 with and without helper threads. a) Chronology of the L2 miss events. b) Chronology of the cycle events. 136

7.7 Performance comparison between the STRIPMINING scheme and the DYNAMIC STRIPMINING scheme. From left to right bars, each bar is labeled to present the thread initiation scheme and the thread synchronization mechanism: "SO" for the STRIPMINING scheme with the OS API, "DO" for the DYNAMIC STRIPMINING scheme with the OS API, "SH" for the STRIPMINING scheme with the hardware mechanism, and "DH" for the DYNAMIC STRIPMINING scheme with the hardware mechanism. 139

7.8 Percentage SD of L2 misses for various sampling periods. 140

8.1 Weighted speedup of the multiprogrammed workloads. "Baseline Simul" denotes simultaneous execution without pre-execution. "Pre-exec Simul" and "Pre-exec Time" denote pre-execution with simultaneous execution and time sharing, respectively. The TDF group that each application belongs to and the baseline IPC for the application are shown in parentheses. 146

Chapter 1

Introduction

1.1 Motivation

Pre-execution is a general latency-tolerance technique, in which one or multiple number of threads, called *helper threads*, run in front of the main computation thread (*main thread* for short) and trigger long-latency events, such as the last-level cache misses, on the main thread's behalf. When helper threads are effective, the main thread runs without experiencing stalls, thereby reducing the execution time. Data prefetching is a good example that clearly shows the usefulness of pre-execution. Among various performance-degrading events, cache misses, especially those incurring memory access, have been the main target of many latency-tolerance techniques due to their huge impact on the program performance. Numerous research proposals have studied tolerating the large memory latency by prefetching based on repetitive memory access patterns in a program. Such *prediction-based* prefetching is very effective for affine-array access, which is easily found in many scientific workloads. However, nonscientific workloads often exhibit irregular memory access patterns that are hard to predict; such irregularities are usually found in pointer-chasing traversal, index-array reference, or hash-table access. Rather than predict, pre-execution actually *executes* certain instructions that are part of the original program to compute the cache-missing memory references so that accurate prefetches can be generated. Thanks to execution-based nature, irregularity in memory access patterns is not a problem in pre-execution; in fact, this is one of the most compelling reasons why pre-execution is a promising data-prefetching technique.

In addition to its execution capability, we find pre-execution also has much potential to uncover large amount of thread-level parallelism, especially *memory-level parallelism.* In pre-execution, prefetching helper threads only execute code to compute effective addresses of long-latency cache-missing loads, called *delinquent loads*, and skip the remaining code, which the main thread must execute to ensure program correctness. Moreover, most pre-execution techniques do not allow helper threads to perform store operations. In other words, computed results by the helper threads are never integrated into the shared machine state so that pre-execution does not disrupt the main computation. Consequently, all dependences through memory are completely eliminated in the helper threads. Unless values necessary to compute the delinquent load addresses are communicated via memory, multiple helper threads can run in parallel without suffering from any dependence violation. Furthermore, since helper threads never affect the correctness of the main computation, they can be speculative and very aggressive. These unique characteristics of helper threads allow to overlap more memory accesses simultaneously, exploiting abundant amount of memory-level parallelism, which conventional multithreading techniques can hardly uncover.

Multithreading has been studied in the research community of computer architecture field for decades. Many research proposals investigate multiprogramming or parallel processing paradigms in large multiprocessor systems or recent single-chip multithreading processors. In particular, we view multithreading as a way to exploit thread-level parallelism that resides in workloads. Multiprogramming is a well-known example of multithreading where multiple independent application threads are executed on different hardware contexts available in a system. In multiprogramming, the amount of thread-level parallelism is the same as the number of applications that run together. As long as there are enough threads to be fed into the system, thread-level parallelism is easily extracted. However, it is sometimes hard to find applications to fully utilize the available hardware contexts in the system. Moreover, launching more threads may not be helpful to boost the overall processor throughput, especially when there exists hardware resource contentions among threads, *e.g.*, memory bus saturation. Another limitation of multiprogramming is that it does not improve the performance of each individual program in the multiprogrammed workload. Thus multiprogramming is not helpful when improving single-thread performance is crucial. On the other hand, parallel processing extracts thread-level parallelism from a single program by dividing it into multiple threads to run in parallel. Parallel processing has been very effective for scientific workloads that perform intensive vector computations on huge data space. For nonscientific workloads, however, parallel processing is often ineffective since complicated dependence structures and control flow in those workloads prevent a program from being parallelized. Therefore, people have searched for new ways to uncover thread-level parallelism in a workload and exploit it, thereby improving the processor performance.

With the recent advent of single-chip multithreading processors such as the Simultaneous Multithreading (SMT) processor or Chip Multiprocessor (CMP), we believe a new era for multithreading has begun. Single-chip multithreading processors have opened up a great deal of opportunities in the multithreading paradigm due to their unique characteristics distinguished from conventional large multiprocessor systems. First, the communication delay between threads is greatly shortened compared to that of conventional multiprocessor systems since all inter-thread communications occur within a single die. Because of the small communication and thread synchronization cost, it becomes possible to exploit new types of thread-level parallelism at a much finer granularity, which was not feasible in large multiprocessor systems. Secondly, some critical hardware resources in single-chip multithreading processors are often shared or partitioned. Thus there exist intriguing tradeoffs regarding how to arbitrate those hardware resources among multiple threads. On the other hand, hardware-resource sharing is not always problematic, but sometimes opportunistic. For instance, sharing of a cache, *e.g.*, the last-level cache that is located right above main memory, makes it possible for some of spare hardware contexts in a multithreading processor to be used for executing judiciously selected instruction sequences and bringing data cache blocks on behalf of the main computation, thereby helping the execution of the program indirectly.

Recent architectural trend in the industry is in favor of multithreading as well. Following Moore's law, the number of transistors integrated on a single die has increased exponentially, reaching almost one billion in the near future. To make use of the available transistor budget, people have started putting multiple cores and/or multiple threads on a single die to support multithreading. Since the first proposals of SMT and CMP processor models from academia, there have been a great deal of studies to examine the potential of exploiting such multithreading processors. After all those efforts, commercial single-chip multithreading processors like the Intel® Pentium® 4 processor with Hyper-Threading Technology or IBM® POWER5 [34] have been recently released. Moreover, Sun Microsystems also announced their plan for productizing single-chip multithreading processors in the near future [51]. As the transistor count on a chip

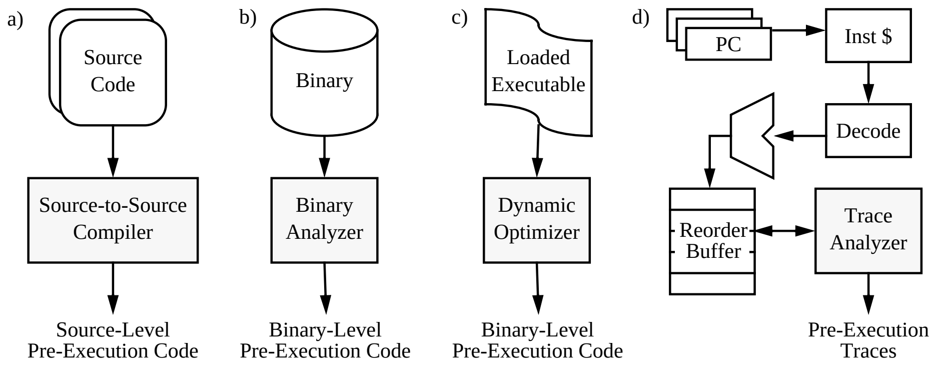


Figure 1.1: Four approaches to automatically extract helper threads: a) compiler-based extraction, b) linker-based extraction, c) dynamic optimizer-based extraction, and d) hardware-based extraction. Techniques differ in when and how the code extraction is performed.

grows, we expect such trend toward multithreading processors to continue and there will be spare hardware contexts available for purposes other than performing useful computations, *e.g.*, *helper threading*. We believe pre-execution can play a crucial role in exploiting multithreading processors and thorough investigation of pre-execution is inevitable.

The most important issues in pre-execution is the construction of effective helper threads. Manual construction of helper threads is error-prone and cumbersome for a programmer. Hence for pre-execution techniques to be widely used for a variety of real-world workloads, this job of constructing helper threads must be fully automated. There are various approaches to generate helper threads automatically that can be categorized based on how and when in the program's lifetime helper threads are constructed.

Figure 1.1 shows four possible approaches to extract helper threads. Figure 1.1a depicts *compiler-based extraction* where a source-to-source compiler analyzes a program code and generates helper threads at the program source level [36]. Figure 1.1b illustrates *linker-based extraction*, which generates helper threads using binary analysis [41, 64]. The extracted helper thread code is in binary format and is attached to the original program binary. Figure 1.1c describes *dynamic optimizer-based extraction*[1], which analyzes and extracts binary-level code similar to linker-based

[1]To our knowledge, dynamic optimizer-based extraction has not been investigated. However, we believe it to be a viable approach and include it here for completeness.

extraction, but does so at runtime using dynamic optimization techniques. Finally, Figure 1.1d shows *hardware-based extraction* [6, 19, 48, 73]. In this approach, helper threads are extracted from instruction traces at runtime by analyzing retired instructions using a special hardware structure. Extracted helper threads are cached so that they can be initiated when the corresponding code region is encountered in the future. By its very nature, the hardware approach is automated.

Each approach in Figure 1.1 exhibits very different characteristics due to the fact that code extraction is performed using different analysis techniques in different phases of a program's lifetime. This leads to several tradeoffs as described below:

Information for code extraction. Dynamic optimizer- and hardware-based extraction makes use of runtime information to construct helper threads. However, the runtime overhead associated with dynamic optimization prevents detailed analysis, and the scope of trace analysis in hardware is limited to the size of the post-retirement queue that analyzes dependences between instructions. On the other hand, such runtime information is not available for compiler- and linker-based approaches. Thus they need to rely on other sources, *e.g.*, offline profiles or static algorithms, to acquire necessary information for code extraction. However, compiler- and linker-based approaches utilize high-level program information like the program or data structure, and such high-level information is more abundant in the earlier phases.

Platform independence of extracted code. As the helper thread is extracted later in time, it becomes more dependent on the hardware platform. Since hardware-based extraction requires a special hardware structure to analyze instruction traces, it is strictly tied to the machine implementation. Whenever a new processor design is introduced, the trace analyzer must be redesigned accordingly. Linker-based extraction is less dependent on processor implementation than hardware-based extraction, but it still depends on the Instruction Set Architecture (ISA) of the target processor. For processors with different ISAs, one should have different binary analyzers to extract helper threads. Dynamic optimizer-based extraction is in the middle of hardware- and linker-based approaches; it sometimes requires special hardware that is tied with processor implementation and depends on the ISA of the processor. However, compiler-based extraction is completely independent of target platforms since it generates source code that can be compiled for any machine architecture. Consequently, this approach generates portable code.

Transparency to the user. As the extraction occurs later in time, it becomes more transparent to the user. In hardware-based extraction, the trace analyzer performs analysis and constructs helper threads at runtime, and all the necessary hardware is implemented within the processor. Therefore, this approach is completely transparent to the user. Dynamic optimizer-based extraction has similar characteristics. However, the other two approaches are less transparent since they require additional compilation steps such as code analysis and offline profiling. Between the two approaches, compiler-based extraction is less transparent since it requires the program source code, which is sometimes unavailable as is the case for legacy codes.

Hardware complexity. Compiler- and linker-based extraction does not affect hardware complexity of a processor since helper threads are extracted by software rather than hardware. Although the processor needs to support some features to handle multithreading such as suspending and forking threads, these are often provided by most multithreading processors. On the other hand, hardware-based extraction, of course, increases the hardware complexity due to the trace analyzer and cache structure for storing the extracted helper threads. Moreover, because of the newly introduced special hardware structures, the testing and validation cost increases in the hardware-based approach. Finally, hardware complexity of dynamic optimizer-based extraction depends on how much special hardware it requires.

All four approaches are worth pursuing since each approach has its own advantages as demonstrated above. In this thesis, however, we focus on compiler-based extraction due to the following reasons. First of all, compiler-based approach requires little modification to existing multithreading processors. Therefore, once a compiler for pre-execution is available, we can immediately apply and evaluate the generated helper threads. In addition, compiler-based extraction is platform-independent and the constructed helper threads, in the form of program source code, are portable. Moreover, we can utilize high-level program information to help construction of effective helper threads, and the capability of inter-procedure analysis of a compiler is another advantage of compiler-based approach. Finally, while other approaches, except for dynamic optimizer-based extraction, are evaluated previously in many research proposals, compiler-based construction of helper threads is relatively new and has not been fully investigated. Therefore, there is much room to explore.

1.2 Contributions

This dissertation has several contributions as listed below.

1. **Development of Compiler Algorithms for Pre-Execution**. We develop several compiler algorithms to construct effective helper threads that perform data prefetching. At the heart of our compiler algorithms are three critical optimization techniques to expedite helper threads so that they can trigger cache misses in front of the main computation. First, *program slicing* eliminates noncritical code that is unnecessary for computing delinquent load addresses. Second, *prefetch conversion* removes blocking in helper threads by converting delinquent loads into nonblocking prefetch instructions. Third, *speculative loop parallelization* helps to overlap multiple memory accesses by exploiting thread-level parallelism within the target code region. In addition to these algorithms to speedup helper threads, we also propose algorithms to select optimal pre-execution regions that encompass the identified delinquent loads and to decide a proper thread initiation scheme for each pre-execution region. Finally, to evaluate compiler-based pre-execution in a real physical system, where the thread synchronization cost is significantly high, we propose three helper threading scenarios to invoke and synchronize helper threads.

2. **Implementation of Pre-Execution Compiler Frameworks using SUIF**. We implement the compiler algorithms and prototype five different compiler frameworks using the Stanford University Intermediate Format (SUIF) compiler infrastructure. We introduce one

very aggressive-complier implementation that makes use of powerful program slicer and offline profiles along with the SUIF. We also propose four reduced compilers, in which the design of compiler framework is simplified by substituting the external tools for static compiler algorithms. To the best of our knowledge, our compiler frameworks are the first source-level compiler to generate helper threads fully automatically without any manual intervention.

3. **Development of Helper Thread Optimization Module in the Intel Compiler**. We work with a group of people at Intel [35] to develop and improve the compiler optimization module to generate helper threads, which is built in the Intel Research Compiler Infrastructure (*Intel compiler* for short). The analyses and optimizations to enable pre-execution are performed on the Intel compiler's intermediate representation, IL0. Thus the compiler module is not dependent on a specific programming language, but applicable for various languages including C, C++, and FORTRAN. While the compiler module is also able to generate binaries targeting the Intel's Itanium Processor Family, in this thesis, we only generate helper threads for the IA-32 architecture as the target platform.

4. **Evaluation of Compiler-Based Pre-Execution in a Research SMT Simulator**. We perform various experiments to evaluate our SUIF-based compiler frameworks using a SimpleScalar-based SMT processor simulator that enables detailed timing simulations. In addition to measuring performance impact of pre-execution, we also investigate several aspects of the compiler algorithms, cache-miss coverage of the helper threads, contribution of each optimization algorithm to the program performance, comparison between the thread initiation schemes, and tradeoffs between the aggressive and reduced compilers. Our experimental results show the compiler-generated helper threads effectively eliminate cache misses in the main thread and significantly improve the performance of selected benchmarks from the SPEC CPU2000 and Olden Benchmark Suites. Moreover, we also show that our compilers make the right decision on selecting pre-execution regions and thread initiation schemes.

5. **Evaluation of Compiler-Based Pre-Execution in a Real Physical System**. We conduct physical experimentation to evaluate prefetching helper threads on real silicon, *i.e.*, the Intel Pentium 4 processor with Hyper-Threading Technology. We provide a complete procedure to perform physical experimentation with helper threads from collecting profile information, to generating helper threads using an optimization module in the Intel compiler, invoking and synchronizing helper threads based on static and dynamic thread initiation schemes, monitoring dynamic behavior of a program at runtime at a fine granularity, and dynamically throttling helper threads to reduce ineffectiveness of pre-execution. We believe our experimental methodology is a good reference for other kinds of physical-system-based evaluation. Our experimental results show that prefetching helper threads indeed provide wall-clock speedup for various benchmarks on the existing hyper-threaded processor.

6. **Identification of Impediments to Speedup on Real Silicon**. From our physical experimentation, we uncover impediments to speedup when applying helper threads in a real physical system. In hyper-threaded processors, hardware resource contention between the two logical processors possibly degrades the performance of the main thread when the helper thread does not help. Moreover, high thread-synchronization cost in a physical system limits frequent thread synchronization and communication at a fine granularity. To overcome these constraints in a physical system, we evaluate two different thread-synchronization mechanisms and show that having light-weight thread-synchronization mechanisms is crucial to achieve good performance gain with helper threads. In addition, certain runtime system to dynamically throttle helper threads by monitoring the program behavior can greatly help to get even more speedup with helper threads on real silicon.

7. **Application of Compiler-Based Pre-Execution in a Multiprogramming Environment**. We also apply prefetching helper threads in a multiprogramming environment. When performing pre-execution in such computing situation, multiple main threads compete with each other to acquire enough hardware contexts to launch helper threads. Since the number of hardware contexts in a multithreading processor is limited, we must intelligently allocate

this critical hardware resources to different main threads in order to maximize the overall processor throughput. We introduce mechanisms to arbitrate the main threads for hardware contexts and evaluate helper threads for selected multiprogrammed workloads on the SMT processor simulator. We observe the effectiveness of helper threads depends on the hardware context requirement of each individual workload to perform pre-execution. In summary, our experimental results show prefetching helper threads help to boost processor throughput in a multiprogramming environment by reducing latencies in individual application threads.

1.3 Roadmap

The rest of the thesis is organized as follows. In Chapter 2, we provide necessary background about helper threading, multithreading execution paradigms, and previous proposals on data prefetching to better demonstrate how we get to focus on compiler-based pre-execution. Chapter 3 presents an overview of critical issues on constructing effective prefetching helper threads using a compiler. We discuss those issues in an implementation-independent way so that anyone can build a compiler framework to generate helper threads without being limited by the programming language or the target platform. Then the next four chapters, Chapters 4 through 7, introduce our compiler frameworks to construct prefetching helper threads and present the evaluation results. Chapter 4 demonstrates compiler algorithms for our SUIF-based compiler frameworks. We present various ways to design compilers for pre-execution and discuss tradeoffs between different implementations. In Chapter 5, we thoroughly investigate our SUIF-based compiler frameworks in a simulation-based evaluation environment. Chapter 6 examines many aspects of the compiler optimization module for pre-execution in the Intel compiler, and Chapter 7 reports the results of physical experimentation with prefetching helper threads in the Intel's hyper-threaded processor. We also discuss impediments to speedup with helper threads on real silicon and solutions to address those constraints. Chapter 8 investigates the performance of pre-execution in a multiprogramming environment and describes mechanisms to arbitrate multiple main threads for limited hardware contexts. Chapter 9 discusses related works, and Chapter 10 concludes the thesis and discusses possible future directions of the research.

Chapter 2

Background

This chapter provides the necessary background to better understand this thesis. We discuss three issues, *i.e.*, helper threading, multithreading, and data prefetching, and examine how pre-execution has evolved from these already-established research areas. First, we present helper threading, also known as *subordinate threading*, which works as the vehicle to realize pre-execution. While helper threading can be used for other purposes such as exception handling or fault tolerance, in this thesis, we focus on its use for latency tolerance, thereby improving a program performance indirectly. Then we introduce pre-execution as one particular example of multithreading execution paradigms. Especially, we view those multithreading paradigms, *e.g.*, multiprogramming, parallel processing, and pre-execution, as ways to exploit thread-level parallelism that resides in workloads. Finally, we focus on pre-execution's role as an effective data prefetching technique, where delinquent load addresses are *computed*, not *predicted*. Furthermore, pre-execution is a thread-based technique, in which the prefetching threads are *decoupled* from the main computation. Hence pre-execution has more freedom to control prefetching compared to conventional software prefetching techniques, where the instruction sequence for prefetching is integrated into the original program code and its progress is strictly tied with that of the main thread.

2.1 Helper Threading

With the advent of single-chip multithreading processors, in which multiple hardware contexts are closely tied while enjoying small communication delays, helper threading has been recognized as a very promising way to make use of the spare hardware contexts available in those multithreading processors such as the SMT or CMP [22, 14]. In helper threading, one or more helper threads run alongside the main thread to help its execution. Helper threading has two unique characteristics that open up many opportunities. First, in most helper threading techniques, the computed results by the helper threads are never integrated into the shared machine state. Thus helper threads do not contribute to boost the processor throughput, but help the execution of the main thread *indirectly.* Since helper threads never disrupt the main computation, one can apply very aggressive optimizations when constructing helper threads as long as such optimization helps to improve the performance of a program. Second, helper threads are asymmetric in that the execution of helper threads are decoupled from that of the main thread, and their code does not have to be extracted from the original program code. For instance, as to be more elaborated later in this section, one can run exception handler code on helper threads or even implement complicated hardware structures in software. Such unique characteristics of helper threading can greatly help the compiler optimization research area to develop various aggressive optimization algorithms without the need for ensuring the program correctness. Moreover, the asymmetry of helper threads also encourages development of various *helping actions* to help the program execution. In this thesis, we focus on pre-execution, which is the most popular helper threading technique to tolerate latencies in a program's execution. Since helper threading is a relatively new research area, we believe there exist many interesting aspects that have not yet fully investigated. Below, we present some of previously proposed uses of helper threading to assist the execution of a program.

Latency Tolerance. Helper threads can improve the performance of the main computation by tolerating latencies on its behalf. In this case, helper threads are extracted from the original program code and trigger long-latency delinquent events in front of the main thread, thereby hiding the latencies behind useful computations. Some examples include data prefetching [19, 36, 37, 35, 41, 42, 63, 81, 84], instruction prefetching [1], branch outcome precomputation [84], and virtual function call target prediction [61]. This type of helper thread is called *run-ahead thread* [7, 24, 73], and it is crucial to expedite helper threads so that they can trigger delinquent events early enough. In Chapters 4 and 6, we introduce various optimization algorithms to speedup helper threads.

Exception Handling. When an exception occurs, one can run the exception handler code on a helper thread and let the main thread keep making progress by executing instructions that are independent of the faulting instruction [86]. In this case, helper threading effectively decouples otherwise serialized code execution and overlaps the execution of the exception handler code and useful computation. The performance improvement depends on the frequency of exceptions that occur during the program execution and the amount of independent instructions, whose execution can be overlapped with the execution of the exception handler code.

Fault Tolerance. By running the same program redundantly on a helper thread, one can detect and recover faults that occur during a program execution. Since helper thread's execution is totally redundant, this type of helper threading neither contributes to processor throughput nor improves program performance. However, it tolerates faults to preserve the correctness of a program [50, 78].

Implementation of Structures and Algorithms in Software. Using helper threading, one can implement complicated hardware structures or algorithms in software, *e.g.*, a value predictor [85] or branch prediction algorithm [14]. This can ease the hardware complexity of a processor and reduce the testing and validation cost associated with introducing new hardware into the processor design. Note, the code running on helper threads has nothing to do with the original program, but it still helps the execution of the main thread.

2.2 Multithreading

Multithreading has been studied extensively in both academia and industry [3, 39, 65]. Traditionally, people have used multithreading to boost the processor throughput or to improve single program performance. In this thesis, we view multithreading as a way to uncover and exploit thread-level parallelism residing in workloads, and pre-execution effectively exploits a new form of parallelism via construction of much aggressive and speculative helper threads. Below, we examine several well-known multithreading execution paradigms and discuss how those paradigms differ from each other in exploiting thread-level parallelism in workloads.

2.2.1 Multiprogramming

Multiprogramming has been a popular way to utilize multiprocessor systems and increase the overall processor throughput by running multiple independent application threads simultaneously.

Since any communication or synchronization between threads is rarely required in a multiprogramming environment, one can easily extract thread-level parallelism as long as there are enough jobs to process. Hence multiprogramming exploits totally unrelated parallelism from different applications, and the amount of thread-level parallelism is the same as the number of application threads that run together. However, it is sometimes not possible to find multiple independent applications to fill up the available hardware contexts in the system. Moreover, in many multiprocessor systems, critical hardware resources such as the memory bus are often shared and even saturated with large number of actively running threads. Thus feeding more threads into the system may not help to increase the processor throughput. On the other hand, while boosting the overall processor throughput is very important, improving the single program performance is often the most crucial issue under certain circumstances. However, multiprogramming does not help to improve single program performance; even worse, it often sacrifices the performance of each individual program for the overall throughput. In the next two subsections, we discuss ways to speedup a single program via multithreading.

2.2.2 Parallel Processing

One way to improve the performance of a single program is to divide the program into multiple subprograms and run them in parallel on a multiprocessor system. This is called *parallel processing*. Below, we discuss two different parallel processing paradigms; one is conventional parallel processing and the other is recent thread-level data speculation technique.

Conventional parallel processing in multiprocessor systems. Parallel processing has been extensively used in conventional multiprocessor systems and particularly successful for scientific workloads. When partitioned into multiple subprograms, each thread usually runs almost independently, thereby exploiting thread-level parallelism in a single program. In parallel processing, all threads are completely nonspeculative and perform useful computations that contribute to the processor throughput. Therefore, a compiler or programmer must guarantee that no dependence violations occur as a consequence of parallelization, and synchronization among different threads should be properly handled. While conventional parallel processing has been very successful in extracting thread-level parallelism from scientific workloads, it is often not effective for nonscientific codes that account for a large portion of modern workloads. This leads to the invention of thread-level speculation technique, which is discussed below.

Thread-level speculation on chip multiprocessors. With the recent advent of the CMP that supports much finer-grain thread synchronization than conventional multiprocessor systems, a new form of parallel processing, called Thread-Level Speculation (TLS), has been introduced and studied widely in the research community. In TLS, a single program is divided into multiple threads *speculatively*, assuming no memory dependence between threads. By running those multiple computation threads in parallel, TLS exploits thread-level parallelism in the program. In order to guarantee the correct program execution, the computed results by each thread are committed sequentially following the original program order. In addition, the TLS technique requires special hardware support to detect and recover from any dependence violation between memory operations, and adopts the *versioning cache* to hold intermediate results until a thread commits. As long as there is no true dependence between a store operation and later load instructions, those partitioned threads can run simultaneously and exploit thread-level parallelism. Therefore, the amount of parallelism in TLS depends on how frequently memory dependence violations occur, and it also varies based on how the target program is partitioned and the granularity of each thread. Hence it is the compiler's or programmer's responsibility to find the appropriate code regions to partition such that optimal performance can be achieved. However, while TLS tries to improve the performance of nonscientific workloads as well as scientific workloads via speculative parallelization, complicated dependence structures in nonscientific workloads often limit successful exploitation of thread-level parallelism.

2.2.3 Pre-Execution

Pre-execution has received much attention from the research community as a novel latency-tolerance technique. In pre-execution, one or more helper threads run in front of the main computation and trigger long-latency delinquent events that significantly degrade the performance of a program. In particular, we view pre-execution as a new source of extracting abundant amounts of thread-level parallelism from a program. This is because the helper thread code contains much less dependences than parallel processing or TLS for the following reasons. In many pre-execution techniques, helper threads are constructed as a separate instruction sequence and they never affect the correctness of the main computation since all the stores to global variables and heap objects are removed to eliminate any side effects from helper threads. In turn, this effectively breaks all dependences through memory. Since helper threads have separate code and no side effects, one has significant freedom to perform aggressive optimizations on the helper thread code without disrupting the main computation, which would be totally unacceptable in conventional optimizing compilers. One important optimization technique in pre-execution is program slicing, which eliminates noncritical code that is unnecessary to compute delinquent events. After program slicing, many of dependences through noncritical computations are removed from the helper threads, and thus helper threads can trigger those targeted delinquent events accurately even though the target code region is speculatively parallelized. In this sense, pre-execution can be viewed as a new form of *asymmetric* parallel processing technique that overlaps critical events executed in helper threads and noncritical computations performed by the main thread. We observe those kernels that compute delinquent events are easily and correctly parallelizable after program slicing and store removal. Thus compared to TLS that carries all the dependences in a program, pre-execution is a much more powerful technique in exploiting thread-level parallelism.

2.3 Data Prefetching

In Section 2.1, we introduce pre-execution as a general latency-tolerance technique to eliminate stalls in a program execution. In this section, we examine the use of pre-execution for data prefetching and show how exploitation of thread-level parallelism, especially memory-level parallelism, via pre-execution can result in higher program performance. We compare pre-execution with other previously proposed data prefetching techniques and discuss the interesting tradeoffs. In this thesis, we choose the data cache miss as the target delinquent event to be pre-executed since tolerating memory latency is very important for several reasons. Wulf and McKee [83] report the clock frequency of a processor increases by approximately 60% every year whereas the main memory access time improves only about 7% for the same period. Therefore, the speed-gap between a processor and memory subsystem is increasing exponentially, and the memory latency will reach thousands of cycles in the near future. This phenomenon is also known as the *Memory-wall Problem* [83]. Since the memory stall time becomes a dominant factor that limits processor performance in many workloads, there have been plethora of proposals to overcome this challenge by having cache structures in the memory hierarchy, executing instructions out-of-order, or prefetching data cache blocks. Because we believe both cache structures that hold frequently accessed data blocks and out-of-order execution are straightforward and have been extensively studied already, we rather focus on data prefetching and discuss the effectiveness of pre-execution as a novel thread-based prefetching technique.

2.3.1 Prediction-Based Prefetching

One way to issue prefetches for future memory references is to watch the memory address stream and detect repetitive patterns. Based on the identified patterns, one can *predict* the next memory reference address and issue a prefetch accordingly. This type of prefetching is called *prediction-based prefetching*. There have been many proposals that belong to this category. One example is the stride prefetcher [16] where a special hardware structure monitors the address stream and captures a stride between two memory references. A slightly advanced technique is to use a hardware Markov table to hold possible future memory references or repetitive address sequences; this is also known as *Markov prefetcher* [32]. Markov prefetcher does not necessarily require stride behavior in memory access pattern, but it makes use of the correlation between load addresses. These prediction-based hardware prefetching techniques are effective for scientific workloads that exhibit regular memory access patterns. However, for those memory references without any repetitive pattern, usually found in nonscientific workloads, prediction-based prefetching techniques cannot predict future memory references and thus they are useless.

2.3.2 Execution-Based Prefetching

To prefetch for memory references with both irregular and regular memory access patterns, one of the most effective ways is to actually *execute* an instruction sequence, which generates the cache-missing load addresses, earlier than the main computation. This type of prefetching is called *execution-based prefetching*. An execution-based prefetching technique relies on program semantics and extracts kernels that compute load addresses from the original program code. Execution-based prefetching has a wide spectrum of examples from conventional software prefetching, to hardware-based techniques, and recent thread-based techniques, as described below.

Software Prefetching. Software prefetching techniques exploit a loop structure's iterative nature. In other words, the prefetch instructions are inserted into the loop body, but they target load instances in the future loop iterations [49]. Software prefetching is the most primitive form of pre-execution, where only a handful of instructions are executed to compute future memory references. However, since the prefetch instructions are integrated into the main program code, there are some limitations. One is the overhead issue; the newly inserted code may delay the execution of the main thread, which is a problem if the targeted loop does not contain much work. Moreover, the speed of prefetching is strictly tied with the progress of the main thread, and thus the prefetches may not be timely depending on the dynamic cache behavior. In addition, software prefetching does not execute control flow and is often neither aggressive nor speculative. In summary, tight coupling between the main program and prefetching makes software prefetching ineffective to handle pointer-chasing traversals, and the poor execution capability in software prefetching limits its use for complicated address computation patterns, which are often found in integer applications.

Hardware-based Prefetching. Execution-based hardware prefetching techniques [38, 60] extract the traversal information from the program code and implement a prefetch engine, which speculatively traverses the internal representation of the target program structure and issues prefetches earlier than the main computation. We can view such prefetch engine as a special-purpose hardware context, which performs a small set of operations and issues prefetch requests for the delinquent loads. Since a prefetch engine supports only a few types of primitive operations such as `add` or `shift`, it is also ineffective for memory references that involve control flow and complex computations. Moreover, introducing a new special hardware for prefetching requires significant efforts to design, test, and validate such hardware structures, and thus hardware-based prefetching is often impractical from the processor designer's perspective unless it provides reasonable performance gain across a variety of workloads in general.

Thread-based Prefetching. Thread-based prefetching [6, 20, 19, 36, 37, 35, 42, 63, 84] is a greatly advanced and generalized form of hardware-based prefetching in that it makes use of the general-purpose hardware contexts that are available in multithreading processors, instead of introducing yet another special hardware structures dedicated to prefetching only. With the full functionality of a processor, a thread-based prefetching technique can perform any computations including both data and control flow in order to generate memory addresses for delinquent loads. Another powerful advantage of thread-based techniques is the progress of helper threads is not limited by that of the main thread due to decoupling of the threads. Depending on how helper threads are constructed, we can roughly group thread-based prefetching techniques into compiler-based, linker-based, dynamic optimizer-based, and hardware-based approaches as shown in Figure 1.1.

Chapter 3

Overview of Compiler Algorithms for Pre-Execution

We group some critical issues, which should be considered to construct prefetching helper threads using a compiler, into four categories: identifying targets, improving effectiveness of pre-execution, minimizing ineffectiveness of pre-execution, and preserving program correctness. In this chapter, we provide a high-level overview of the compiler algorithms and necessary architectural support to address those issues in an implementation-independent way. Later in this thesis, however, we show how those issues are addressed in the two different compiler frameworks and provide the implementation details; Chapter 4 describes the compiler algorithms in the SUIF-based compiler frameworks, and Chapter 6 discusses the pre-execution optimization module in the Intel compiler.

3.1 Identifying Targets

The first step in constructing helper threads is to identify both the target delinquent loads and surrounding code regions, for which helper threads will be constructed. In pre-execution, choosing the right set of targets has a huge impact on the effectiveness of the technique. This section examines several possible options to identify targets to be pre-executed and discusses the tradeoffs.

3.1.1 Identification of Delinquent Loads

What to identify? Choosing the right target is very important in pre-execution to maximize the performance improvement with the technique. As Section 3.3 will show, pre-execution consumes processor resources, thus incurring some runtime overhead. In other words, helper threads occupy hardware contexts, cache ports, memory bus, execution units, and other shared or partitioned hardware resources in the SMT processor that might have been better used by computation threads. To minimize the detrimental effect of such resource contention problem, we must target only those delinquent loads that will provide a performance improvement when pre-executed.

In identifying delinquent loads, interesting tradeoffs exist between the cache-miss coverage and pre-execution effectiveness. As more loads are identified as delinquent, the cache-miss coverage of pre-execution increases and the helper threads possibly eliminate more memory stalls. However, to accommodate a larger number of delinquent loads, more and larger code regions are identified as the target pre-execution regions. In turn, this results in more frequent invocations of helper threads, and each helper thread will execute more instructions to compute larger set of delinquent loads, potentially incurring more runtime overhead. Furthermore, as to be more elaborated later, covering larger code regions in helper threads reduces the effectiveness of other optimization techniques due to the remaining dependences in the code. Therefore, it is crucial to have certain algorithms to carefully identify delinquent loads, thereby maximizing pre-execution effectiveness.

How to identify? We introduce three methods to identify delinquent loads: two using profile information and one relying on static compiler analysis. In the profile-based methods, we find two pieces of information play a key role; the cache-miss profiles indicate which static loads in a program incur most of the cache misses, and the associated cycle counts help to estimate whether those cache misses are critical to the program performance and thus worth targeting. Below, we present the three ways to identify the three ways to identify the delinquent loads in a program and discuss the pros and cons of each method.

1. **Offline profile-based identification**. In offline profile-based identification, one collects memory and cycle profiles using a cache simulator or specialized tools such as the Intel VTune® Performance Analyzer [79]. Offline profiling is widely used in the research community to examine program's behavior. In this method, profile information is acquired over the entire or part of the program execution, and it does not introduce any runtime overhead to the actual program run since the profiling is performed offline. However, this method requires profiling tools and a separate profile acquisition phase during compilation. Another drawback is it may not accurately reflect dynamic cache behavior if the actual program inputs or target platforms differ from those used to acquire the profiles.

2. **Online profile-based identification**. Cache-miss or cycle information can be collected during the actual program run. Collins *et. al.* adopt this method to identify delinquent

loads dynamically [19]. In online profile-based identification, one can make use of dynamic information of a program and capture the dynamic program behavior at runtime. Moreover, neither additional offline profiling nor static compiler analysis phase is required in compilation steps. However, this approach introduces some overhead to the actual program run due to the online profiling and it sometimes requires special hardware support to collect and store profile information. In addition, due to the runtime overhead and limited hardware space to hold profiles, the information that the online profiling collects may be not as much as that of the offline profiling.

3. **Static identification using compiler analysis**. Ozawa *et. al.* introduce very simple heuristics to identify delinquent loads; index-array reference or pointer dereference is likely to incur cache misses [53]. Panait *et. al.* demonstrate more advanced algorithms and show it is possible to cover a large portion of cache misses by identifying only a small number of static loads as delinquent [57]. One advantage of such a method based on compiler analysis is that it requires neither additional profiling phase nor any tools to collect profiles. The static algorithms can be easily integrated into compilers as an additional optimization phase. Moreover, like offline profiling, this method has zero impact on the execution time of the actual program run. However, static algorithms have some limitations to identify dynamic events such as the cache miss or clock-cycle, and thus the degree of selectivity is often poor. For instance, in [57], their static algorithms successfully capture load instructions that account for most of the cache misses in a program. However, significant amounts of loads are identified as delinquent by the algorithms, but they rarely incur cache misses, producing many falsely identified delinquent loads. Hence when the pre-execution overhead is critical, this method may be unacceptable.

In this thesis, we evaluate all three methods. First, our SUIF-based aggressive compiler uses offline cache-miss profiles acquired from a cache simulator as demonstrated in Section 4.1.1, while our SUIF-based reduced compiler relies on simple heuristics to identify delinquent loads by static compiler analysis, shown in Section 4.2.4. Section 5.3.3 compares the performance difference between these two methods using a simulation-based evaluation environment. On the other hand, the Intel compiler acquires offline memory profile information using the VTune, which is described in Section 6.2.1. Finally, Section 6.3 introduces user-level library routines that support light-weight online profiling.

3.1.2 Identification of Target Code Regions

Candidates for target regions. Once delinquent loads are identified, we need to pick code regions that surround those selected delinquent loads to construct helper threads. We identify a few conditions to determine good candidates for a pre-execution region; 1) reasonable amount of static code within the code region is sliced out so that helper threads can acquire enough speed advantage over the main thread, 2) the code region is easily parallelizable and there rarely exists any dependence chains within the parallelized codes, and 3) once the main thread enters the code region, it runs for a long time until exiting the region in order to amortize the thread spawning and synchronization overhead. Program structures that satisfy these conditions include loops, recursive call structures, or program continuation such as function calls, after which the program code is guaranteed to be executed. We are especially interested in loops for the following reasons. First, delinquent loads with the greatest impact on the program performance execute a large number of times over the lifetime of the program execution; except for hard disk accesses, we cannot think of any event that has huge impact on performance, but occurs only infrequently. Hence the most important delinquent loads are likely to be inside loops. Moreover, by nature, loops are easily parallelized and provide a good source of thread-level parallelism; one can divide loops into multiple threads. Therefore, we choose loops as the target pre-execution regions in this thesis.

Loop selection. Even for a single delinquent load, there are usually multiple loop-nests that encompass the load, *e.g.*, inner-most loop-nest, second-level outer loop-nest, third-level outer loop-nest, and so on. To avoid redundancy in prefetching and reduce the pre-execution overhead at runtime, we allow only one code region to be selected for each delinquent load. In order to choose a single pre-execution region among multiple loop-nests, we need a decision algorithm so that pre-execution can provide an optimal performance gain. There exist tradeoffs between selecting inner-loops and outer-loops. Usually, inner-loops contain fewer dependence chains across loop iterations and thus when parallelized, the helper threads can still generate accurate addresses for the delinquent loads. However, if the selected inner-loop iterates just a small number of times, a portion of the prefetching gain can be lost due to the spawning cost of helper threads. Furthermore, prefetching may be ineffective since the main thread exits the loop even before the helper threads get ahead. Therefore, the loop-trip count information must be taken into account to decide which loop-nest to choose for the pre-execution region. In Section 4.1.1 and 6.2.1, we demonstrate how both our SUIF-based compiler and the Intel compiler choose the right loops to achieve the best performance gain with pre-execution.

3.1.3 Code Cloning

Once the target pre-execution regions are identified, we construct helper threads for each selected code region. We introduce two methods to generate helper threads; one is sharing the same code with the main thread, and the other is creating a separate code region for the helper thread. In Luk's work [42], the helper threads execute the same code as the one that the main thread executes. This kind of code sharing helps to reduce the burden on the instruction cache. However, it also prevents performing aggressive optimization and code-transformation techniques on the helper thread code in order to preserve the correctness of the program execution. Because of the limited opportunities for optimization in code sharing, most pre-execution techniques adopt *code cloning* and create separate code for the helper threads. In code cloning, instructions or part of the program code necessary to compute the delinquent load addresses are extracted from the main thread code and stored as a separate code. This method has a few advantages over the code-sharing approach. First, various optimizations can be performed on the cloned code to generate speculative and aggressive helper threads. Usually, optimization algorithms introduced in this thesis are, by all means, neither valid nor safe from the conventional optimizing compiler's perspective. Nevertheless, we observe our aggressive compiler optimizations rarely break correctness of the helper thread code, and the constructed helper threads still generate accurate addresses for most of the delinquent loads. This is mainly because those optimization techniques are only performed on the kernels that compute the delinquent loads, not for the whole program code within the pre-execution region. Finally, although code cloning increases the instruction working set size, we find no evidence that instruction cache miss is a performance bottleneck for the benchmarks used in our experiments.

3.2 Improving Effectiveness of Pre-Execution

In pre-execution, helper threads should be able to get ahead of the main thread so that they issue prefetches early enough and the main thread enjoys cache hits when it actually executes the targeted loads. In this section, we discuss four important issues to speedup the execution of the helper threads and improve the effectiveness of pre-execution. First, when constructing helper threads for a target code region, we extract only those instructions that are necessary to compute the delinquent loads and remove everything else. This technique is called *program slicing* and it helps the helper threads get ahead by executing fewer instructions than the main thread. Another way to expedite helper threads is to eliminate blocking by converting delinquent loads to the nonblocking prefetch instructions. This is called *prefetch conversion.* By performing prefetch conversion, we prevent helper threads from being stalled on the delinquent loads, thereby making forward progress. On the other hand, we also parallelize the target code region and assign each loop iteration to different helper threads, which is called *loop parallelization.* Although each individual helper thread does not run faster by parallelizing loops, multiple helper threads as a group can. These three techniques enable pre-execution to uncover a greater degree of memory-level parallelism and allow the helper threads to overlap more memory accesses simultaneously. In summary, program slicing cuts dependence chains through the value computations by eliminating unnecessary instructions, prefetch conversion removes blocking loads so that each individual helper thread can issue prefetches even faster, and loop parallelization helps multiple helper threads as a group to issue more prefetches simultaneously. In addition to these three techniques to speedup helper threads, there is one more important issue to improve pre-execution effectiveness: *synchronization*. Synchronization prevents *run-away* helper threads by forcing the main thread and the helper threads to keep the right distance so that useful cache blocks are not evicted out of the cache. This section discusses each of these issues to improve the pre-execution effectiveness.

3.2.1 Program Slicing

For pre-execution to be effective, helper threads should issue prefetches early enough so that when the main thread executes targeted delinquent loads, the corresponding cache blocks are found in the cache. Program slicing helps to speedup the helper threads by filtering out unnecessary code that is not required to compute the delinquent load addresses. In addition to cutting out unnecessary instructions, program slicing also plays another very important role to improve the effectiveness of pre-execution; it breaks dependence chains along with the value computations that have nothing to do with the address computation of delinquent loads.

One can perform program slicing in several ways. In a hardware approach or post-pass binary analysis to construct helper threads, instructions producing register values that affect computation of delinquent loads are extracted from a pre-execution region. On the other hand, in our compiler-based approach, we examine three different methods to enable program slicing. Section 4.1.2 demonstrates how we perform program slicing in our SUIF-based aggressive compiler using a publicly available slicing tool, called Unravel [45]. Section 4.3 introduces our SUIF-based reduced compiler that does not use an external slicing tool, but relies on the backend compiler's dead-code elimination optimization. Finally, Section 6.2.2 describes program slicing in the Intel compiler as a separate compiler optimization phase.

3.2.2 Prefetch Conversion

Prefetching helper threads, by design, incur a large number of cache misses. Unless we take proper action to reduce blocking, the helper threads may not make sufficient forward progress, resulting in ineffective prefetching. Thanks to program slicing's instruction removal, a loaded value of a certain delinquent load is not consumed by other instructions in the helper thread.

Once we identify such loads, we convert them into the nonblocking prefetch instructions. The biggest advantage of prefetch conversion comes from the fact that in many processor designs, the nonblocking prefetch instructions can immediately leave the load-store queue, which is responsible for the memory disambiguation and value forwarding from store to load instructions. Thus, with prefetch conversion, the helper threads can keep continuing their execution without being stalled. In our SUIF-based compiler frameworks, we examine two ways to enable prefetch conversion. As demonstrated in Section 4.1.2, our aggressive compiler relies on the program slicing analysis to identify the delinquent loads that can be converted into prefetches. In Section 4.2.2, we also present a simple heuristics for prefetch conversion in our reduced compilers. Section 5.2.3 reports the experimental results that show the impact of prefetch conversion on the program performance.

3.2.3 Loop Parallelization

Program slicing and prefetch conversion speedup individual helper threads by eliminating unnecessary code and blocking. On the other hand, loop parallelization helps multiple helper threads to run faster by overlapping more memory accesses. While loop parallelization is an effective way to extract thread-level parallelism from a program, not all loops need to be parallelized when constructing helper threads. For instance, some loops contain totally serial dependence chains across the loop iterations, and thus parallelizing such loops breaks the correctness of the resulting code. For those loops with loop-carried dependences, we can only expect the speed advantage of a single helper thread over the main thread by program slicing and prefetch conversion. On the other hand, prefetch conversion sometimes eliminates all blocking within the target code region, converting every single delinquent load in the helper thread to a prefetch. Considering the overhead associated with loop parallelization such as managing multiple threads and supporting synchronization between threads, it is often more advantageous not to parallelize the target code region in such a case. However, if blocking loads are still remaining even after prefetch conversion, we parallelize the target code region.

In this thesis, we examine two parallelizing schemes, *i.e.*, DoAll and DoAcross. When the loop induction variable is updated arithmetically, *e.g.*, `i++`, we choose the DoAll scheme where all helper threads are symmetric and execute different loop iterations in a round-robin fashion. On the other hand, if the loop induction variable is updated by dereferencing a pointer, *e.g.*, `ptr=ptr->next`, we choose the DoAcross scheme to parallelize the target code region. In this situation, even though the execution of each loop iteration is completely serialized along with the pointer chain, we can still overlap the loop-body executions across different loop iterations. In the DoAcross scheme, we construct a single helper thread, called *backbone thread*, to compute the pointer chain, and also construct multiple helper threads, called *rib threads*, to execute the loop body. Our DoAcross scheme is very similar to the Chaining scheme introduced by Collins *et. al.* [19], where all helper threads execute the identical code, but each helper thread passes the next pointer value to the next available helper thread. Section 4.1.2 discusses the details of our loop parallelization technique, and Section 5.2.3 examines how the loop parallelization contributes to the performance improvement. Finally, Section 5.2.4 compares the different thread initiation schemes that are applied to the same target region and evaluates our algorithm to select a proper scheme to initiation the helper threads.

The loop parallelization schemes employed in our compiler frameworks are neither valid nor safe under all circumstances. In other words, our compilers parallelize loops *speculatively*. Hence it is always possible for our loop parallelization to break the correctness of a code and generate helper threads that issue inaccurate prefetches. However, since many dependences outside the cache-miss kernels get removed by program slicing, we observe our loop parallelization generates the correct helper threads most of the time, which is further discussed in Section 5.2.6. In summary, there is great potential to uncover abundant memory-level parallelism in the cache-miss kernels, and pre-execution is very effective in exploiting such parallelism.

3.2.4 Synchronization

Although synchronization does not expedite helper threads like the above three techniques, it is still crucial to ensure effective pre-execution. To be effective, the helper threads must keep the right run-ahead distance, or *prefetch distance*, with respect to the main thread. If the helper threads run too far ahead of the main thread, they bring cache blocks into the cache too early, allowing prefetched cache blocks to be evicted before being accessed by the main thread. On the other hand, if the helper threads run just barely ahead of the main thread, the prefetched cache blocks may not arrive sufficiently early, causing the pipeline stall. Occasionally, the helper threads run behind the main thread. In such situation, running helper threads is totally useless and it may even degrade the performance of the main computation.

To keep the right prefetch distance, we synchronize the helper threads with the main thread by tracking the relative distance between the threads. Using a thread synchronization mechanism, we detect when the helper threads run too far ahead of the main thread and suspend them to avoid the cache pollution. In our SUIF-based compiler frameworks, we examine two thread synchronization mechanisms, *i.e.*, hardware- and software-based mechanisms. Section 4.1.2 compares the performance impact of these two synchronization mechanisms in a simulation-based evaluation environment. We also evaluate two thread synchronization mechanisms in a real physical system. Section 6.1.2 discusses using the OS API and a hardware mechanism to synchronize between the main thread and the helper thread in the hyper-threaded processor, and Section 7.1.4 provides the experimental results to show the impact of these synchronization mechanisms on performance.

3.3 Minimizing Ineffectiveness of Pre-Execution

When performing pre-execution on an SMT processor, the helper threads share the processor resources with the main thread. Due to the contention in the hardware resources, the progress of the main computation is often limited if the helper threads do not help. Moreover, the helper threads become speculative after the aggressive compiler optimizations introduced in the previous section, which means that prefetches issued by the helper threads can be inaccurate and may not help to reduce the cache misses incurred in the main thread. In such a situation, performance degradation of the main thread by pre-execution becomes even more severe. There are many cases when it is more desirable not to run helper threads; a) helper threads generate inaccurate memory addresses that are not accessed by the main thread, b) helper threads issue correct prefetches but they run behind or barely ahead of the main thread, and thus those issued prefetches are not useful, c) helper threads are still active and issue prefetches even after the main thread has already left the pre-execution region, and d) some critical hardware resources such as the instruction fetch unit, functional units, and reorder buffer may not be optimally distributed to the main thread and the helper threads, thereby reducing the performance gain from pre-execution. This section introduces two mechanisms to minimize the ineffectiveness of pre-execution and reduce the runtime overhead incurred by pre-execution.

Runtime performance monitoring and dynamic throttling. A program behavior changes dynamically during execution. For instance, a load instruction, which severely suffers from cache misses in general, may incur only a negligible amount of cache misses during certain time periods. If helper threads are launched to pre-execute the load instruction for such time phases with few cache misses, they just consume processor resources and possibly degrade the performance of the main thread. To avoid such detrimental situations, we can monitor the dynamic behavior of the helper threads at runtime to estimate the effectiveness of pre-execution. When we detect pre-execution is not helpful for improving the main thread's performance, we can simply terminate the currently running helper threads or prevent the future invocations. To implement a runtime system that monitors the program behavior and throttles the helper threads dynamically, we should consider several issues. First, since the main purpose of having such runtime system is to reduce the performance degradation with pre-execution, the system itself should not introduce an additional runtime overhead. To address this issue, Section 6.3 introduces a light-weight performance monitoring tool, called EmonLite, and discusses the relevant issues regarding its implementation and usage model. Second, we need to decide which performance events most accurately reflect the effectiveness of pre-execution for a program and thus should be monitored; possible performance events include clock cycle, cache miss, bus utilization, and retired instruction count, but other events can serve the role as well. Third, when runtime profiles are given, we should be able to decide whether currently running helper threads or future helper thread invocations must be avoided. Although we do not actually implement such mechanism to dynamically throttle helper threads in this thesis, we estimate the potential performance improvement with dynamic throttling by assuming a hypothetical perfect throttling mechanism in Section 7.1.5.

Terminating useless helper threads after exiting pre-execution region. Occasionally, helper threads remain active and issue prefetches even after the main thread exits a targeted pre-execution region. Possible scenarios for this include a) helper threads simply run behind the main thread, b) some speculative optimization techniques result in producing inaccurate helper threads that follow wrong control paths, and c) the target pre-execution region is parallelized without disrupting correctness of helper threads, but only one helper thread executes the termination condition correctly and the others are not aware of it, *e.g.*, exiting with `break` statement in a loop. All these scenarios describe totally useless pre-execution, and thus helper threads must be terminated upon detection of such cases. Our solution is to suspend all active helper threads immediately after the main thread leaves the target pre-execution region. To actually implement this, one can rely on certain ISA support so that when the main thread executes a special instruction, all active helper threads are suspended. On the other hand, the main thread may signal the OS at the loop-exit point to terminate those useless helper threads. In our SUIF-based compiler frameworks, we assume a `kill` instruction to do the job (see Section 4.1.3).

3.4 Preserving Program Correctness

Thus far, we discuss issues to improve the performance of a program using prefetching helper threads. Another important issue is that the helper threads should never disrupt the correctness of the main computation. To guarantee the correct execution of a program, we consider two issues; helper threads cause no side effects on the main thread, and exceptions signaled from the helper threads should be properly handled.

3.4.1 Removing Side Effects

Helper threads can cause side effects on the main computation through the memory or hard disk. We identify two sources of the side effects from the helper threads; one is the store operation to global variables or heap objects, and the other is system calls. To eliminate the side effects, one can use a hardware mechanism [42], where store operations in the helper threads are still allowed, but the store values are written to a hardware scratch pad, not to main memory directly. By keeping stores in the helper threads with such hardware support, we guarantee that the helper threads generate accurate addresses for the memory references by preserving dataflow through memory. Another approach to eliminate side effects is to rely on the compiler to remove all store operations except for those to the helper thread's stack, and system calls from the helper thread code. Such compiler-based approach does not require any hardware support. However, in cases where values necessary to compute the memory addresses are passed through memory, generated addresses can be wrong, causing inaccurate prefetching.

3.4.2 Handling Exceptions

After speculative loop parallelization and store removal, the helper threads may follow a wrong path and generate illegal memory addresses. In this case, an exception occurs and the whole process that contains both the main thread and helper threads will be terminated. To avoid such an undesirable situation, we handle exceptions signaled by the helper threads. There are two possible solutions to support exception handling, one by hardware and the other by software. In a hardware approach, when an exception is signaled in a hardware context used by a helper thread, the faulting helper thread is terminated immediately without any OS involvement. This is a good choice when the helper threads are invisible to the OS. On the other hand, in a software approach, upon detecting an exception signaled by a helper thread, the control of the thread is handed over to the OS to terminate the faulting helper thread and properly manage the hardware context. Since this approach involves invoking an OS thread, it may be costly. However, if exceptions do not occur frequently, the cost may be affordable.

Chapter 4

SUIF-Based Compiler Frameworks for Pre-Execution

The previous chapter provides a high-level overview of the compiler algorithms and discusses the relevant issues for pre-execution. In this chapter, we demonstrate how such algorithms are actually implemented in our SUIF-based compiler frameworks to construct effective prefetching helper threads. We also show how to address those related issues to perform pre-execution on a time-accurate simulator, which models the research SMT processor. We prototype five compiler frameworks: one aggressive and four reduced ones. Our aggressive compiler framework implements several powerful algorithms to optimize helper threads using an external program slicer, *i.e.*, Unravel, and offline profile information for cache behavior and loop-trip counts. While our aggressive compiler framework provides a good performance gain, it requires many compilation steps, thereby complicating the design of such a compiler. To simplify the compiler implementation, we propose four reduced compiler frameworks in which Unravel and the offline profiles are selectively eliminated and replaced with static compiler algorithms to perform prefetch conversion and target identification. This chapter is organized as follows. Section 4.1 introduces the compiler algorithms for the aggressive compiler framework, and Section 4.2 shows the design of the reduced compilers. After discussing our compiler optimization algorithms for pre-execution, Section 4.3 addresses the implementation issues and shows how our algorithms are integrated into the SUIF compiler infrastructure to build compiler frameworks to generate helper threads. We also discuss the necessary architectural support that our SUIF-based compiler frameworks assume from the target SMT processor.

4.1 Compiler Algorithms for SUIF-Based Aggressive Framework

We first describe the compiler algorithms used in our SUIF-based aggressive compiler framework (*aggressive compiler* for short). Following the classification in Chapter 3, we group algorithms into four categories, *i.e.*, algorithms to identify target delinquent loads and pre-execution regions, to improve the effectiveness of pre-execution, to minimize the ineffectiveness of pre-execution, and to guarantee the correctness of the main computation.

4.1.1 Identifying Targets

To construct prefetching helper threads, we need to know which static loads in a program incur most of the cache misses, accounting for a significant portion of the program's execution time. Once the delinquent loads are identified, we choose the surrounding code regions for which helper threads are constructed.

Delinquent Load Identification

In our aggressive compiler, we rely on *summary cache-miss profiles* [2] to characterize the memory behavior of a program. To perform such profiling, we use a cache simulator derived from the SimpleScalar toolset's *sim-cache*. Our cache simulator is a functional (not time-accurate) simulator that issues one instruction at a time in the original program order, and it counts the number of first- and second-level cache misses (*L1 misses* and *L2 misses* for short, respectively) for every static load in a program. This cache-miss profiling step is performed as a separate program run, and thus it does not affect the execution of the actual run for performance measurement. After collecting cache-miss profiles, we sort all static loads in descending order by the L1 miss count.[1] Starting from the most cache-missing load at the top of the list, we keep selecting loads until the accumulated cache-miss count exceeds 90% of the total L1 misses. Once identified, however, all delinquent loads are given the same priority and treated equally regardless of the cache-miss count. As shown in Table 5.3 in Section 5.2.1, we find only a small number of static loads account for a large portion of the total cache misses (a similar observation has been made by Abraham

[1] In this thesis, the SimpleScalar-based evaluation targets L1 misses whereas evaluation in a physical system targets L2 misses. This is because eliminating L1 misses provides some performance improvements on our simulator while targeting L1 misses does not make noticeable difference in the hyper-threaded processor and it sometimes pre-executes too many loads in a program, thereby increasing the runtime overhead. In summary, whether to target L1 misses or L2 misses depends on the underlying system and involves several factors to consider such as the latency of the delinquent events, hardware resource management in the target platform, or runtime overhead of running helper threads.

et. al. [2]). The final list of the selected static loads is fed into the compiler framework as the delinquent-load set so that helper threads can be constructed to cover all the loads in the list.

Pre-Execution Region Selection

Once delinquent loads are identified, we define a *pre-execution region* for each delinquent load. A pre-execution region encompasses one or more delinquent loads and limits the scope of pre-execution within the code region. As already mentioned in Section 3.1.2, we select loops as the candidate for pre-execution regions; in particular, we only consider the inner-most and next-outer loops in the global loop-nest graph of a program. In other words, if a delinquent load is located outside two looping nests, it is simply discarded from the delinquent-load list and not covered by pre-execution. Between choosing the inner-most loop and the next-outer loop, intricate tradeoffs exist. On the one hand, the likelihood of loop-carried dependences increases as the helper threads execute more distant codes. Hence selecting the next-outer loop for a pre-execution region tends to increase the possibility to disrupt the correctness of the loop parallelization, which will be discussed later in Section 4.1.2. On the other hand, if the helper threads are spawned targeting the inner-most loop, the benefit from pre-execution can be reduced by the thread spawning cost and even nullified if the loop does not contain enough work. To roughly estimate the amount of work in a loop, we rely on simple heuristics using the loop-trip count information; the larger the number of iterations a loop executes, the more work it contains. Therefore, our aggressive compiler uses the loop-trip count profiles to help select the best pre-execution regions.

To identify all inner-most and next-outer loops in a program, we construct a global loop-nest graph, G_L. G_L is a Directed Acyclic Graph (DAG) in which nodes represent loops and edges denote the loop-nesting relationships; an upper node encompasses the one located below in the graph. The DAG specifies nesting information between all loops in the entire program taking into consideration nesting across procedure calls as well as within procedures.[2] To choose an optimal loop nest for the pre-execution region, our loop-selection algorithm examines the amount of work in all inner-most

[2]We use the program's procedure-call graph to capture the inter-procedure loop-nesting relationships. However, we do not account for nesting across indirect calls.

```
Given: Global loop nest graph, $G_L$
       Loop iteration count profiles
Compute: Pre-Execution Region Set, P

 1: P = $\Phi$;
 2: for each loop L in $G_L$ from inner-most to outer-most {
 3:   if (level(L) == INNER_MOST) {
 4:     if (iteration_count(L) $\geqslant$ 25)
 5:       P = P $\cup$ {L};
 6:   } else {
 7:     if {P $\cap$ nested_loops(L) == $\Phi$)
 8:       P = P $\cup$ {L};
 9:   }
10: }

11: for each inner-most loop L in $G_L$ {
12:   if (P $\cap$ outer_loops(L) == $\Phi$)
13:     P = P $\cup$ {L};
14: }
```

Figure 4.1: Algorithm to compute the set of pre-execution regions, P. Φ denotes the empty set. To estimate the loop work, we rely on offline profiles for loop-trip counts.

loops that contain one or more delinquent loads. Based on the loop-trip count profiles collected offline, if the inner-most loop iterates a large enough number of times, we assume thread spawning cost will be amortized and select the loop as a pre-execution region. Otherwise, we find the next-outer loop and select it. However, a next-outer loop sometimes contains more than one inner-most loop, and both the inner-most and next-outer loops can be selected for pre-execution. Since we do not allow nested pre-execution to avoid redundant prefetching and additional runtime overhead, we simply discard the next-outer loop from the pre-execution region set upon encountering such a situation.

Figure 4.1 presents our algorithm to compute the set of pre-execution regions, P, given the global loop-nest graph, G_L, and loop-trip count profiles. We first start with an empty set P (line 1) and visit all loops in G_L in inner-most to outer-most order (line 2). If a loop is at inner-most level and contains delinquent loads (line 3), we check whether the loop-trip count is greater than or equal to a certain number to see if the loop has enough work (line 4). We choose 25 loop iterations as the threshold loop-trip count and observe this works well for most pre-execution regions we have encountered in our study.[3] If the loop-trip count exceeds the threshold value, the loop is

[3]However, the threshold loop-trip count should be large enough to amortize the pre-execution start-up cost, and thus it may vary depending on the overhead of the thread synchronization mechanisms of the target platform.

selected as a pre-execution region and added to P (line 5). Otherwise, we choose the next-outer loop unless it nests loops that are already selected as pre-execution regions, in order to avoid nested pre-execution (lines 7 and 8). After all inner-most and next-outer loops in the program have been visited, it is possible that some inner-most loops that iterate fewer than 25 times are excluded from the pre-execution region set because a *sibling loop* in graph G_L was added to P, thus preventing their common next-outer loop from being selected as a pre-execution region. To ensure all delinquent loads are pre-executed, we visit all inner-most loops again (line 11) and select loops that contain delinquent loads and whose next-outer loop is not chosen as a pre-execution region (lines 12 and 13). However, since all loops selected in this second phase of the algorithm have a small loop-trip count, it is possible to degrade the program performance by targeting such small loops. Generally, we find the increase in the cache-miss coverage outweighs the additional overhead incurred by pre-executing these small loops.

Code Cloning

Once pre-execution regions are selected, we construct helper threads. The best way to generate accurate prefetches for delinquent loads is to execute code that is very similar to the original program code; hence we adopt code cloning when constructing helper threads. Once delinquent loads and the surrounding pre-execution regions are identified, our compiler clones each target code region and creates a separate procedure. If the target code region consists of multiple procedures, all the procedures are cloned as well. To clone a code region, the SUIF provides a library routine, `clone()`, that clones an object in a program and manages the symbol table properly. These newly generated procedures are used as the helper thread code. As to be illustrated in Section 4.1.3, when the main thread encounters a pre-execution region, it retrieves the function pointer of the corresponding cloned procedure so that helper threads can indirectly jump to the procedure to initiate pre-execution. In addition to ensuring accurate prefetching, code cloning provides another very powerful advantage over code sharing; it allows us to perform aggressive optimizations on the cloned code. In our compiler frameworks, all the optimization techniques such as program slicing, prefetch conversion, and loop parallelization would have been infeasible if the main thread and helper threads had shared the same code as in Luk's Software-Controlled Pre-Execution [42].

4.1.2 Improving Effectiveness of Pre-Execution

Once the target code regions are cloned, we consider ways to speedup the helper threads so that they can trigger cache misses early enough in front of the main computation. We introduce four algorithms used in our aggressive compiler to improve the effectiveness of pre-execution. First, program slicing eliminates all noncritical code in the helper threads that is not required to compute the delinquent load addresses. Then prefetch conversion, which is driven by the program slicing analysis, removes blocking in the helper threads. Note, these two optimization techniques are to speedup the individual helper threads. On the other hand, loop parallelization partitions the target pre-execution region into multiple helper threads to run them in parallel. Hence multiple helper threads as a group can run faster than the main thread. Finally, thread synchronization guarantees the helper threads do not run too far ahead of the main thread, thereby avoiding eviction of useful cache blocks due to excessive prefetching.

Program Slicing

We first discuss program slicing to remove noncritical code in helper threads. We present a powerful program slicer, Unravel, and its analyses for extracting slices, and then demonstrate our modifications to Unravel so that it can generate slices customized for pre-execution.

Unravel. To perform program slicing, our aggressive compiler uses Unravel, a publicly available program slicer for ANSI C language developed from the National Institute of Standards and Technology (NIST).[4] Unravel is a software evaluation tool designed to assist programmers with debugging and testing programs. It consists of two components, an *analyzer*, which parses all .c and .h source files in a program and constructs a program dependence graph (PDG) [26], and a *slicer*, which traverses the PDG iteratively, performing data and control flow analyses to extract

[4]Source code for Unravel can be downloaded from http://www.itl.nist.gov/div897/sqg/unravel/unravel.html.

the program slice.

Basic Analysis: Eqs. 4.1 and 4.2 present the basic program slicing algorithm performed by Unravel's slicer.

$$S_{<m,v>} = \begin{cases} S_{<n,v>} & \text{if } v \notin defs(n) \\ Sdef_{<n,v>} & \text{otherwise} \end{cases} \tag{4.1}$$

$$Sdef_{<n,v>} = \{n\} \bigcup \left(\bigcup_{x \in refs(n)} S_{<n,x>} \right) \bigcup \left(\bigcup_{y \in refs(k)} \quad \bigcup_{k \in control(n)} S_{<k,y>} \right) \tag{4.2}$$

In Eq. 4.1, $S_{<m,v>}$ denotes the program slice for the slice criterion $<m, v>$, or variable v at statement m. The algorithm examines all statements n that are predecessors of m. If n does not assign v, we omit n from the slice and recursively evaluate $S_{<n,v>}$, the program slice for variable v at statement n. Otherwise, if n assigns v, we follow Eq. 4.2. In this case, we add n to the slice and recursively evaluate the program slice for all referenced variables x used to compute v at statement n (the first two terms in Eq. 4.2). This captures those statements that affect the dataflow to statement n. In addition, we also recursively evaluate the program slice for all referenced variables y at all statements k, which control the execution of n, denoted by the *control*(n) function (the last term in Eq. 4.2). This captures those statements that affect the control flow to statement n.

Advanced Analysis: In addition to the basic slicing algorithm presented in Eqs. 4.1 and 4.2, Unravel's slicer also performs several advanced analyses to provide more accurate dependence information, thereby detecting fewer false dependences and improving the quality of extracted program slices. Specifically, Unravel's advanced analyses address the following features found in the C programming language:

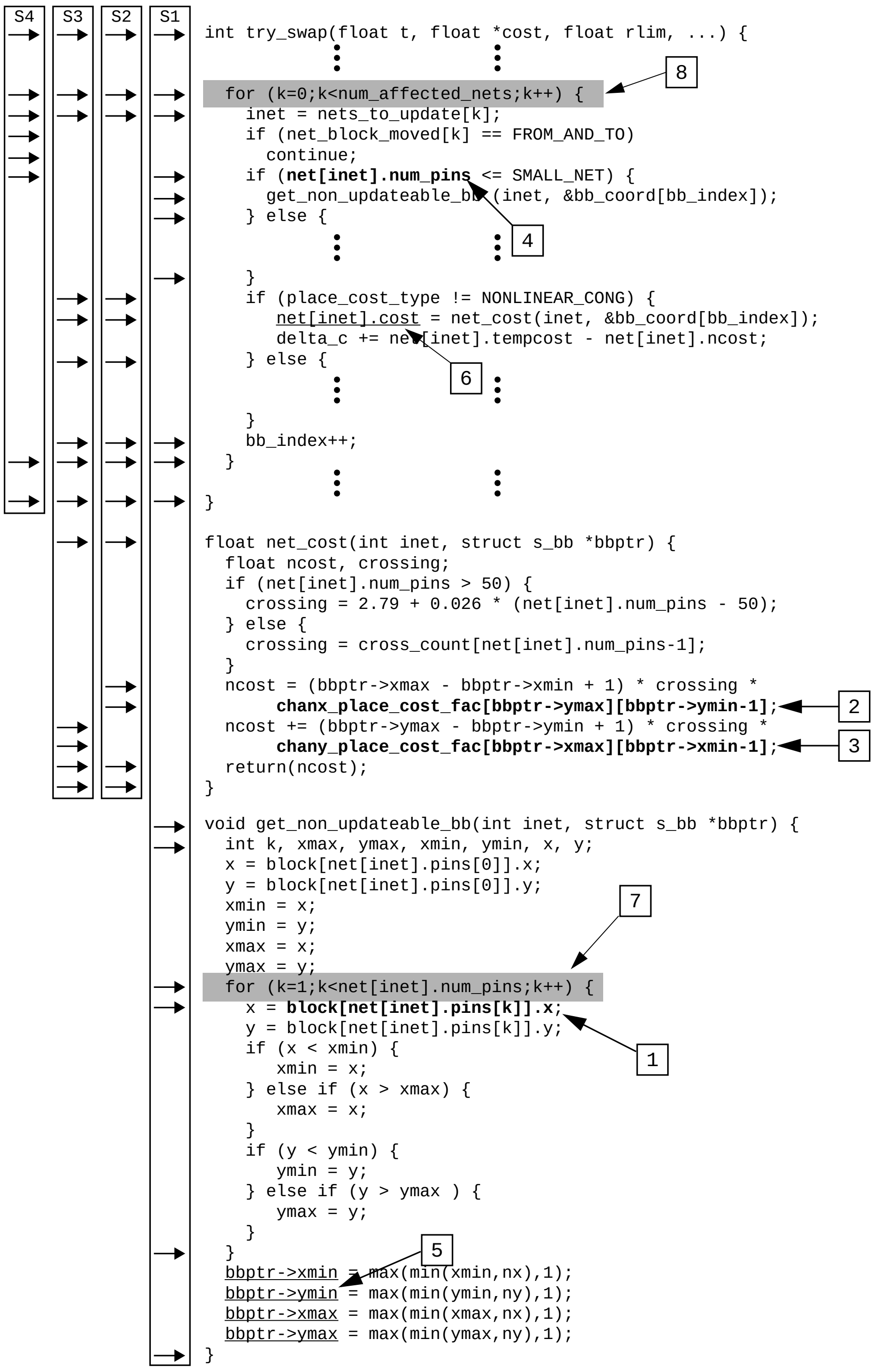
Arrays and Structures. Unravel performs an index analysis on array references and resolves different structure fields. Thus an assignment or reference to an array element or structure field does not cause the slicer to access the entire array or structure, but only the individual element, thereby trimming down the resulting slice.

Pointers. Unravel performs a pointer analysis for statically allocated objects. For every assignment and reference through a pointer to a static object, Unravel keeps track of the set of objects that can possibly be reached. This analysis takes into consideration accesses through multiple levels of indirection, treating objects at each indirection level separately. Unravel uses this information to prune away those objects that cannot be reached at each pointer access, thus disambiguating accesses to separate objects.

Procedures. Unravel constructs program slices across procedure boundaries. To enable inter-procedure slices, Unravel performs inter-procedure analysis, matching actual parameters with formal parameters and handling return values at call sites to track data dependences across procedure calls. However, Unravel ignores indirect procedure calls, and thus program slicing terminates at the boundary of any indirectly-called procedure, which sometimes occurs in our benchmarks.

Slicing for Pre-Execution. We use Unravel to compute program slices for memory references that suffer from frequent cache misses by specifying each memory reference to Unravel as a separate slice criterion. We modify Unravel to address five issues related to our *memory-driven* program slices: slice criterion specification, store removal, slice termination, slice merging, and code pinning. This section describes our modifications to Unravel using the code example in Figure 4.2 from VPR, an application from the SPEC CINT2000 benchmark suite.

Slice Criterion Specification. As discussed in Section 3.1, one can use either cache-miss profiles or static compiler analysis to identify the delinquent loads in a program. Our aggressive compiler employs offline profiling, in which delinquent loads are identified by a separate memory profiling tool as program counters (PCs) of load instructions. We translate each of these load PCs into a source file name, line number, and variable name using debugging information. In Figure 4.2, four frequent cache-missing memory references in the VPR application are shown in bold face, labeled “1” – “4.” Note, these memory references occur across three different procedures,



```
int try_swap(float t, float *cost, float rlim, ...) {
               ⋮          ⋮
  for (k=0;k<num_affected_nets;k++) {
    inet = nets_to_update[k];
    if (net_block_moved[k] == FROM_AND_TO)
      continue;
    if (net[inet].num_pins <= SMALL_NET) {
      get_non_updateable_bb(inet, &bb_coord[bb_index]);
    } else {
               ⋮          ⋮
    }
    if (place_cost_type != NONLINEAR_CONG) {
       net[inet].cost = net_cost(inet, &bb_coord[bb_index]);
       delta_c += net[inet].tempcost - net[inet].ncost;
    } else {
               ⋮          ⋮
    }
    bb_index++;
  }
               ⋮          ⋮
}

float net_cost(int inet, struct s_bb *bbptr) {
  float ncost, crossing;
  if (net[inet].num_pins > 50) {
    crossing = 2.79 + 0.026 * (net[inet].num_pins - 50);
  } else {
    crossing = cross_count[net[inet].num_pins-1];
  }
  ncost = (bbptr->xmax - bbptr->xmin + 1) * crossing *
       chanx_place_cost_fac[bbptr->ymax][bbptr->ymin-1];
  ncost += (bbptr->ymax - bbptr->ymin + 1) * crossing *
       chany_place_cost_fac[bbptr->xmax][bbptr->xmin-1];
  return(ncost);
}

void get_non_updateable_bb(int inet, struct s_bb *bbptr) {
  int k, xmax, ymax, xmin, ymin, x, y;
  x = block[net[inet].pins[0]].x;
  y = block[net[inet].pins[0]].y;
  xmin = x;
  ymin = y;
  xmax = x;
  ymax = y;
  for (k=1;k<net[inet].num_pins;k++) {
    x = block[net[inet].pins[k]].x;
    y = block[net[inet].pins[k]].y;
    if (x < xmin) {
       xmin = x;
    } else if (x > xmax) {
       xmax = x;
    }
    if (y < ymin) {
       ymin = y;
    } else if (y > ymax ) {
       ymax = y;
    }
  }
  bbptr->xmin = max(min(xmin,nx),1);
  bbptr->ymin = max(min(ymin,ny),1);
  bbptr->xmax = max(min(xmax,nx),1);
  bbptr->ymax = max(min(ymax,ny),1);
}
```

Figure 4.2: VPR code example to demonstrate our program slicing. Labels "1" – "4" indicate the cache-missing memory references selected for the slicing criteria. Labels "5" and "6" indicate the memory references that will be eliminated by store removal. Labels "7" and "8" indicate loops that bound the scope of slicing. Labels "S1," "S2," "S3," and "S4" show the slice result for the selected memory references.

`try_swap()`, `net_cost()`, and `get_non_updateable_bb()`. We perform a single slicing run for each memory reference by specifying it as a slice criterion to Unravel.

Store Removal. As discussed in Section 3.4, helper threads should never modify the memory state which is visible to the main thread to ensure correct execution of the program. Therefore, when generating helper threads, our compiler frameworks remove all stores to statically allocated global variables and to heap variables through pointers. Such store removal enables more aggressive program slicing. In addition to removing codes off the critical path of cache-missing memory references, our program slicer also removes codes associated with stores that will eventually be eliminated when helper threads are constructed. Hence before running the slicer, we delete all DEFs to global and heap variables in the PDG produced by Unravel. When we run the slicer, all codes associated with the removed DEFs will themselves be sliced away. In Figure 4.2, the underlined references labeled “5” and “6” represent stores to heap and global variables, respectively, and our slicer removes the DEFs associated with these references.

While store removal is necessary to guarantee the correctness of the main computation, it can possibly disrupt the correctness of the generated helper threads. For instance, computations at “5” in Figure 4.2 are necessary to execute the cache-missing memory references at “2” and “3.” Therefore, by removing stores at “5” in the helper threads, the delinquent loads in `net_cost()` will not be correctly pre-executed each time `net_cost()` is entered following a call to `get_non_updateable_bb()`. Fortunately, we find the memory references “2” and “3” are exceptional cases, and the dataflow through the global or heap variables within a pre-execution region rarely leads to the cache-missing memory references. Section 5.2.6 presents data that support our observation and discusses why this observation is generally true. In exceptional cases like those in VPR, the speculation support for pre-execution ensures that incorrect helper threads never compromise the integrity of the main computation.

Slice Termination. After modifying the PDG to reflect store removal, we run the slicer once for every slice criterion corresponding to a delinquent load instruction. For each slicer run, Unravel computes a program slice across the entire program. Such a slice is too large; in fact, we are interested in slicing only the code that will eventually form the pre-execution region for the delinquent load. Unfortunately, Unravel does not know the extent of pre-execution regions since they are determined in a separate compiler pass. However, as described in Section 4.1.1, a pre-execution region is defined by a loop containing one or more delinquent loads, and our pre-execution region selection algorithm chooses either the inner-most loop or the next-outer loop encompassing the delinquent loads to serve as the pre-execution region. Hence we modify Unravel to terminate slicing once two nested looping statements above the slice criterion have been encountered (if two nested looping statements cannot be found, we terminate slicing after one looping statement).

Figure 4.2 illustrates the slice termination for VPR. Memory reference "1" is contained inside the loop labeled "7." The next-outer loop, labeled "8," is where slicing terminates for this memory reference. Memory references "2," "3," and "4" are contained inside the loop labeled "8." The next-outer loop, which is not shown in Figure 4.2, is where slicing terminates for these three memory references.

As illustrated in Figure 4.2, our slice termination policy permits slices to span multiple procedures while there is no limit on procedure-call depth. From our experience, we observe the inter-procedure analysis is very important since loops are often nested across procedure boundaries, particularly in non-numeric applications like VPR. When slicing across procedures, however, multiple paths to reach the slice criterion occur if a procedure is called from multiple sites. Our slicer pursues all call paths and searches for two nested looping statements along every path, possibly identifying multiple loops where slicing terminates for a single delinquent load instruction. Smaller slices can be constructed if the slicer only considers the most frequently-executed paths within procedures as well as across procedures; however, this requires incorporating path profile information into our compiler frameworks, which we do not support in our current compiler implementation.

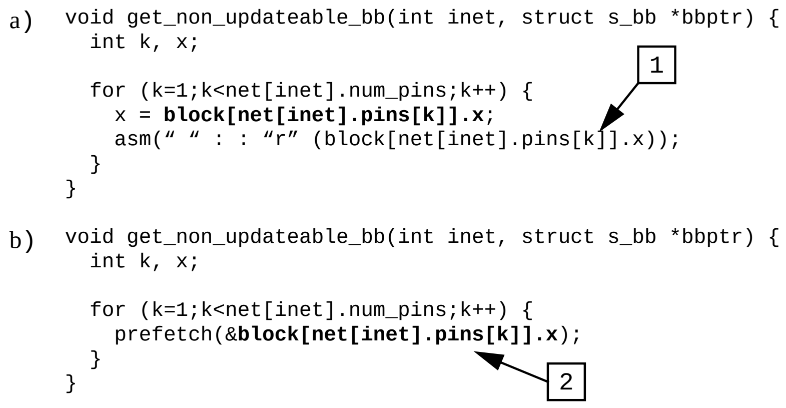
Slice Merging. After the slicing analysis completes, we have a program slice for each targeted memory reference. Figure 4.2 illustrates slices computed for the four cache-missing memory references in VPR by placing an arrow to the left of each source-code line that is contained in the slice. The slices for memory references "1" – "4" are specified by the columns of arrows labeled

"S1," "S2," "S3," and "S4," respectively. Note, slices "S2," "S3," and "S4" should continue up to the next-outer loop, which is not shown in this figure. Unravel stores each program slice as a bitmask with one bit per line of source code in the program.

Since invoking helper threads for each individual slice possibly incurs significant runtime overhead, we merge multiple slices for loads that are contained within the same pre-execution region, and invoke helper threads only once for each pre-execution region to cover all the delinquent loads within each merged slice. Slice merging occurs at the granularity of pre-execution regions. Once the pre-execution regions have been selected following the algorithm shown in Section 4.1.1, we "OR" together the bitmasks of all slices whose delinquent loads reside in the same pre-execution region. We also clear any bits that lie outside the selected pre-execution region. For example, if the loop labeled "8" in Figure 4.2 were selected as a pre-execution region, we would merge the bitmasks from slices "S1" – "S4" since memory references "1" – "4" are included within loop "8." This merged slice contains 28 out of the original 57 lines of code in Figure 4.2.

Code Pinning. Since our SUIF-based compiler frameworks generate C source code where the helper threads are attached as separate procedures, they must eventually be translated into the machine code by a C compiler. We use *gcc* (GNU C Compiler) for this purpose. Unfortunately, pre-execution code, by its very nature, is dead code since store removal eliminates all side effects, and thus it is likely to be removed during C compilation (we compile pre-execution code with the "-O2" flag, which activates dead code elimination optimization in *gcc*). For instance, after store removal and program slicing, the `get_non_updateable_bb()` function in Figure 4.2 is reduced to a single loop that *touches* the elements in the `block` array. Since this code performs no useful computation, *gcc* would remove it anyways.

To avoid pre-execution code removal during C compilation, we insert an `asm` macro that artificially consumes the loaded value from each delinquent load instruction, thus *pinning* the load and all associated pre-execution code. Figure 4.3a illustrates how our compilers perform code pinning. In Figure 4.3a, we show the pre-execution code for the `get_non_updateable_bb()` function from Figure 4.2 after program slicing is applied. An `asm` macro containing a null instruction, labeled



a)
```
void get_non_updateable_bb(int inet, struct s_bb *bbptr) {
  int k, x;

  for (k=1;k<net[inet].num_pins;k++) {
    x = block[net[inet].pins[k]].x;
    asm(" " : : "r" (block[net[inet].pins[k]].x));
  }
}
```

b)
```
void get_non_updateable_bb(int inet, struct s_bb *bbptr) {
  int k, x;

  for (k=1;k<net[inet].num_pins;k++) {
    prefetch(&block[net[inet].pins[k]].x);
  }
}
```

Figure 4.3: Code generated by our aggressive compiler for the `get_non_updateable_bb()` function from Figure 4.2. a) Pre-execution code after program slicing in which an `asm` macro is added for code pinning (labeled "1"). b) Pre-execution code after program slicing and prefetch conversion (labeled "2"). Bold-face code denotes the cache-missing memory references.

"1," has been added to force the data from the `block` array memory reference, *i.e.*, the slice criterion used by the program slicer, to be consumed. This creates a virtual data dependence between the consumer null instruction and the pinned instruction, *i.e.*, `block` array memory reference in Figure 4.3a. Since *gcc* does not remove the `asm` code, inserting `asm` code in turn prevents the removal of the pre-execution code.

Prefetch Conversion

As shown in the previous section, program slicing removes noncritical instructions unnecessary to compute the delinquent memory references, thereby expediting pre-execution. To further speedup the helper threads, we find another opportunity for optimization. In many processor designs that support out-of-order execution, a load instruction enters the reorder buffer (ROB) which commit instructions in order and guarantee precise interrupts. Moreover, the load instruction also enters the load-store queue (LSQ) for memory-address disambiguation and correct value forwarding from an earlier store instruction with the same address. Hence a normal load instruction cannot leave both the ROB and LSQ until it is resolved, and thus it blocks the processor pipeline upon incurring cache misses and prevents other independent instructions from leaving the ROB and LSQ. During program slicing, however, some of the consumer instructions for delinquent loads are identified as a part of noncritical code and thus eliminated from the pre-execution code. For those delinquent loads without any consumer instructions, we convert them into nonblocking prefetch instructions, and this is called *prefetch conversion*.

Prefetch conversion is a relatively simple optimization when combined with program slicing because the dependence information necessary to perform prefetch conversion is already available from the program slicing analysis. Our aggressive compiler performs prefetch conversion in the following way. We examine every delinquent load in the helper thread code. If the loaded value of a delinquent load is not consumed by any other instructions within the same helper thread, *i.e.*, the program slicer has removed those statements that depend on the delinquent load, the load is converted to a prefetch instruction. Applying this simple algorithm to the VPR code example in Figure 4.2, we see that memory references "1," "2," and "3" can be converted into prefetches. Figure 4.3b illustrates the final sliced helper thread code for the `get_non_updateable_bb()` function after converting the blocking memory reference (labeled "1" in Figure 4.2) into a nonblocking prefetch instruction (labeled "2" in Figure 4.3b). As to be discussed in Section 4.3.6, our compiler assumes the target architecture supports a prefetch instruction, which is inlined into the helper thread code using the `prefetch` macro shown in Figure 4.3b.[5]

Helper Thread Initiation

Helper threads can be forked and synchronized in many ways depending on characteristics of the pre-execution region and properties of the helper thread code after applying program slicing and prefetch conversion optimizations. In our SUIF-based compiler frameworks, we introduce three schemes to initiate helper threads, *i.e.*, one serial (Serial) and two parallel (DoAll, DoAcross) schemes as illustrated in Figure 4.4. For each identified pre-execution region, we assign one thread initiation scheme and transform the helper thread code accordingly to enable helper threads to

[5]Prefetch instruction is supported in many architectures, *e.g.*, `lfetch` in EPIC (Explicitly Parallel Instruction Computing) architecture for the Intel's Itanium / Itanium 2 processors. It is often implemented as nonblocking and nonfaulting, and usually has lower priority than a normal load, thereby being dropped when normal loads need to be serviced immediately.

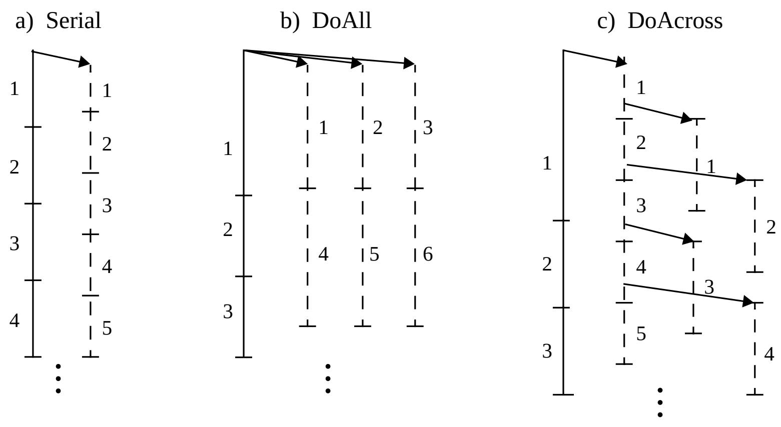


Figure 4.4: Three helper thread initiation schemes: a) Serial, b) DoAll, and c) DoAcross. Solid lines denote the main thread, dotted lines denote the helper threads, arrows denote the thread spawning, and numeric labels denote the loop iteration counts.

get ahead of the main thread. Our two parallel schemes rely on loop parallelization to run multiple helper threads simultaneously and thus speedup pre-execution. This section provides details about how we parallelize the pre-execution region and construct helper threads to implement the three different thread initiation schemes in our compiler frameworks. Later in Section 5.2.4, the experimental results show our algorithm to select the thread initiation scheme always chooses the best scheme for selected pre-execution regions.

1. **Serial**. Occasionally, prefetch conversion optimization successfully converts all delinquent loads within the pre-execution region into prefetch instructions, thereby leaving no blocking loads in the helper thread. Since program slicing has already removed noncritical instructions in the pre-execution region, even a single helper thread can easily get ahead of the main thread without being stalled on cache misses and develop an enough prefetch distance, resulting in effective prefetching. Therefore, when prefetch conversion has eliminated all blocking loads in the pre-execution region, we use Serial scheme to initiate a single helper thread to perform pre-execution. In this scheme, the main thread (solid line) spawns one helper thread (dotted line) before entering the pre-execution region as shown in Figure 4.4a. Then the spawned helper thread executes code for the entire pre-execution region sequentially. One advantage of the Serial scheme is that it occupies only one hardware context, which is

very helpful especially when spare hardware contexts in a processor are limited as in the Intel Pentium 4 processor with Hyper-Threading Technology which supports two logical processors. Moreover, since the main thread launches only one helper thread, it does not have to manage multiple helper threads and synchronize with them as in our parallel schemes described later. Finally, since we do not parallelize loops speculatively, there is less chance to generate inaccurate memory addresses. (Note, store removal can still cause generation of inaccurate addresses.) However, in many cases, prefetch conversion cannot eliminate all blocking loads within a pre-execution region. For instance, memory reference "4" in Figure 4.2 is a delinquent load that cannot be converted into a prefetch instruction because the value it loads is required by another instruction in the helper thread code. Such blocking loads will cause the sole helper thread to stall, preventing it from getting ahead of the main thread.

2. **DoAll**. When a single helper thread cannot get ahead of the main thread due to remaining blocking loads in the helper thread, we distribute the pre-execution code using loop parallelization across multiple helper threads so that we can execute the pre-execution region in parallel. This allows to overlap more memory accesses simultaneously and helps to remove significant memory stalls in the main thread compared to using a single helper thread. Although each individual helper thread does not run fast due to blocking loads, multiple helper threads *as a group* can run faster than the main thread, resulting in effective pre-execution. In conventional optimizing compilers, loop parallelization requires the compiler to exactly analyze dependences in order to guarantee that the optimization is completely safe and valid. This is nearly impossible for those loops that we handle in our study because of complicated control flow and frequent use of pointers. On the other hand, in our aggressive compiler for pre-execution, program slicing has already broken many dependences in the helper threads, and thus very few or even zero loop-carried dependences exist in our sliced codes. Moreover, our loop parallelization is speculative and aggressive in that our compiler only analyzes the loop induction variables and it does not perform any dependence analysis in the loop body.

However, thanks to the speculation support that we assume from the SMT processor (see Section 4.1.4), the correctness of the main thread is guaranteed. Hence our loop parallelization has much potential to extract abundant amount of thread-level parallelism in a program.

We propose two parallelization schemes to initiate helper threads based on the type of the loop induction variables. The first parallelization scheme is called DoAll scheme which parallelizes the affine loops, *i.e.*, the induction variable is updated arithmetically like `i++`. Upon detecting an affine loop, our compiler parallelizes the loop by assuming each loop iteration is fully independent. Figure 4.4b shows how helper threads are forked and executed under the DoAll scheme. Before entering a pre-execution region, the main thread spawns multiple helper threads and makes them execute different loop iterations in round-robin fashion (denoted by loop iteration labels). In this scheme, each helper thread keeps a private copy of the loop induction variable and updates it locally every loop iteration. In order to guarantee all loop iterations are executed exactly once without missing any of them, each helper thread starts from a different loop iteration and updates the loop induction variable by *the number of spawned helper threads* times more than the original amount; for instance, the helper thread updates induction variable by `i=i+3` when the original code is `i++` and the main thread launches 3 helper threads. Note, however, in our loop parallelization schemes, multiple helper threads share the same code as shown in the code example (see Figures 4.10b and 4.11).

3. **DoAcross**. The second parallelization scheme is called DoAcross scheme which parallelizes loops whose induction variable is updated by dereferencing a pointer. Such loops are called *pointer-chasing* loops. Pointer-chasing loops are serialized by nature because of the sequential update of the loop induction variables. However, we observe each loop-body execution can be performed independently even though there exists loop-carried dependence through the induction-variable update. As shown in Figure 4.4c, our compiler generates a single helper thread, called *backbone thread*, to execute the induction variable update code sequentially. At every loop iteration, the backbone thread gets the pointer value of the loop induction

```
Given:  Pre-Execution Region Set, P
Compute: Serial Loop Set, SE
         DoAll Loop Set, DA
         DoAcross Loop Set, DX
         Procedure Set, F
------------------------------------------
 1: SE = DA = DX = F = Φ;
 2: for each loop L in P {
 3:   if (num_blocking_load(L) == 0)
 4:      SE = SE ∪ {L};
 5:   else {
 6:      if (induction(L) == AFFINE)
 7:         DA = DA ∪ {L};
 8:      else
 9:         DX = DX ∪ {L};
10:   }
11:   F = F ∪ called_procedures(L);
12: }
```

Figure 4.5: Algorithm to decide the thread initiation scheme for pre-execution regions.

variable for the current iteration, and then spawns another thread, called *rib thread*, to execute the loop body. Meanwhile, the backbone thread dereferences the current pointer of the induction variable so that the loop-body execution and pointer dereference can be overlapped. Moreover, multiple loop-body executions are often performed in parallel as well, issuing multiple prefetches simultaneously. The DoAcross scheme requires two types of inter-thread communication: one between the main thread and the backbone thread, and another between the backbone thread and the rib threads. First, the backbone thread keeps checking the relative distance, called *run-ahead distance*, between the main thread and itself. Once it reaches the pre-defined prefetch distance, *i.e.*, 25 loop iterations in our study, the backbone thread either sleeps on a hardware semaphore or enters a busy-waiting loop until the main thread catches up, reducing the run-ahead distance below 25 iterations. Synchronization between the backbone thread and a rib thread occurs as a consequence of passing the correct induction variable value and live-ins.

Figure 4.5 presents our compiler algorithm to select thread initiation schemes for pre-execution regions. For each loop in the pre-execution region set, P, we first check whether or not all the blocking loads have been removed by prefetch conversion and store removal (line 3) and choose the Serial scheme if there remains no blocking load in the region (line 4). Otherwise, we select one of the two parallel schemes. When the loop induction variable is affine and updated arithmetically, we pick the DoAll scheme (lines 6 and 7). For any other patterns where induction variable update is serialized as in a pointer-chasing loop, we choose the DoAcross scheme (lines 8 and 9) so that a backbone thread can be dedicated to update the induction variable and distribute live-ins to rib threads.

Before moving on, we find an issue regarding overhead to be worth noting. In our pre-execution model, helper threads are spawned at the loop entrance point, and thus both the main thread and helper threads enter the targeted pre-execution region at the same time. Consequently, helper threads take some time to reach the prefetch distance and they provide little benefit to the main thread for the first few loop iterations. While such a *warm-up* phase in pre-execution is not desirable, it can be alleviated by choosing loops with a large loop-trip count or the next-outer loops, which is the rationale behind our pre-execution region selection algorithm. Although it is also possible to make helper threads skip the first few iterations and start from the middle of the loop, we observe this does not affect the performance noticeably and deciding how many loop iterations to skip adds more complexity to our algorithm. Moreover, for some pre-execution regions like pointer-chasing loops, live-in variables for the future loop iterations are often unavailable at the loop entrance point due to serialization, and thus all loop iterations must be executed to acquire necessary values.

Synchronization

After applying the above three optimization techniques to expedite pre-execution, helper threads may acquire too much speed advantage over the main thread, thereby running too far ahead. Such run-away helper threads issue prefetches too early and evict useful cache blocks that will be accessed by the main thread in the near future. To avoid such undesirable effects, we synchronize the helper threads with the main thread so that helper threads cannot run ahead of the main thread by more than a certain distance. In this study, we set the prefetch distance to 25 loop iterations, which seems reasonable for many pre-execution regions in our benchmarks.[6] Since the purpose of synchronization is to keep the run-ahead distance below the prefetch distance, we do not add synchronization code for those loops with loop-trip count of less than 25 iterations to reduce the runtime overhead associated with synchronization.

In our SUIF-based compiler frameworks, we evaluate two thread synchronization mechanisms: hardware semaphore and software semaphore. First, when using hardware semaphore as the synchronization mechanism, the main thread resets the value of the hardware semaphore counter, which is a special hardware register, to the prefetch distance before entering a pre-execution region. At the end of every loop iteration, the main thread increments the hardware semaphore register by 1. On the other hand, a helper thread decrements it by 1 after executing one loop iteration. Hence when the helper threads run faster than the main thread, the semaphore counter value decreases toward zero. Once the counter value reaches zero, it implies that the helper threads have executed 25 more loop iterations than the main thread since the beginning of the targeted loop. From then on, whenever a helper thread decrements the counter resulting in a negative value, it registers itself in the FIFO (First-In First-Out) queue and the processor suspends the corresponding helper thread immediately. Suspended helper threads remain dormant until the main thread increments the hardware semaphore again and wakes up one helper thread at the head of the FIFO queue.

While hardware semaphores provide low overhead, they also have a drawback; they require new instructions to support thread synchronization and modification to the processor architecture to implement hardware semaphore registers. Therefore, we also investigate a software-only synchronization mechanism. In our software mechanism, both the main thread and the helper threads increment their own counters that are declared as global variables. For every loop iteration, each helper thread computes its run-ahead distance by taking the difference between its counter and the main thread's counter. Once a helper thread reaches the prefetch distance, it enters a busy-waiting loop and does nothing but check the run-ahead distance. After the main thread finishes one more loop iteration and increments its global counter, the polling helper thread detects it, exits the

[6]We also varied the prefetch distance and observe that the performance of most programs is not sensitive to the prefetch distance value.

busy-waiting loop, and starts execution of the next loop iteration from where it was suspended. Although our software mechanism has the advantage that it does not require additional hardware support, the busy-waiting loop injects many useless instructions into the processor's pipeline, and thus wastes valuable resources such as the fetch units and reorder-buffer entries that could have been used by the main thread. Section 5.2.5 compares the performance of pre-execution with these two different synchronization mechanisms and discusses tradeoffs.

4.1.3 Minimizing Ineffectiveness of Pre-Execution

To reduce the runtime overhead associated with pre-execution, we introduce two new methods: terminating all active helper threads after the main thread exits a pre-execution region and recycling helper threads. First, for several reasons, helper threads can still remain active even after the main thread leaves a pre-execution region. For instance, due to the speculative loop parallelization in Section 4.1.2 and store removal in Section 4.1.4, the constructed helper threads can be incorrect and not finish properly following the wrong control paths. On the other hand, helper threads occasionally run behind the main thread simply because they do not acquire enough processor resources. Since any helper thread activity after the execution of the corresponding pre-execution region in the main thread is useless, the main thread should ensure the termination of such wasteful helper threads. We assume ISA support from our processor and insert a `kill` instruction immediately after a pre-execution region in the main thread code. However, instead of introducing a new instruction, we can also implement this kind of thread-killing mechanism alternatively by checking the liveness of each forked helper thread and executing the `suspend` instruction, which we assume from the processor to manage threads, when it is active.

Second, we employ a software mechanism to recycle helper threads to reduce the thread spawning cost. Thread initiation in SMT processors can be expensive due to context initialization (our thread initiation code contains 25 instructions). To minimize the thread initiation overhead, we *recycle* threads. We first create a helper thread for each idle hardware context only once during program startup. After creation, each helper thread enters a dispatch loop and suspends itself immediately. To perform a “fork,” the forking context communicates a PC value through memory and executes a `resume` instruction to unblock one of the suspended threads. The “forked” thread then jumps indirect through the PC argument. If the forked thread completes normally, it returns to the dispatch loop and suspends itself until the next fork, thus recycling the thread. However, if the forked thread is halted by the main thread via a `kill` instruction, it cannot simply be resumed. Instead, to prevent the thread from resuming its path of execution prior to the kill, we assume the `kill` instruction sets the killed thread’s PC to point to the instruction immediately following the `suspend` instruction in the dispatch loop. Consequently, a fork performed on a killed thread resumes the thread as if it had returned to the dispatch loop normally.

4.1.4 Preserving Program Correctness

Prefetching helper threads are meant to affect the performance of a program only; they should never disrupt the correctness of the main computation. This section discusses two issues, store and system call removal and exception handling, to preserve the correctness of the main computation while performing pre-execution.

Store and System Call Removal

To prevent pre-execution from disrupting the main computation, results computed by the helper threads should never be integrated into the shared machine state. Disrupting the program correctness can only occur through memory and the hard disk space. As mentioned in Section 4.1.2, we remove all store instructions and system calls in helper threads to guarantee that the helper threads never write to the heap memory region or hard disk space. However, we still allow stores to stack memory since the stack is allocated for each individual helper thread. This ensures our helper threads properly handle procedure calls if the original program does so. As discussed in Section 4.1.2, store and system call removal helps to eliminate even more instructions by program slicing. Since store instructions and system calls within the pre-execution region are removed eventually in the final helper thread code anyway, our program slicer ignores them during the analysis phase so that code affecting stores and system calls is not included in the slice. This in turn improves the correctness of our speculative loop parallelization as well since loop-carried dependences are themselves more likely to be removed. In addition, delinquent loads often belong to the code that is associated with the removed stores. One scenario which is easily found in normal programs is that a program loads a value, does some computation, and stores the computed result back to memory. Since the store will be removed from the helper threads, the delinquent load brings a value to perform useless computation. In that case, we remove the destination operand of such delinquent loads and convert them to prefetch instructions.

Exception Handling

Due to aggressive loop parallelization and store removal in our compiler frameworks, our constructed helper threads are speculative, which means they may issue inaccurate or invalid memory addresses.[7] For instance, since our compiler does not perform dependence analysis in the loop body, a loop can be parallelized anyway even though it indeed contains loop-carried dependences. On the other hand, load address computations occasionally involve store to memory, which our helper threads cannot correctly perform since such stores have been already removed in the helper thread code. When illegal memory addresses are generated by helper threads, an exception occurs. To avoid the main thread termination as a consequence of exceptions signaled by helper threads, we should handle such exceptions. Instead of involving the OS, we assume a certain hardware mechanism to detect an exception raised by helper threads and terminate the faulting helper thread. Once terminated, the helper thread remains suspended until the main thread resumes it again before entering the next pre-execution region. Therefore, it is possible that the main thread's benefit from pre-execution reduces when an exception occurs. However, we observe that our helper threads issue valid addresses most of the time and rarely incur exceptions.

[7]Program slicing and prefetch conversion optimizations never affect the correctness of helper threads. Our program slicing extracts all the necessary code to compute delinquent loads, including control statements as well as address computation code. In addition, the analysis for prefetch conversion guarantees there exist no instructions in the slice that are dependent of the load instruction to be converted into prefetch.

4.2 Compiler Algorithms for SUIF-Based Reduced Frameworks

Our aggressive compiler generates effective helper threads by using offline profiles for the cache-miss behavior and loop-trip counts as well as Unravel for program slicing and prefetch conversion. Although such offline profiles are widely used and Unravel is publicly available, it would be desirable from the compiler's implementation standpoint if our pre-execution compiler frameworks did not rely on these additional compilation steps and still provided comparable performance with the aggressive compiler. In this section, we introduce our SUIF-based reduced compiler frameworks (*reduced compilers* for short) where offline profiles and program slicer are removed one at a time and replaced with static compiler algorithms to simplify the design of the compiler frameworks for pre-execution.

We present four reduced compilers. Our first reduced compiler eliminates program slicing that is previously done by Unravel in the aggressive compiler. Instead, it relies on *gcc*'s dead code elimination optimization to remove noncritical code in helper threads. The second reduced compiler introduces a static compiler algorithm to enable prefetch conversion in the absence of the program slicing analysis. The third reduced compiler eliminates offline memory profiles along with the program slicer and uses simple heuristics to identify delinquent loads statically. Finally, the fourth reduced compiler additionally removes loop-trip count profiles and selects pre-execution regions without using profile information.

4.2.1 Eliminating Program Slicing

Our program slicing analysis, described in Section 4.1.2, essentially consists of 3 steps: store removal, slicing, and code pinning. Among these 3 steps, the slicing step can be redundant under certain circumstances. Because store and system call removal eliminates all side effects inside a pre-execution region, the code removed by program slicing is dead code anyways and can potentially be removed during C compilation by dead code elimination optimization. Furthermore, while program slicing exactly identifies critical computations leading up to the delinquent-load instructions to ensure that the relevant code still remains in the helper threads, code pinning achieves a similar

```
void get_non_updateable_bb(int inet,
                           struct s_bb *bbptr) {
  int k, xmax, ymax, xmin, ymin, x, y;

  x = block[net[inet].pins[0]].x;
  y = block[net[inet].pins[0]].y;
  xmin = x;  ymin = y;  xmax = x;  ymax = y;

  for (k=1;k<net[inet].num_pins;k++) {
    x = block[net[inet].pins[k]].x;
    asm(" " : : "r" (block[net[inet].pins[k]].x));
    y = block[net[inet].pins[k]].y;
    if (x < xmin) {
       xmin = x;
    } else if (x > xmax) {                   1
       xmax = x;
    }
    if (y < ymin) {
       ymin = y;
    } else if (y > ymax ) {
       ymax = y;
    }
  }
}
```

Figure 4.6: Code generated by our compiler without program slicing for the `get_non_updateable_bb()` function from Figure 4.2. An `asm` macro is added for code pinning (labeled "1").

effect by introducing artificial side effects for the delinquent loads and their associated code, thus pinning them down and preventing their removal at compile time. Considering this redundancy in slicing, we propose a simpler approach in which store removal and code pinning are performed alone, getting rid of the program slicing step. This approach allows us to eliminate Unravel and its integration with other compiler modules, thereby simplifying the implementation of the compiler framework.

As an example, Figure 4.6 shows the helper thread code generated by our reduced compiler for the `get_non_updateable_bb()` function from Figure 4.2. The code in Figure 4.6 is identical to the original code from Figure 4.2 except for the removal of side effects (underlined code labeled "5" in Figure 4.2) and the insertion of an `asm` macro (labeled "1" in Figure 4.6). In particular, the conditional tests in the loop body, which are unnecessary for computing the delinquent load, still remain in the helper thread since program slicing has been omitted. However, these extraneous statements do not have side effects because of store removal. Hence even though the code in Figure 4.6 appears to be less efficient than its sliced counterpart from Figure 4.3a, we have observed the unnecessary code is eventually removed by *gcc* during backend compilation. As long as dead code elimination removes the unnecessary code, the binary generated from Figure 4.6 is as efficient as the binary generated from Figure 4.3a. Section 5.3.1 reports the performance impact of eliminating program slicing and also compares the ability to remove noncritical code between Unravel and backend code optimizer.

4.2.2 Prefetch Conversion without Program Slicing

In our aggressive compiler, prefetch conversion is coupled with program slicing as described in Section 4.1.2. Hence eliminating Unravel requires new algorithms to enable prefetch conversion. Without program slicing analysis, prefetch conversion becomes harder since no code, except for stores and their associated code, gets removed from the helper thread. Therefore, converting delinquent loads into prefetches is limited whenever noncritical code, which would have been removed by program slicing, consumes data loaded by the delinquent loads. To ensure all candidate loads for prefetch conversion are accurately identified in the absence of program slicing, it is necessary to disregard the dependences that delinquent load instructions may have with the noncritical code in anticipation of its removal during backend compilation.

We propose a static compiler algorithm for prefetch conversion, which does not require program slicing analysis. The algorithm first computes the set of active variables that are used to compute delinquent loads contained inside a pre-execution region. Then it visits each delinquent load and converts loads whose destination operand is not included in the active-variable set into prefetches. Since noncritical code that will be removed during C compilation does not produce any active variables, their presence in the helper thread code does not limit opportunities for prefetch conversion. Eqs. 4.3 and 4.4 show how to compute the active-variable set $V_{<m,v>}$ for a memory variable v corresponding to a delinquent load instruction at statement m:

$$V_{<m,v>} = \begin{cases} V_{<n,v>} & \text{if } v \notin defs(n) \\ Vdef_{<n,v>} & \text{otherwise} \end{cases} \tag{4.3}$$

$$Vdef_{<n,v>} = \left(\bigcup_{x \in refs(n)} \{x\} \right) \bigcup \left(\bigcup_{x \in refs(n)} V_{<n,x>} \right)$$

$$\bigcup\left(\bigcup_{y\in refs(k)}\ \bigcup_{k\in control(n)} V_{<k,y>}\right) \tag{4.4}$$

These equations are almost identical to Eqs. 4.1 and 4.2 from Section 4.1.2 that are used for the program slicing analysis. In particular, the first two terms in Eq. 4.4 account for dataflow within the critical code for computing variable v similar to the first two terms in Eq. 4.2, while the last term in Eq. 4.4 accounts for control flow similar to the last term in Eq. 4.2. This similarity is not accidental. Rather, it reflects the fact that prefetch conversion is fundamentally enabled by the program slicing analysis. Even if we do not use external program slicers like Unravel and rely on backend compilers to remove unnecessary code, we should still perform a slicing-like analysis to identify candidate loads for prefetch conversion. Note, however, our standalone prefetch converter only performs a basic data and control flow analysis, and thus it is far less complex than Unravel, which performs several sophisticated analyses (see Section 4.1.2).

Eqs. 4.3 and 4.4 compute the active-variable set for a single memory reference v. We compute $V_{<m,v>}$ for all problematic memory references in the pre-execution region and take their union to form the set of all active variables. Using this merged active-variable set, we identify candidate loads to be converted into prefetches as mentioned above. If the data from a delinquent load does not assign a variable belonging to the merged active-variable set, it means the load instruction does not contribute to the critical data or control flow within the pre-execution region; hence we convert the load into a prefetch instruction. Otherwise, the load remains untouched and an `asm` macro is inserted for code pinning. In Section 5.3.2, we examine the effectiveness of our static compiler algorithm for prefetch conversion.

4.2.3 Eliminating Cache-Miss Profiles

Our aggressive compiler relies on offline memory profiles to identify delinquent loads in a program. We collect summary profiles that provide the total number of L1 and L2 misses for each static load instruction over the entire simulation region. While our memory profiles provide a very accurate overall picture of the memory behavior for a particular run of a workload, they also have several drawbacks. First, offline profiling is often cumbersome to perform; it requires additional profiling tools such as a cache simulator or the VTune, and one or more separate profiling runs. Moreover, the profile results may not be adaptable; even for the same workload, memory profile results may vary for different input sets or target platforms. In addition, summary profiles provide a single value of information for the entire program execution, and thus it is hard to get the chronology of dynamic cache behavior of a program.

For our third reduced compiler, we propose a simple compiler algorithm to statically identify delinquent loads so as to get rid of the additional compilation steps for memory profiling.[8] Unfortunately, accurately predicting memory behavior at compile time using static compiler analyses is nearly impossible. Our algorithm relies on simple heuristics to identify those static loads that are likely to miss in the cache frequently. In particular, we observe the following heuristics to be very useful for identifying delinquent loads in a loop structure: a load instruction whose effective address is computed as a function of the loop induction variables has a high probability to exhibit poor memory performance. This is not surprising. Since loop induction variables, by definition, are updated every loop iteration, those load instructions dependent upon the loop induction variables usually produce different effective addresses for each access and they are likely to cause cache misses. Hence our reduced compiler performs loop-level analysis to identify all loads that satisfy the above heuristics, which is a fairly straight-forward task, and we pick all such loads as delinquent loads to be pre-executed. We observe this simple compiler algorithm successfully identifies 88% of the total L1 misses that occur in 10 of our memory-bound benchmarks.[9]

While our static algorithm successfully identifies most of the delinquent loads in our benchmarks, it also has a tendency to identify too many loads as delinquent. This causes pre-execution to be performed even when it is not necessary, potentially increasing the runtime overhead. To minimize performance degradation due to unhelpful pre-execution, we modify the pre-execution

[8]Our static analysis, however, does not resolve all three drawbacks of offline summary profiles that are mentioned above. It is not aware of the input set for the workload when analyzing program code, producing the same output regardless of the input sets or target platforms.

[9]This result is obtained from manual inspection of benchmarks. Unfortunately, for 3 benchmarks, the majority of cache misses occur either in libraries, for which we do not have source code, or in too many loops, making it infeasible to analyze by hand. However, we expect our heuristics is effective for these 3 benchmarks as well.

region selection algorithm presented in Figure 4.1 to omit the second pass through the inner-most loops (lines 11 – 13). This pass picks the inner-most loops with fewer than 25 iterations as the pre-execution regions in order to guarantee all identified delinquent loads are covered by helper threads. Without memory profiles, however, many short inner-most loops may have falsely identified delinquent loads, and thus we refrain from choosing any short inner-most loops in our reduced compiler. The performance impact of eliminating cache-miss profiles in our compiler framework will be discussed in Section 5.3.3.

4.2.4 Eliminating Loop Profiles

In addition to memory profiles, our aggressive compiler requires loop-trip count profiles to estimate the amount of loop work and select optimal pre-execution regions following the algorithms shown in Section 4.1.1. Similar to memory profiles, collecting loop-trip count profiles can also be cumbersome. To perform such profiling, we construct the basic-block diagram of a program and run the workload on a modified processor simulator to count the number of accesses to each basic block and edge. For each loop identified from the basic-block diagram, we compute loop-trip counts by dividing the backward-edge count by the number of visits to the landing pad.[10] To avoid such additional work, we identify pre-execution regions without using loop-trip count profiles in our fourth reduced compiler.

Figure 4.7 presents a reduced algorithm for computing the set of pre-execution regions, P, in the absence of the loop-trip count profiles. This algorithm is identical to the original algorithm from Figure 4.1 except it naively assumes all inner-most loops iterate fewer than 25 iterations, which is the threshold value to determine whether or not a loop contains a small amount of work.[11] Therefore, on the first pass of the algorithm, none of the inner-most loops are selected as pre-execution regions (lines 3 and 4); instead, the algorithm picks the next-outer loops as the

[10]Landing pad is usually the first basic block of loops, which is located right before the basic block pointed to by the backward edge. Executed only once for each loop entrance, it initializes the loop-induction variable and performs the loop-termination condition check with the initial value.

[11]We observe this simple assumption is correct for 83% of the inner-most loops in our benchmarks.

```
Given: Global loop nest graph, $G_L$
Compute:  Pre-Execution Region Set, P

 1: P = $\Phi$;
 2: for each loop L in $G_L$
           from inner-most to outer-most {
 3:    if (level(L) == INNER_MOST)
 4:       continue;
 5:    else
 6:      if (P $\cap$ nested_loops(L) == $\Phi$)
 7:          P = P $\cup$ {L};
 8: }

 9: for each inner-most loop L in $G_L$ {
10:    if (P $\cap$ outer_loops(L) == $\Phi$)
11:       P = P $\cup$ {L};
12: }
```

Figure 4.7: Algorithm to compute the set of pre-execution regions, P, without loop-trip count profiles. Φ denotes the empty set.

pre-execution regions as long as nesting of pre-execution regions does not occur (lines 5 – 7). Also, notice this algorithm performs the second pass through the inner-most loops to pick any remaining loops that are not pre-executed (lines 9 – 11) as is done in Figure 4.1, a step that was eliminated in the algorithm from Section 4.2.3 to reduce potential pre-execution overhead. Although the second pass can add loops containing falsely identified delinquent loads, we find it is worthwhile to ensure none of the important inner-most loops are omitted. On the other hand, since the algorithm in Figure 4.7 assumes all inner-most loops iterate a small number of times, several inner-most loops that indeed iterate a large number of times may not be selected at the inner-most loop level, but included as a part of their next-outer loops. In this case, helper threads can run away from the main thread due to lack of synchronization in the inner-most loop body. Section 5.3.4 reports the performance results with and without loop-trip count profiles and discusses some potential drawbacks of the static algorithm for pre-execution region selection.

4.3 Implementation

Having presented various compiler algorithms for pre-execution, we now demonstrate how such algorithms are actually implemented in the SUIF compiler infrastructure to build the compiler frameworks for pre-execution. For our aggressive compiler, we also show how Unravel and SimpleScalar-based simulators can be integrated into the compiler framework to provide program slicing infor-

Table 4.1: Compilers used to generate the helper thread code for our experiments on the SMT processor simulator. The compilers differ in how they remove the unnecessary code, select the prefetch conversion candidates, identify the delinquent load instructions, and estimate the inner-most loop work.

	Code Removal	Prefetch Conversion	Problematic Load Identification	Loop Work
A	Program Slicing	Slicing-Based	Profile	Profile
B	Dead Code Elimination	Slicing-Based	Profile	Profile
C	Dead Code Elimination	Standalone	Profile	Profile
D	Dead Code Elimination	Standalone	Static Analysis	Profile
E	Dead Code Elimination	Standalone	Static Analysis	Static Analysis

mation, cache-miss profiles, and loop-trip count information. Then we discuss issues on live-in passing, and present the actual code examples generated by our aggressive compiler and compare differences in the output code for three thread initiation schemes. Finally, we discuss ISA support that our SUIF-based compiler frameworks assume from the SMT processor to enable pre-execution.

4.3.1 Summary of Compiler Frameworks

We propose five different compiler frameworks to construct helper threads. As presented in Section 4.2, we modify the implementation of our aggressive compiler by substituting alternatives to program slicing and offline profiling, thus forming four reduced compilers. Table 4.1 summarizes our five compiler frameworks along with the algorithms each compiler uses to distinguish them. Compiler A is the aggressive compiler from Section 4.1, and the remaining entries in Table 4.1, lettered "B" through "E," represent the reduced compilers from Section 4.2. First, Compiler B eliminates program slicing pass and relies on *gcc*'s dead code elimination optimization to remove noncritical code in the pre-execution region although this compiler still uses Unravel to identify candidate loads for prefetch conversion. Compiler C implements the static compiler algorithm for prefetch conversion described in Section 4.2.2 to extract prefetch conversion information without program slicing analysis, thus completely eliminating the need for Unravel. Compiler D substitutes

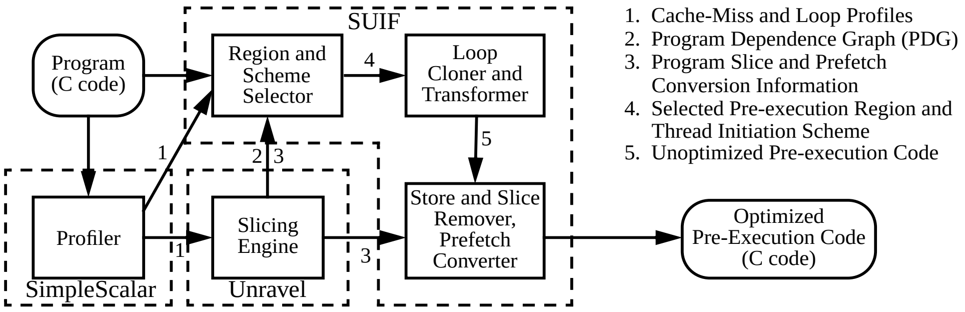

SUIF
Program (C code)
Region and Scheme Selector
4
Loop Cloner and Transformer
5
1
2 3
Profiler
1
Slicing Engine
3
Store and Slice Remover, Prefetch Converter
Optimized Pre-Execution Code (C code)
SimpleScalar
Unravel
1. Cache-Miss and Loop Profiles
2. Program Dependence Graph (PDG)
3. Program Slice and Prefetch Conversion Information
4. Selected Pre-execution Region and Thread Initiation Scheme
5. Unoptimized Pre-execution Code

Figure 4.8: Design of our most aggressive prototype compiler. This compiler uses Unravel to perform program slicing and offline profiles to drive optimizations. Arrows denote the interactions between the compiler modules.

summary cache-miss profiles with static analysis to identify delinquent loads. Finally, Compiler E relies on compile-time heuristics described in Section 4.2.4 to select pre-execution regions, thus removing loop-trip count profiles and eliminating the need for any offline profiling completely.

4.3.2 Aggressive Prototype Compiler

Figure 4.8 illustrates the design of our aggressive prototype compiler. While all necessary algorithms for pre-execution are implemented as a separate optimization pass in the SUIF, our aggressive compiler also employs external tools such as the SimpleScalar-based profilers and Unravel to acquire critical information to identify target delinquent loads and pre-execution regions, and program slicing information, respectively. We modify the SimpleScalar's cache simulator, *sim-cache* [10], to collect summary cache-miss profiles.[12] We also modify the SimpleScalar's functional simulator, *sim-fast*, to perform path profiling and derive loop-trip count information for every loop in a program. For both cache-miss and loop-trip count profiles, we use the same input set used for the actual performance run, but we collect cache-miss profiles only for the simulation region as shown in Table 5.2 to most accurately reflect program's memory behavior for the region we simulate. On the other hand, to eliminate noncritical code within pre-execution regions and

[12]Although we use a cache simulator to acquire summary cache-miss profiles, one can also use other profiling tools such as DCPI [5], Shade [18], or the Intel VTune performance analyzer [79]. However, we do not explore these other approaches since profiling efficiency is not a concern in our work.

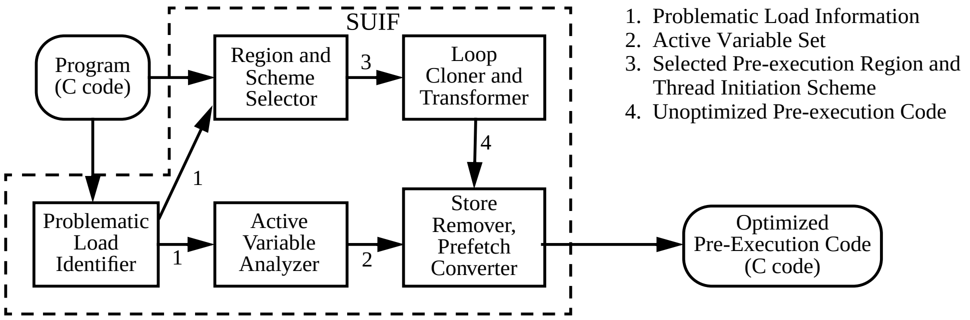

SUIF
Program (C code)
Region and Scheme Selector
Loop Cloner and Transformer
Problematic Load Identifier
Active Variable Analyzer
Store Remover, Prefetch Converter
Optimized Pre-Execution Code (C code)
1
2
3
4
1. Problematic Load Information
2. Active Variable Set
3. Selected Pre-execution Region and Thread Initiation Scheme
4. Unoptimized Pre-execution Code

Figure 4.9: Major components in our least aggressive prototype compiler. This compiler uses dead code elimination to remove the unnecessary code and static analysis to identify the delinquent loads and select the pre-execution regions. Arrows denote the interactions between the compiler modules.

drive prefetch conversion, we modify Unravel to implement the slicing algorithms presented in Section 4.1.2 and algorithms to identify prefetch conversion candidates shown in Section 4.1.2. Then our modified Unravel generates necessary information for program slicing and prefetch conversion for each program source line. Those offline profiles and information from Unravel are fed into the SUIF. Then our SUIF compiler pass identifies appropriate pre-execution regions that encompass the targeted delinquent loads following the algorithm shown in Section 4.1.1 and also chooses an optimal thread initiation scheme for each pre-execution region based on the algorithm in Section 4.1.2. When selecting pre-execution regions, we discard all regions contributing less than 3% of the application's total L1 misses. Filtering out potentially unhelpful pre-execution regions helps minimize the runtime overhead incurred by pre-execution. The SUIF compiler pass also clones each identified pre-execution region, creates separate procedures for helper threads, and transforms the code region properly. Code transformation includes parallelizing loops, removing noncritical code, pinning delinquent loads, converting blocking loads into prefetches, removing stores and system calls in pre-execution regions, and inserting codes for thread spawning and synchronization.

4.3.3 Reduced Prototype Compilers

Figure 4.9 illustrates the design of our most reduced prototype compiler, *i.e.*, Compiler E from Table 4.1. The block diagram is very similar to that of the aggressive compiler except that all external tools are replaced with static compiler algorithms described in Section 4.2. As is done for the aggressive compiler algorithms, we also implement such reduced algorithms in the SUIF compiler so that the whole helper thread construction process can be performed solely by the SUIF. In our most reduced compiler, when program source files are given, the SUIF compiler pass searches for all loops in the program and identifies delinquent loads statically following the algorithm in Section 4.2.3. Then proper pre-execution regions are selected assuming all inner-most loops iterate fewer than 25 iterations as described in Section 4.2.4. From then on, the helper thread construction is almost identical to that of the aggressive compiler. Two differences are prefetch conversion is done in the SUIF based on the algorithm presented in Section 4.2.2, and removing noncritical codes relies on *gcc*'s dead code elimination optimization (see Section 4.2.1). While Compiler E is implemented fully in the SUIF, other reduced compilers (Compiler B – D) described in Section 4.3.1 are implemented easily by substituting the reduced algorithms incrementally.

4.3.4 Live-In Passing

To generate accurate prefetches for delinquent loads, helper threads must start execution with correct live-in values. To identify live-in variables for a pre-execution region, we first call a SUIF library routine, `find_exposed_refs()`, to find all the variables used within the selected pre-execution region. Then for each identified variable, we check whether it is referenced prior to being overwritten in the pre-execution region. This is easily done by traversing all instructions within the target code region in the program execution order and examining whether the variable first appears in a source operand or a destination operand of an instruction. A variable that appears in a source operand first is a live-in variable for the pre-execution region. After all live-in variables are identified, our compiler module inserts code to pass those variables from the main thread to the helper threads. Since each hardware context has its own register file in our SMT processor model, communication through registers between threads is not feasible. Therefore, we pass live-in variables through memory using global variables that are declared solely for that purpose. To ensure the helper threads get the correct live-in values, all live-in variables are passed before the main thread forks the helper threads at the loop entrance.

4.3.5 Code Generation

Our pre-execution compiler frameworks generate the final output in the form of the SUIF intermediate format. While the SUIF can directly generate code for a number of platforms, we unfortunately do not have a SUIF-to-binary compiler for the SimpleScalar ISA.[13] Therefore, we run *s2c*, a utility provided by the SUIF, to convert the SUIF intermediate format into C source code, and then we use *gcc* as the backend compiler to generate the final executable. In Figures 4.10 and 4.11, we provide actual code examples of the SUIF output in C for our three thread initiation schemes and discuss details about the code generation of our compiler frameworks.

Figure 4.10a shows how to construct a single helper thread by applying the Serial scheme to an inner-most loop in the AMMP benchmark. In addition, in Figure 4.10b, we apply the DoAll scheme to a next-outer loop in the VPR benchmark. Note that a few lines in the VPR code example have been erased to save space. Code generations for the Serial and DoAll schemes are very similar, following five steps as indicated by the numeric labels in Figure 4.10. First, we clone the pre-execution region containing both the loop header and loop body and place the code in a single procedure (labeled "1"), which is created as a separate user procedure solely for pre-execution.[14] Then we perform our optimization algorithms for store removal, program slicing, and prefetch conversion on the loop body of the cloned code. We also generate code to spawn helper thread(s) and pass the live-in variables to the helper threads (labeled "2"). For the DoAll scheme, we find the code to update the loop-induction variable, which is already available in the internal SUIF representation, and adjust it to distribute iterations to threads in round-robin fashion (labeled "3"). Next, we insert a counting semaphore (labeled "4"), which is called "T," and initialize the value to "PD" (prefetch distance). Semaphore "T" blocks helper threads that run ahead of the main thread by more than the prefetch distance, preventing them from getting too far away.

[13]The SimpleScalar ISA is called PISA (Parallel Instruction Set Architecture), and is a derivative of the MIPS ISA.

[14]For the DoAll scheme, multiple helper threads share the same code, but just execute different loop iterations by starting with different initial conditions. This is same for the DoAcross scheme; multiple rib threads share the same rib thread code, but start execution with different loop induction variable and live-ins.

Finally, we add a `kill` directive to halt any active helper threads after the main thread leaves the pre-execution region (labeled "5"). As shown in the code example, our SERIAL scheme can be viewed as a subset of the DOALL scheme where the number of launched helper threads is one.

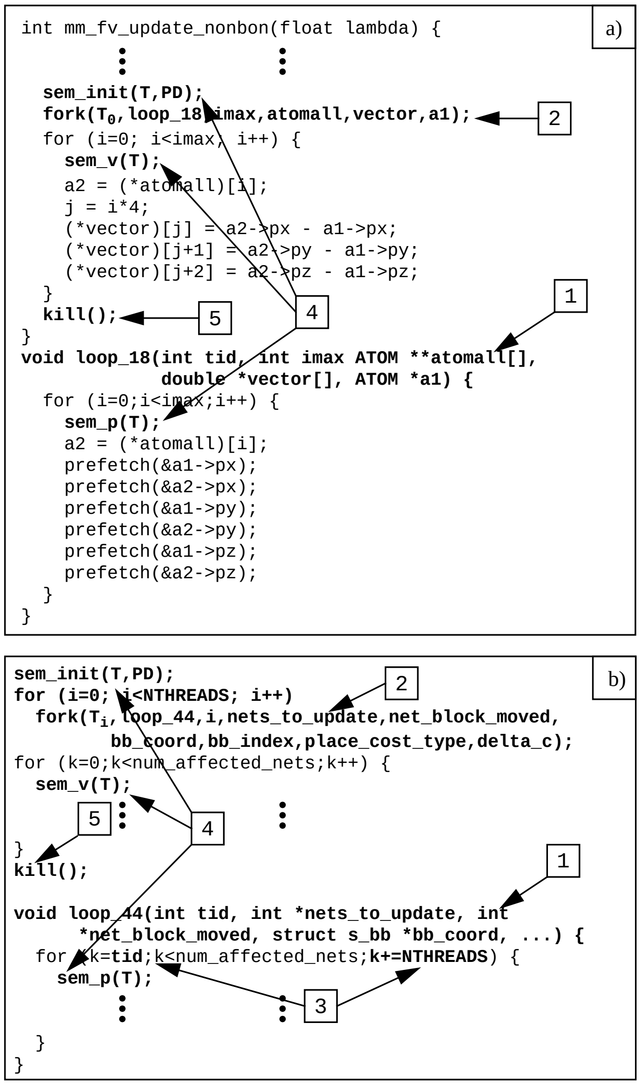
Figure 4.11 shows a code example where we apply the DOACROSS scheme to a next-outer loop, which consists of two nested pointer-chasing loops, from the TWOLF benchmark. As described in Section 4.1.2, our DOACROSS scheme produces two types of helper threads, the backbone and rib threads. The backbone thread performs the serialized update of the loop-induction variable, whereas the rib threads perform the loop-body execution in parallel with the backbone thread and other rib threads, thereby overlapping multiple memory accesses simultaneously. Generating codes for the DOACROSS scheme follows six steps. First, we clone the loop header and place it in a separate backbone procedure. Likewise, we clone the loop body and put it in a rib procedure (labeled "1"). Then for the rib thread code, we apply store removal and the same code optimizations from Figure 4.10. We also generate code to fork a single backbone thread in the main thread (labeled "2") and code to fork multiple rib threads in round-robin order in the backbone thread (labeled "3"), passing live-ins as required. As in Figure 4.10, we insert counting semaphores to synchronize threads. We insert a single semaphore, called "T_0," to keep the backbone thread from getting more than the "PD" iterations ahead of the main thread (labeled "4"), and multiple semaphores, one per rib thread, to synchronize each rib thread with the backbone thread during communication of the induction variable and live-in values (labeled "5"). Finally, we insert a `kill` directive (labeled "6").

In addition to cloning the loop headers and loop bodies as shown in Figures 4.10 and 4.11, we must clone any user procedure(s) as well, which are called from within the loop bodies (not shown in the figures) to accurately generate memory addresses for the delinquent loads. Once cloned, these procedures should also undergo store removal and code optimizations since they are part of the helper thread code and should not disrupt the main computation. For example, the `net_cost()` and `get_non_updateable_bb()` routines from Figure 4.2 should be cloned and optimized along with the loop from `try_swap()`. However, we cannot clone any system calls and library calls since the source code for those functions is not available.

4.3.6 ISA Support

Our SUIF-based compiler frameworks assume an SMT processor with following ISA support to enable pre-execution. First, we assume a `fork` instruction that specifies a hardware context ID and a PC. The fork initializes the program counter of the specified hardware context to the PC value and activates the context. Second, we assume `suspend` and `resume` instructions. These instructions are used to "recycle" threads for low overhead thread initiation as describe in Section 4.1.3. Both instructions specify a hardware context ID to suspend or resume. In addition, `suspend` causes a pipeline flush of all instructions belonging to the suspended context. While the processor state of a suspended context remains in the processor, the associated thread discontinues fetching and issuing instructions after the suspend and pipeline flush.[15] Third, we assume a `kill` instruction that halts all currently active helper threads as described in Section 4.1.3. Only the main thread can execute the `kill` instructions. Finally, we assume a `prefetch` instruction, which is a nonblocking and nonfaulting load instruction. Unlike prefetch instructions in some other platforms, our prefetch instruction has the same priority as the normal load instructions and the prefetch requests are never dropped.

[15]Note, however, those three instructions, `fork`, `suspend`, and `resume`, are not solely for pre-execution, but can be used to support user-level multithreading in general. While such functions can be also performed using the OS API, we do not consider using the OS to manage helper threads because our simulator cannot emulate the OS effect.

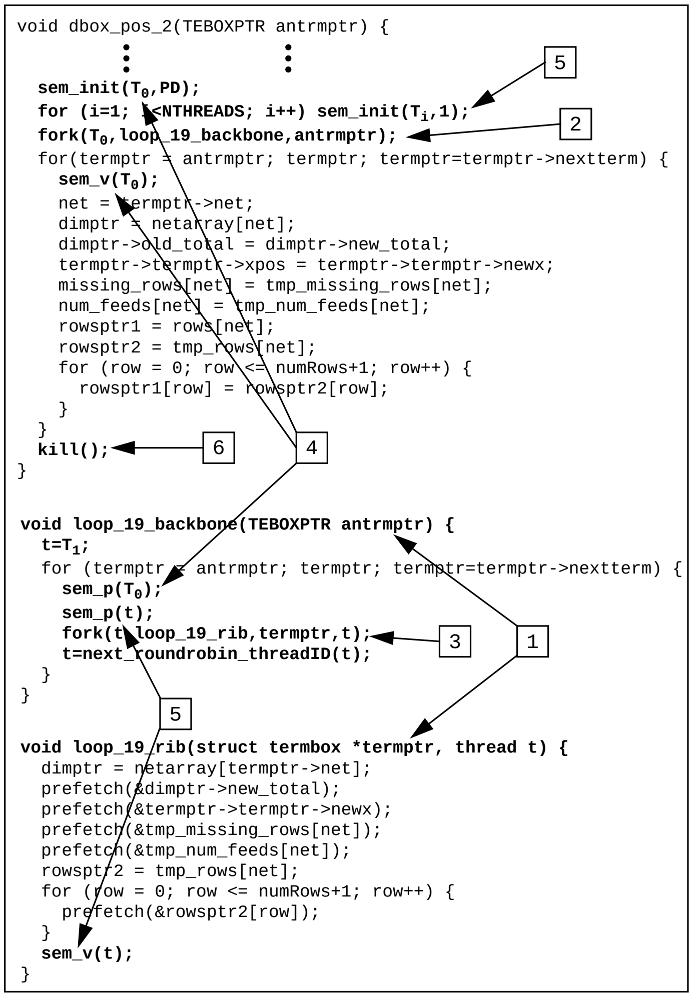


```
int mm_fv_update_nonbon(float lambda) {
          ⋮              ⋮
  sem_init(T,PD);
  fork(T0,loop_18,imax,atomall,vector,a1);
  for (i=0; i<imax; i++) {
    sem_v(T);
    a2 = (*atomall)[i];
    j = i*4;
    (*vector)[j] = a2->px - a1->px;
    (*vector)[j+1] = a2->py - a1->py;
    (*vector)[j+2] = a2->pz - a1->pz;
  }
  kill();
}
void loop_18(int tid, int imax ATOM **atomall[],
             double *vector[], ATOM *a1) {
  for (i=0;i<imax;i++) {
    sem_p(T);
    a2 = (*atomall)[i];
    prefetch(&a1->px);
    prefetch(&a2->px);
    prefetch(&a1->py);
    prefetch(&a2->py);
    prefetch(&a1->pz);
    prefetch(&a2->pz);
  }
}
```

```
sem_init(T,PD);
for (i=0; i<NTHREADS; i++)
  fork(Ti,loop_44,i,nets_to_update,net_block_moved,
         bb_coord,bb_index,place_cost_type,delta_c);
for (k=0;k<num_affected_nets;k++) {
  sem_v(T);
          ⋮              ⋮
}
kill();

void loop_44(int tid, int *nets_to_update, int
      *net_block_moved, struct s_bb *bb_coord, ...) {
  for (k=tid;k<num_affected_nets;k+=NTHREADS) {
    sem_p(T);
          ⋮              ⋮
  }
}
```

Figure 4.10: Code generated by the aggressive compiler to implement a) the Serial scheme for the AMMP benchmark and b) the DoAll scheme for the VPR benchmark. Code to initiate pre-execution and synchronize the helper threads appears in bold face.



```
void dbox_pos_2(TEBOXPTR antrmptr) {
          ⋮              ⋮
  sem_init(T0,PD);
  for (i=1; i<NTHREADS; i++) sem_init(Ti,1);
  fork(T0,loop_19_backbone,antrmptr);
  for(termptr = antrmptr; termptr; termptr=termptr->nextterm) {
    sem_v(T0);
    net = termptr->net;
    dimptr = netarray[net];
    dimptr->old_total = dimptr->new_total;
    termptr->termptr->xpos = termptr->termptr->newx;
    missing_rows[net] = tmp_missing_rows[net];
    num_feeds[net] = tmp_num_feeds[net];
    rowsptr1 = rows[net];
    rowsptr2 = tmp_rows[net];
    for (row = 0; row <= numRows+1; row++) {
      rowsptr1[row] = rowsptr2[row];
    }
  }
  kill();
}

void loop_19_backbone(TEBOXPTR antrmptr) {
  t=T1;
  for (termptr = antrmptr; termptr; termptr=termptr->nextterm) {
    sem_p(T0);
    sem_p(t);
    fork(t,loop_19_rib,termptr,t);
    t=next_roundrobin_threadID(t);
  }
}

void loop_19_rib(struct termbox *termptr, thread t) {
  dimptr = netarray[termptr->net];
  prefetch(&dimptr->new_total);
  prefetch(&termptr->termptr->newx);
  prefetch(&tmp_missing_rows[net]);
  prefetch(&tmp_num_feeds[net]);
  rowsptr2 = tmp_rows[net];
  for (row = 0; row <= numRows+1; row++) {
    prefetch(&rowsptr2[row]);
  }
  sem_v(t);
}
```

Figure 4.11: Code generated by the aggressive compiler to implement the DoAcross scheme for the TWOLF benchmark. Code to initiate pre-execution and synchronize the helper threads appears in bold face.

Chapter 5

Evaluation in a Simulation-Based Environment

In this chapter, we evaluate the performance of compiler-based pre-execution by generating the helper threads using our five SUIF-based compiler frameworks introduced in the previous chapter. The experimental evaluation is performed on a detailed architectural simulator of an SMT processor. We first provide the experimental methodology in Section 5.1, and then report the evaluation results for our aggressive and reduced compilers in Sections 5.2 and 5.3, respectively.

5.1 Experimental Methodology

For the experiments, we process a number of benchmarks and construct helper threads using the compiler frameworks from Chapter 4. The construction of helper threads, including the offline profiling and program slicing steps, is done fully automatically without any manual intervention. The generated binaries, to which the helper thread code is integrated, are executed on a time-accurate SimpleScalar-based SMT processor simulator. This section presents the configuration of our simulator and introduces the benchmarks we choose for our experiments.

5.1.1 Simulator Configuration

Our simulator faithfully implements a research SMT processor model proposed by Tullsen *et. al.* [76]. We improve the SMT processor simulator from Madon *et. al.* [46], which is originally derived from the SimpleScalar toolset's out-of-order processor simulator, *sim-outorder* [10]. Since the SimpleScalar toolset is well known in the research community and is assumed to be sufficiently reliable, we use the same execution units, register renaming logic, branch predictor, and cache models that are provided by the original SimpleScalar toolset. In addition, we modify the simulator so that it can support multiple hardware contexts as in an SMT processor. Therefore, we replicate the program counter, register file, and branch predictor[1] for each hardware context, and

[1]Providing branch predictor to each hardware context can be an expensive design choice. Although our experimental results are collected using per-context branch predictor, our later experiment, which is not presented in this thesis, shows that sharing the large second-level branch look-up table among all contexts does not affect the

Table 5.1: SMT processor simulator configuration.

Processor Pipeline			
Issue width	8-way	# hardware contexts	4
RUU size	128 entries	Instruction fetch queue	32 entries
Load-store queue	64 entries	Functional units	8 Int, 4 FP units
Int add/ mult/ div	1/ 3/ 20 cycles	FP add/ mult/ div	2/ 4/ 12 cycles
Branch Predictor			
Gshare predictor	2K entries	Return of stack	8 entries
Branch target buffer	2K entries, 4-way set-associative		
Memory Hierarchy			
Level 1 cache	Split I & D, 32KB, 2-way set-associative, 32B block, 1 cycle latency		
Level 2 cache	Unified, 1MB, 4-way set-associative, 64B block, 10 cycle latency		
Main memory access time	122 cycles		

modify the fetch logic to implement the ICOUNT fetch policy from [76]. Our simulator keeps track of the number of instructions in the reorder buffer (also known as the Register Update Unit or RUU in SimpleScalar) for each hardware context and assigns the fetch slots to a currently-active hardware context with the lowest ICOUNT. The simulator fetches instructions from a maximum of 4 hardware contexts every cycle. Except for the program counter, register file, and branch predictor, all other processor resources are shared among multiple hardware contexts. Note that our simulator does not model a stride-based hardware prefetcher, nor does the *gcc* compiler insert any software prefetch instructions into the original program code for prefetching. While some of our benchmarks may benefit from such conventional prefetching techniques, it should be pointed out that many of our targeted loops contain pointer dereferences or nested loop structures for which conventional stride-based prefetching and software prefetching are not usually effective. Finally, new instructions to support pre-execution, which are described in Section 4.3.6, are properly implemented as well. Upon detecting a `suspend` instruction at the decode stage, the processor stops

performance of pre-execution noticeably.

fetching from the suspended hardware context. Likewise, when a `resume` instruction is detected, the processor sets a proper bit, which denotes whether a hardware context is active or not, in order to restart fetching from the resumed hardware context. Table 5.1 reports our simulator configuration.

5.1.2 Benchmarks

In our simulation-based study, we evaluate 13 benchmarks from various benchmark suites, *i.e.*, the SPEC CINT2000 and CFP2000 suites [71], and the Olden benchmark suite [59]. Unfortunately, we could not evaluate all benchmarks in those 3 benchmark suites. 1 SPEC CINT2000 benchmark is written in C++ and the remaining 10 benchmarks in the SPEC CFP2000 benchmark suite are written in Fortran. Since Unravel can only process ANSI C programs, those benchmarks are excluded from our benchmark list. 4 other benchmarks in the SPEC CINT2000 benchmark suite are not evaluated; 2 benchmarks cannot be analyzed by Unravel, 1 benchmark is not handled by the SUIF, and the last benchmark performs system calls that are not currently supported by the SimpleScalar toolset. On the other hand, 8 benchmarks in the Olden benchmark suite are not evaluated since they perform recursive binary- or quad-tree traversals, which our current compiler frameworks cannot analyze.

Table 5.2 reports the benchmarks used in our study and 4 pieces of information for each benchmark. The column labeled "Input" reports the input sets used to run the benchmark. For the SPEC benchmarks, we choose the reference input set provided by SPEC [71], and for the Olden benchmarks, we select input parameters that are normally used for these benchmarks. The next two columns, labeled "FastFwd" and "Sim," show the number of skipped, or *fast-forwarded*, instructions before entering the detailed timing simulation region, and the number of instructions simulated under detailed timing, respectively. Note, the instruction counts include instructions executed outside the pre-execution regions as well as instructions within the pre-execution regions. Finally, the column labeled "IPC" presents the committed instructions per cycle on our simulator without performing pre-execution. As mentioned earlier, both profile and actual performance runs

Table 5.2: Benchmark characteristics.

Suite	Name	Input	FastFwd	Sim	IPC
SPEC CINT2000	256.bzip2	reference	186,166,461	123,005,773	1.3849
	175.vpr	reference	364,172,593	130,044,367	1.4039
	300.twolf	reference	124,205,135	112,809,146	1.1595
	254.gap	reference	147,923,793	127,932,004	3.3558
	197.parser	reference	245,277,302	126,593,730	1.9844
	181.mcf	reference	12,149,459,578	137,280,363	0.7914
	164.gzip	reference	162,442,542	135,592,391	1.9840
SPEC CFP2000	183.equake	reference	2,570,651,646	21,850,552	0.7586
	188.ammp	reference	2,439,723,993	129,357,604	1.3600
	179.art	reference	12,899,865,395	113,811,999	1.0900
	177.mesa	reference	262,597,404	54,117,618	2.8605
Olden Benchmarks	mst	1024 nodes	183,274,940	24,361,256	0.1153
	em3d	20K nodes	53,331,921	108,341,604	0.5929

use the same simulation regions and input sets so that our results do not account for discrepancies between the two runs.

5.2 Results of Aggressive Compiler

Having introduced the simulator configuration and benchmarks, we now report the experimental results for our aggressive compiler and provide detailed analyses. Section 5.2.1 presents various static and dynamic information of the helper threads constructed by our aggressive compiler. Section 5.2.2 reports the performance results as well as the cache miss coverage of compiler-based pre-execution. Section 5.2.3 breaks down the execution time to see the contribution of each compiler algorithm for effectiveness, *i.e.*, speculative loop parallelization, program slicing, and prefetch conversion, to the overall performance improvement. In Section 5.2.4, we apply three different thread initiation schemes, *i.e.*, Serial, DoAll, and DoAcross, to the pre-execution

Table 5.3: Static characterization of the helper thread code generated by our aggressive compiler.

Benchmark	Load	Lines	Slice	Store	Pref	SE	DA	DX	Back	Span
256.bzip2	28	62	0	6	9	2	1	0	-	0
175.vpr	32	318	130	14	21	1	2	0	-	1
300.twolf	55	132	23	35	25	1	0	5	0.24	0
254.gap	8	14	8	1	4	1	0	0	-	0
197.parser	5	51	0	11	0	1	0	1	8.58	2
181.mcf	26	69	0	30	10	0	0	1	41.8	1
164.gzip	3	23	0	1	0	0	0	1	57.7	0
183.equake	67	21	0	6	24	0	1	0	-	0
188.ammp	41	67	19	24	35	3	0	0	-	0
179.art	13	40	11	7	12	5	2	0	-	0
177.mesa	1	11	0	2	1	1	0	0	-	0
mst	4	34	16	1	0	0	0	1	0.00	1
em3d	13	12	2	0	7	0	1	0	-	0
TOTAL:	296	854	209	138	148	15	7	9	-	5

regions in selected benchmarks and show that our thread initiation scheme selection algorithm always chooses the best scheme for those benchmarks. Section 5.2.5 compares the performance results between the hardware and software mechanisms for thread synchronization and discusses tradeoffs. Finally, in Section 5.2.6, we carefully inspect each of our 31 pre-execution regions and show that our speculative loop parallelization and store removal rarely disrupt the accuracy of the generated helper threads.

5.2.1 Characterization of Helper Threads

We first present some static and dynamic characteristics of the helper threads generated by our aggressive compiler. Table 5.3 reports 10 static measurements collected from the constructed pre-execution code, and Table 5.4 reports 4 dynamic measurements gathered at runtime for each benchmark with pre-execution. We believe these measurements provide useful insights to better understand characteristics of compiler-generated prefetching helper threads.

In Table 5.3, we report 10 static measurements from the generated helper thread code. “Load” is the number of delinquent loads identified using offline memory profiles, “Lines” is the number of original source code lines within the pre-execution regions (it also takes into account lines in procedure calls), “Slice” is the number of source code lines eliminated by Unravel’s program slicing, “Store” is the number of source code lines removed as a consequence of store and system call removal in addition to program slicing (note, measurements for “Slice” and “Store” are mutually exclusive), and “Pref” is the number of delinquent loads converted into prefetches. Then the following 3 measurements, “SE” for Serial, “DA” for DoAll, and “DX” for DoAcross, report the number of pre-execution regions transformed using each thread initiation scheme, respectively. For the pre-execution regions parallelized by the DoAcross scheme, we also present an additional measurement, “Back,” which is the percentage of cache misses incurred in the backbone as opposed to the loop bodies, or ribs. Finally, “Span” shows the number of pre-execution regions that contain procedure calls and thus span multiple procedures.

From these measurements, we can see four important characteristics of our compiler-generated helper thread code. First, by looking at the “Load” column, we see only a small number of static loads, *i.e.*, less than 23 loads on average, account for more than 90% of the total L1 misses in each benchmark. Note, however, not all the identified loads are pre-executed since our compiler does not generate helper threads for those pre-execution regions that account for less than 3% of the total misses as mentioned in Section 4.3.2. Therefore, many delinquent loads that contribute to only a small portion of cache misses are discarded so that our compiler can concentrate on a small set of very important pre-execution regions that are responsible for a significant portion of the overall memory stall time. Second, program slicing and store removal together eliminate a large portion of static code within the pre-execution regions; across all benchmarks, 347 source code lines are removed out of 854 original source code lines or 40% of the static code on average. Among those removed codes, Unravel’s program slicing is responsible for 209 lines or 60% of the

Table 5.4: Dynamic characterization of the helper thread code generated by our aggressive compiler.

Benchmark	Forks	Inst-Fork	Inst-Pre	Inst-Miss
256.bzip2	252003	488	238.6	20.7
175.vpr	275816	1103	228.7	15.0
300.twolf	591343	460	45.6	6.6
254.gap	32739	3908	940.9	110.5
197.parser	4093	27.6K	511.5	13.6
181.mcf	3997152	68.6K	25.7	4.1
164.gzip	1478363	513	53.9	30.3
183.equake	3	21.9M	4.123M	3.2
188.ammp	15951	8110	3.2K	9.5
179.art	612	243K	128.9K	8.3
177.mesa	23750	2218	33.6	26.4
mst	393471	47.6K	53.0	4.9
em3d	300	1.08M	324.2K	2.6
AVG:		16.03K	1038	10.9

removed code while store and system call removal accounts for 138 lines or 40% of the removed code. Such massive code removal partly explains why our helper threads can quickly get ahead of the main thread and trigger cache misses early. Third, comparing the "Pref" and "Load" columns, we can see our compiler successfully converts about 50% of the identified delinquent loads into prefetches. This in turn enables our aggressive compiler to select the Serial scheme for 15 out of 31 pre-execution regions, thus using only a single helper thread to perform pre-execution for these regions. For the remaining 16 pre-execution regions for which our compiler could not convert all delinquent loads into prefetches, either the DoAll or DoAcross schemes is selected instead to overlap multiple blocking memory accesses and thus speedup the main thread. Finally, the "Span" column shows 5 out of 31 pre-execution regions span across multiple procedures, implying the inter-procedure analyses in both Unravel and the SUIF are useful for some of our benchmarks.

Table 5.4 reports 4 dynamic thread measurements gathered at runtime for each benchmark on our SMT processor simulator by launching the generated helper threads. “Forks” is the number of helper threads spawned. Note, for those pre-execution regions using the DoAll scheme, 3 helper threads are forked each time the main thread enters the targeted region. On the other hand, for pre-execution regions using the DoAcross scheme, 1 backbone thread is forked whenever the main thread enters the pre-execution region, but the backbone thread invokes the rib threads multiple times since a rib thread executes only one loop iteration and suspends itself. “Inst-Fork” is the number of instructions executed in the main thread between two consecutive pre-execution regions. This gives an idea about how frequently helper threads are launched in our benchmarks. “Inst-Pre” is the number of instructions executed by each helper thread for every fork. This measurement only includes instructions to generate memory addresses for the delinquent loads, and it does not account for instructions due to the thread synchronization and live-in passing. Finally, “Inst-Miss” is the number of helper thread instructions executed to trigger each cache miss in a program, which is computed by dividing the total instructions executed in helper threads by the total number of cache misses the helper threads trigger.

From these dynamic measurements, we can see two important characteristics of our compiler-generated helper threads. First, by looking at the “Inst-Fork” and “Inst-Pre” columns, we notice our helper threads are coarse-grained since the frequency of helper thread spawning is relatively low. Once invoked, our helper threads execute thousands of instructions before being suspended; this is a very large amount compared to other pre-execution techniques (*e.g.*, Roth and Sohi [63] or Zilles and Sohi [84]). In spite of adopting a thread recycling model as described in Section 4.1.3, spawning a helper thread is still expensive in our system because dispatching helper threads and passing live-in values are performed in software. Therefore, this type of coarse-grain thread management is especially helpful for minimizing such software overhead associated with thread startups, and we believe it will be even more crucial for higher thread synchronization cost. On the contrary, in many previous proposals on pre-execution techniques where helper threads are forked in hardware [6, 19, 20, 41, 48, 63, 64, 73, 84], fine-grain thread management is affordable. Second, the “Inst-Miss”

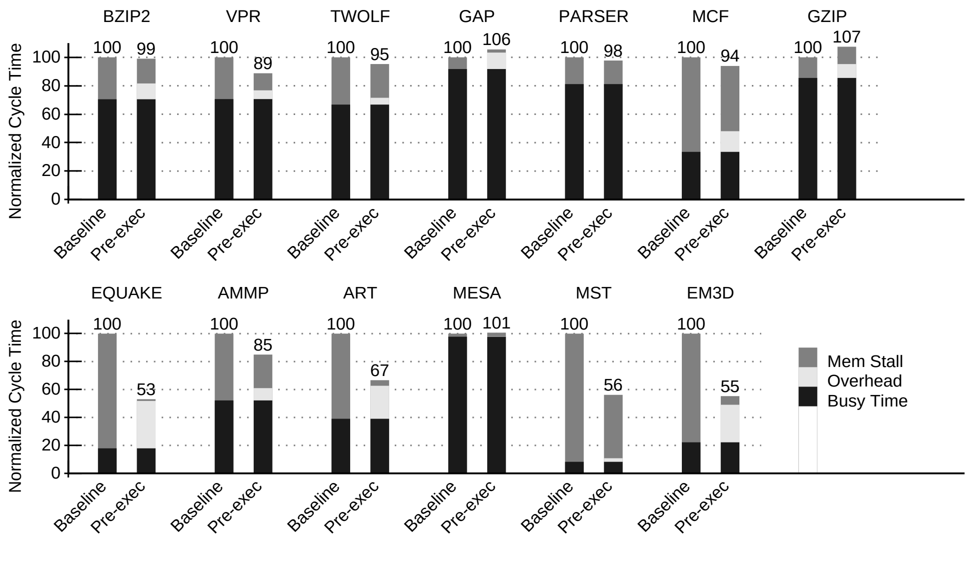


Figure 5.1: Normalized execution time broken down into the "Busy Time," "Overhead," and "Mem Stall" components. The "Baseline" and "Pre-exec" bars show the performance without and with pre-execution, respectively.

column shows our helper threads are very effective in triggering cache misses. To compute memory address for a delinquent load and issue a prefetch, our helper threads execute only a few instructions most of the time, 10.9 instructions on average across all benchmarks. This demonstrates that Unravel's program slicing successfully extracts minimum code necessary to compute the targeted delinquent loads.

5.2.2 Performance Results

Now, we report the performance results of our compiler-based pre-execution. Figure 5.1 presents the execution times for the 13 benchmarks we evaluate. The 7 applications in the top row are the SPEC CINT2000 benchmarks, whereas the 6 applications in the bottom row are from the SPEC CFP2000 and Olden benchmark suites. For each benchmark, we show two bars; the first bar, labeled "Baseline," is the execution time without pre-execution, which is normalized to 100, while the second bar, labeled "Pre-exec," is the execution time with pre-execution, which is normalized to the execution time of "Baseline." Each bar is broken down into three components. "Busy Time" is the execution time without performing pre-execution, assuming a perfect data memory system where every data cache-block request is serviced in 1 cycle, *i.e.*, L1 cache hit. Note, however, all I-cache accesses are handled normally. “Overhead” is the incremental increase in execution time over the “Busy Time” by performing pre-execution with a perfect data memory system. This component reflects the time spent executing the pre-execution related instructions in helper threads. Although the behavior of helper threads on a perfect memory system is a bit different from that of on a real memory system, we believe this component roughly shows the overhead associated with pre-execution. Finally, “Mem Stall” is the memory-stall time assuming a real memory system, which is the incremental increase in execution time over “Busy Time” + “Overhead.”

Figure 5.1 shows our aggressive compiler improves the performance of 10 out of 13 benchmarks. For those 10 benchmarks with speedup, the execution time with pre-execution is reduced from 1% up to 47%, providing 20.9% reduction in execution time on average. However, pre-execution degrades the performance of 3 benchmarks, GAP, GZIP, and MESA, by 4.7% on average. Overall, we achieve a harmonic mean of 17.6% speedup for all 13 benchmarks fully automatically using our aggressive compiler. It is noticeable that compiler-based pre-execution provides much larger speedup for the SPEC CFP2000 and Olden benchmarks (the bottom row in Figure 5.1) as compared to the SPEC CINT2000 benchmarks. This is partly because these integer applications in the SPEC CINT2000 suite spend less time on memory stalls compared to the other benchmarks.[2] In addition, pre-execution also incurs nontrivial runtime overhead; for all 13 applications, the “Overhead” component accounts for 11.9% of the “Baseline” time on average. In particular, EQUAKE, ART, and EM3D have a huge pre-execution overhead. This is because our software synchronization mechanism injects a large amount of garbage instructions into the processor pipeline due to busy waiting.

To better understand the effectiveness of compiler-based pre-execution as a form of prefetch-

[2]The 7 SPEC CINT2000 benchmarks spend 28.6%, on average, of their execution time on memory stalls and achieve 1.7% speedup. On the other hand, applications from the SPEC CFP2000 and Olden benchmark suites in the bottom row, except for MESA, spend an average of 72.0% of their time on memory stalls and achieve 58.2% speedup.

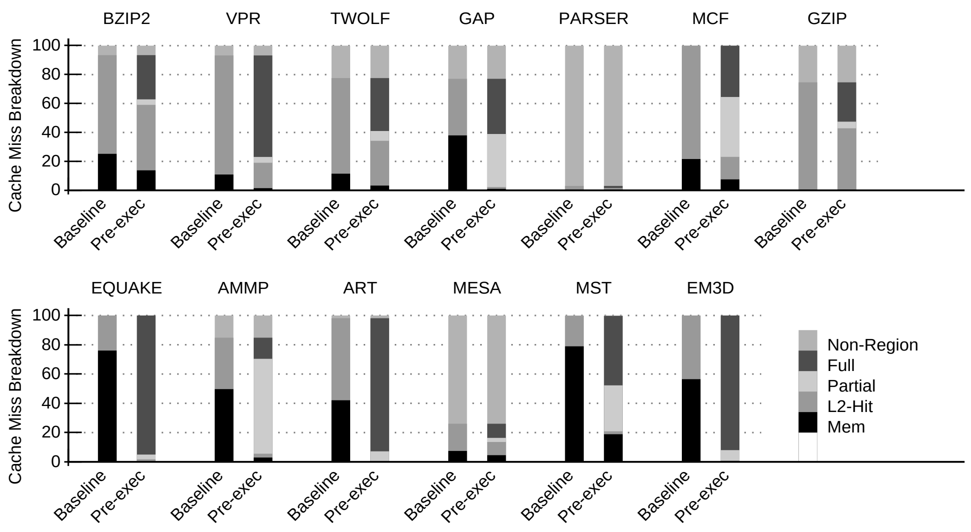


Figure 5.2: Cache-miss coverage broken down into five categories: uncovered cache misses occurring outside the pre-execution regions ("Non Region"), fully covered cache misses by pre-execution ("Full"), partially covered cache misses by pre-execution ("Partial"), remaining cache misses that hit in the L2 cache ("L2-Hit"), and remaining cache misses satisfied from main memory ("Mem").

ing, we report the cache-miss coverage in Figure 5.2, classifying the L1 misses into five categories. Again, each benchmark has two bars; the first bar, labeled "Baseline," is for the baseline run without pre-execution, and the second bar, labeled "Pre-exec," is with pre-execution. The "Baseline" bar is divided into three components: the L1 misses that occur outside the pre-execution regions (labeled "Non-Region"), misses that miss in the L1 cache but hit in the L2 cache (labeled "L2-Hit"), and misses that are served from main memory (labeled "Mem"). The "Pre-exec" bar has two more categories in addition to those three components; "Full" represents the fully covered cache misses that originally missed in the L1 cache but now hit in the L1 cache thanks to pre-execution, and "Partial" shows the partially covered cache misses that had been prefetched by helper threads but have not arrived at the L1 cache in time, thereby providing partial benefit to the main thread. To collect such statistics, we modify our simulator to track all the memory references issued from the helper threads and record the cache hit/miss status for the memory references from the main thread.

Figure 5.2 provides many insights. First, across all 13 benchmarks, 79.1% of the total L1 misses on average that occur in the original program are contained in the selected 31 pre-execution regions, and most of them are covered by pre-execution.[3] However, PARSER and MESA have 85.5% of cache misses in the "Non-Region" component. This is because our compiler does not select loops that account for less than 3% of the total cache misses (see Section 4.3.2), and PARSER and MESA contain lots of loops incurring a very small number of cache misses. Second, our aggressive compiler generates helper threads that are very effective in prefetching for many applications. For VPR, GAP, MCF, EQUAKE, AMMP, ART, MST, and EM3D, the helper threads convert from 74.2% to 99.1% of the main thread's cache misses, 84.9% on average across all 8 benchmarks, into either fully or partially covered cache misses. This clearly shows our compiler-based pre-execution satisfies all three conditions for effective prefetching: namely, timeliness, accuracy, and high cache-miss coverage. However, the miss coverage is somewhat lower for BZIP2, TWOLF, and GZIP, only 36.5% on average. By carefully examining each benchmark, we find two reasons for the reduced miss coverage. First, the helper threads issue memory references after the main thread has already accessed the corresponding load instances. This situation sometimes happens especially for short loops since it takes some time, or a few loop iterations, to initiate helper threads and allow them to get ahead of the main thread. Such late pre-execution accounts for some uncovered cache misses in BZIP2, VPR, and TWOLF that contains many pre-execution regions at the inner-most level with less than 10 loop iterations. Second, as explained in Section 5.2.6, some of our compiler optimizations can disrupt the correctness of the generated helper threads, thereby reducing the cache-miss coverage. This factor accounts for some uncovered cache misses in BZIP2, VPR, and MCF due to store removal, and in GZIP due to speculative loop parallelization.

5.2.3 Contributions of Algorithms

The previous section presents the performance impact and cache-miss coverage of our aggressive compiler, which applies all 3 compiler optimization algorithms, *i.e.*, program slicing, prefetch

[3]We stop identifying delinquent loads once the accumulate cache-miss percentage of the already selected loads exceeds 90% of the total cache misses. Therefore, not all cache-missing loads within a pre-execution region are pre-executed.

conversion, and loop parallelization. In this section, we examine the contribution of each individual algorithm to the performance improvement of the benchmarks. Figure 5.3 reports the experimental results for the 10 benchmarks that provide speedup with pre-execution (see Figure 5.1). In this experiment, we apply each optimization algorithm incrementally and measure the execution time.[4] Again, the first bar, labeled "Baseline," is the execution time without pre-execution. The second bar, labeled "Parallel," is the execution time where we apply speculative loop parallelization only, without program slicing and prefetch conversion. The third bar, labeled "Slicing," shows the execution time with program slicing as well as speculative loop parallelization, but still without prefetch conversion. Finally, for the fourth bar, labeled "Pre-exec," we perform all 3 optimization techniques including prefetch conversion and measure the execution time (this is the same as the "Pre-exec" bars in Figure 5.1). All 3 bars are normalized to the execution time of the "Baseline" bar. As in Figure 5.1, all bars are broken down into 3 components, "Mem Stall," "Overhead," and "Busy Time." Note, for "Parallel" and "Slicing" experiments, since prefetch conversion is not performed, a pre-execution region always contains some blocking loads, and thus the SERIAL scheme is never selected unless store removal converts all delinquent loads within the pre-execution region into prefetches (see Section 4.1.4). Therefore, the thread initiation scheme used for the first two experiments may be different from the one used for the "Pre-exec" experiments.

Figure 5.3 clearly shows speculative loop parallelization improves the performance of all benchmarks except for EM3D; this means it is very important to exploit thread-level parallelism in order to achieve speedup with pre-execution. In particular, for BZIP2, PARSER, and MCF, the entire performance gain is achieved solely by speculative loop parallelization, and it also accounts for most of the gain in TWOLF, EQUAKE, and MST. For those benchmarks, the execution time rarely changes after "Parallel" bar, which implies program slicing and prefetch conversion do not contribute much to the performance improvement. However, by comparing the "Parallel" and "Slicing" bars, we can see program slicing improves only 1 benchmark, *i.e.*, VPR. This is not too surprising because noncritical code unnecessary to compute the delinquent loads is often

[4]Store removal and code pinning are always performed for all the experiments to preserve the correctness of the main computation and to ensure the execution of delinquent loads by helper threads, respectively.

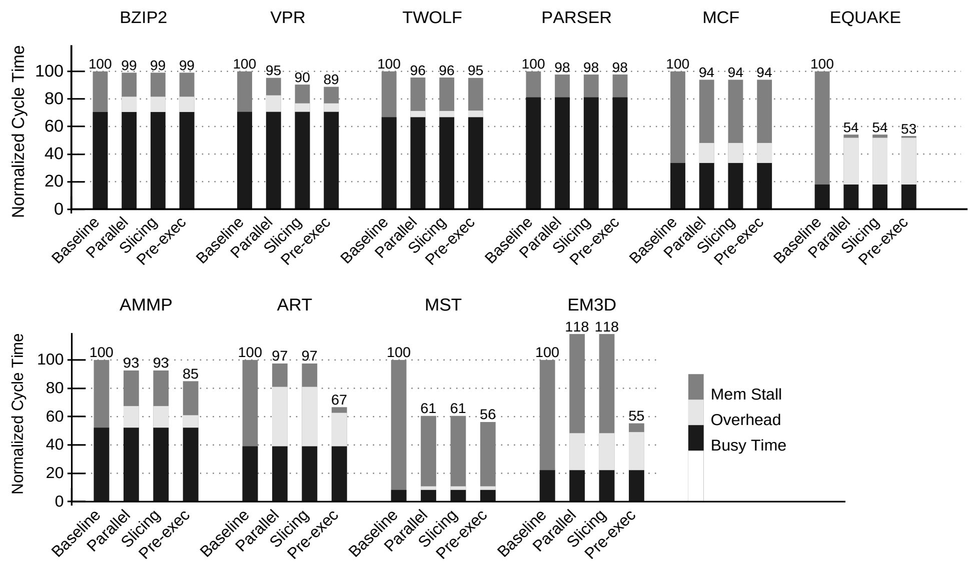


Figure 5.3: Impact of individual algorithms on overall performance in the aggressive compiler. The "Parallel" bars perform pre-execution with speculative loop parallelization only; the "Slicing" bars perform pre-execution with program slicing and speculative loop parallelization, but without prefetch conversion; and the "Pre-exec" bars perform pre-execution with all optimizations including prefetch conversion.

removed via dead-code elimination optimization in the backend compiler. Program slicing is effective for VPR because it contains a pre-execution region, which accounts for a large portion of the total execution time and spans multiple procedures. While Unravel performs slicing across the procedure boundaries, *gcc* does not perform dead-code elimination globally. Nonetheless, this observation that program slicing improves only 1 benchmark is strong motivation for eliminating Unravel from our compiler framework, which is further discussed in Section 4.2.1. Finally, going from the "Slicing" to "Pre-exec" bars, the performance of 7 benchmarks is improved thanks to prefetch conversion optimization. In fact, prefetch conversion accounts for most of the performance gain in ART and EM3D, and half the gain in AMMP. Prefetch conversion removes blocking load instructions that would be otherwise stalled in the reorder buffer, thereby speeding up the execution of helper threads. (More detailed discussion on the benefit of prefetch conversion is provided in Section 4.1.2.) Having demonstrated prefetch conversion plays a crucial role in improving a program's performance with pre-execution, program slicing becomes important as well because prefetch conversion is driven by the program slicing analysis in our compiler framework. In other words, although program slicing itself does not enhance the performance of many of our benchmarks, it indirectly contributes to the performance gain through prefetch conversion. Therefore, it is a still crucial component in our aggressive compiler.

5.2.4 Comparison of Thread Initiation Schemes

We examine the impact of helper thread initiation schemes on the performance and see whether our aggressive compiler chooses the right initiation scheme for the selected pre-execution regions. For this purpose, we choose 4 pre-execution regions from AMMP, EQUAKE, EM3D, and MST (1 region from each benchmark). Each selected pre-execution region has different blocking load characteristics and induction variable update patterns, as shown in Figure 5.4. First, in AMMP, prefetch conversion has removed all the blocking loads and converted them into prefetches, thus our compiler framework selects the SERIAL scheme to initiate a single helper thread for the region. On the other hand, the pre-execution regions in EQUAKE, EM3D, and MST still contain blocking loads, and one of the parallel schemes is chosen by the compiler. Concerning the induction variable, AMMP, EQUAKE, and EM3D have affine induction variables, whereas MST has a pointer-chasing induction variable. For each pre-execution region, we force our compiler to generate helper threads using all three thread initiation schemes, SERIAL, DOALL, and DOACROSS, regardless of the blocking load characteristics and induction variable type. However, for MST, since the induction variable is updated by dereferencing a pointer value, we cannot apply the DOALL scheme where the induction variable is updated independently and locally in each helper thread. To show the effectiveness of our thread-initiation scheme selection algorithm, the scheme selected by our aggressive compiler is denoted in bold face.

Figure 5.4 shows choosing the right initiation scheme makes a big difference in performance of pre-execution. For each pre-execution region, the best scheme outperforms the worst scheme by 25.9%, 39.8%, 51.3%, and 44.6% for AMMP, EQUAKE, EM3D, and MST, respectively. When compared to the second-best scheme, the best scheme still provides much larger gain, *i.e.*, 16.7%, 32.9%, 49.1%, and 44.6% for 4 benchmarks, respectively. Fortunately, our aggressive compiler

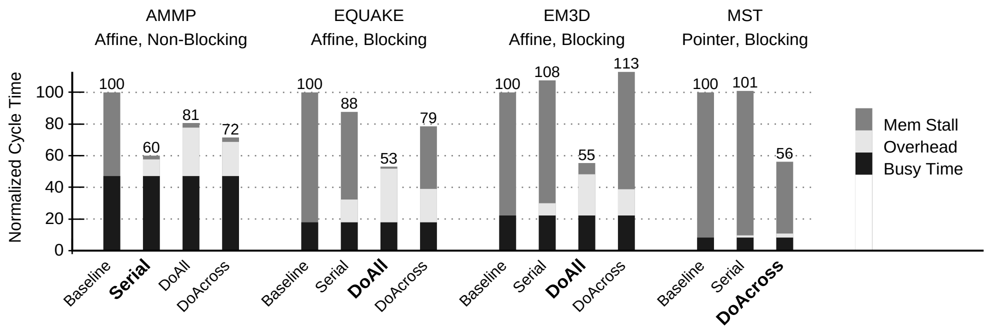


Figure 5.4: Comparing the helper thread initiation schemes. The "Baseline" bars report the execution time without pre-execution. The remaining bars report the execution time when the Serial, DoAll, and DoAcross schemes are applied, respectively. The thread initiation schemes selected by our compiler appear in bold face.

always chooses the best thread initiation scheme for all 4 pre-execution regions, demonstrating that the scheme selection algorithm presented in Section 4.1.2 is effective. We expect our compiler to select the right initiation scheme for other pre-execution regions as well that are not investigated in this section.

Next, let us consider why our scheme selection algorithm is likely to choose the best scheme most of the time. For AMMP, the Serial scheme performs best. Since prefetch conversion removes all the blocking loads and program slicing reduces number of instructions to be executed, even a single helper thread is easily able to get ahead of the main thread. On the other hand, while our two parallel schemes, DoAll and DoAcross, provide similar level of speed advantage to the helper threads, they must manage multiple helper threads and thus perform worse than the Serial scheme. Another important factor to consider is synchronization. In the absence of blocking loads, the helper thread in AMMP is likely to run ahead of the main thread most of the time, thereby spending much time on synchronization. While the Serial scheme needs only 1 synchronization point between the main thread and the single helper thread, the DoAll scheme requires 3 synchronization points between the main thread and the 3 helper threads. Moreover, this partly explains why the DoAcross scheme performs better than the DoAll scheme in AMMP since the DoAcross scheme has a maximum of 1 active synchronization point at any instance of the program execution. In other words, the backbone thread either synchronizes with the main thread or one of the rib threads, and it does not do both at the same time. Thread synchronization is inevitable to avoid run-away helper threads, but it also has a negative impact on the performance since a busy-waiting thread is likely to have the lowest ICOUNT, thereby occupying a large portion of the fetch units that could have been utilized by other threads.

For EQUAKE and EM3D, where pre-execution regions contain blocking loads and have affine induction variables, the DoAll scheme performs the best. In this case, it is obvious that a single helper thread cannot get ahead of the main thread due to the blocking loads, and thus the Serial scheme is not a better choice than the two parallel schemes. Between the DoAll and DoAcross schemes, the DoAll scheme can overlap more memory accesses than the DoAcross scheme, in which one helper thread, *i.e.*, the backbone thread, is solely dedicated to update the induction variable and communicates with the rib threads to pass the next induction variable value, which is totally unnecessary for affine loops. Finally, for MST, the DoAcross scheme outperforms the Serial scheme due to the remaining blocking loads in the helper thread.

5.2.5 Comparison of HW/SW Synchronization Mechanisms

Thus far, we have used software mechanisms for thread synchronization where a helper thread executes a busy-waiting loop until the main thread catches up, as described in Section 4.1.2. Since the busy-waiting helper threads sometimes slow down the execution of the main thread, this section also evaluates a hardware synchronization mechanism and compares the performance between the two synchronization mechanisms. We modify our simulator to implement such special hardware support for pre-execution.[5] We create 32 hardware registers for storing values of counting semaphores. Our counting semaphores work similarly to the hardware locking mechanism proposed in Tullsen *et. al.* [77]. We also assume a new ISA support, `Sem_P` and `Sem_V` instructions, to perform "P" and "V" operations on the semaphore registers, respectively. Both the `Sem_P` and `Sem_V` instructions have a single operand to specify the corresponding semaphore register. When a thread executes a `Sem_P` instruction, the specified semaphore register value is decremented by 1 if it is greater than zero; otherwise, the thread is blocked until the register value becomes non-zero.

[5]Note, however, hardware synchronization mechanism can be used for any multithreading techniques in general.

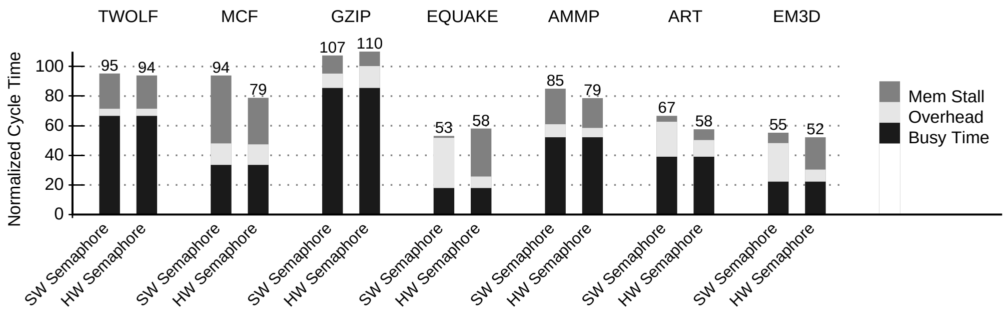


Figure 5.5: Impact of architectural support for synchronization. The “SW Semaphore” and “HW Semaphore” bars report the execution time with the software and hardware counting semaphores, respectively. All bars are normalized to the “Baseline” bars from Figure 5.1.

Each semaphore register is accompanied by a FIFO queue so that when multiple threads sleep on the same semaphore register, they are enqueued sequentially. When a thread is blocked on `Sem_P`, it causes a pipeline flush of all instructions that belong to the hardware context, as is done for the `suspend` instruction described in Section 4.3.6. On the other hand, a `Sem_V` instruction increments the specified semaphore register value by 1. If the FIFO queue of the semaphore register is not empty, the context at the head of the queue is resumed. To reduce the latency due to semaphore operations, we handle both the `Sem_P` and `Sem_V` instructions in the execution stage, and the resumed context can start fetching instructions the next cycle. This approach is similar to the speculative restart mechanism proposed in [77].

In Figure 5.5, we make our aggressive compiler generate two versions of pre-execution binaries, one with the software synchronization mechanism, labeled “SW Semaphore,” and the other with the hardware synchronization mechanism, labeled “HW Semaphore,” and report the execution time normalized to the “Baseline” time, which is not shown in the figure. Among 13 benchmarks, we present the performance of 7 benchmarks, TWOLF, MCF, GZIP, EQUAKE, AMMP, ART, and EM3D; the other 6 benchmarks are not shown since there is no difference in execution time between the two synchronization mechanisms. Figure 5.5 shows hardware mechanism improves the performance by 3.7% on average for the 7 benchmarks. Overall, hardware semaphores increase the speedup of pre-execution from 17.6% to 20.5% across all 13 benchmarks. This performance improvement is mainly because the hardware synchronization mechanism incurs significantly less overhead compared to the software mechanism, which injects a large number of useless instructions into the processor pipeline as a consequence of busy-waiting, and wastes hardware resources. However, in two cases, *i.e.*, GZIP and EQUAKE, the hardware mechanism actually degrades the performance. In GZIP, the constructed helper threads are incorrect due to aggressive loop parallelization as to be described in Section 5.2.6. With the hardware mechanism, the helper threads follow the wrong path even faster, thereby limiting the execution of the main thread. On the other hand, in EQUAKE, the helper threads also run faster with the hardware mechanism and bring data cache blocks into the cache quickly. While cache thrashing can be a part of the reason for performance degradation, we also observe that helper threads hold too many load-store queue entries themselves, thereby limiting the progress of the main thread.

5.2.6 Accuracy of Helper Threads

To ensure the generated helper threads issue correct addresses for the delinquent loads, our compiler frameworks adopt code cloning, thereby making helper threads execute the same code that the main thread executes. While program slicing and prefetch conversion optimizations never affect the correctness of the constructed helper threads as already mentioned in Section 4.1.4, two of our transformation techniques, *i.e.*, store removal and speculative loop parallelization, can possibly disrupt the correctness of the helper threads, resulting in useless prefetches. Store removal can eliminate code that affects data flow or control flow leading to the delinquent loads. Speculative loop parallelization may inadvertently parallelize serial loops. As described in Section 4.1.2, when parallelizing loops, our compiler only analyzes the loop induction variables and ignores potential loop-carried dependences through the loop body to enable aggressive loop parallelization. Whenever such true dependences need to be executed to correctly compute delinquent loads, but compiler transformations violate the dependences, then the generated helper threads can be wrong and issue inaccurate addresses for the delinquent loads, degrading the effectiveness of pre-execution. To quantify how often our compiler transformations compromise the correctness of helper threads, we inspect all the constructed helper threads manually and check whether store removal or speculative loop parallelization has a negative impact on the helper threads.

First, we observe store removal causes a problem for 3 out of 31 total pre-execution regions selected by our aggressive compiler from 13 benchmarks. One case is from VPR and it is already discussed in Section 4.1.2. The other 2 cases occur in MCF and BZIP2. Fortunately, even for these 3 cases, store removal does not completely disrupt the correctness of the helper threads, but it only affects a few load instances, generating inaccurate addresses for them. For the remaining 28 pre-execution regions, store removal does not harm the helper threads at all. Based on this data, we conclude store removal rarely compromises the correctness of the generated helper threads. To see why our conclusion is valid, we closely look at each pre-execution region and examine how addresses for the delinquent loads are calculated. We find, for most of the cases, control and data flow leading to delinquent loads usually involves the loop induction variables.[6] For instance, the address computations for memory references "1" and "4" in Figure 4.2 depend on the induction variables from loops "7" and "8." Since a loop induction variable is used for every loop iteration, it is usually kept as a register value, not a global or heap variable that is stored in memory. Thus the update of an induction variable mostly occurs via computation of the register values only, and it rarely involves values in memory. Since the register and stack variables are not affected by our store removal, the memory address computations are not disrupted most of the time.

While store removal causes only a small impact on the correctness of the constructed helper threads, we find our speculative loop parallelization has an even smaller impact, generating inaccurate helper threads for only 1 pre-execution region from GZIP out of 31 total selected regions. To better understand the impact of speculative loop parallelization over all our benchmarks, Figure 5.6 presents the breakdown of our pre-execution regions, showing three bars categorized by parallelizability of loops. The first bar reports the number of pre-execution regions that can be correctly parallelized by either one of our two parallel thread initiation schemes, DoAll and DoAcross. In other words, these pre-execution regions are fully parallel, and thus they can be

[6]This observation becomes the basis of our reduced algorithm to statically identify delinquent loads in a program as demonstrated in Section 4.2.3. Our simple heuristics is that a load whose address computation involves the loop induction variable is likely to miss in the cache.

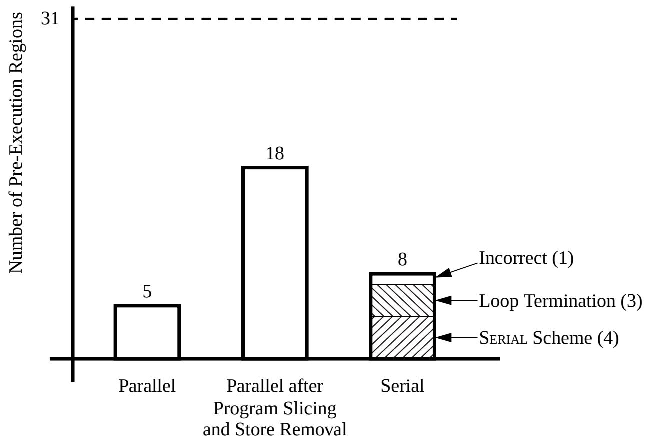


Figure 5.6: Breakdown of the pre-execution regions into 3 major categories: originally parallel, parallel after program slicing and store removal, and serial. The serial category is further broken down into regions using the SERIAL scheme, regions speculatively parallelized affecting control flow only, and regions that are incorrectly parallelized.

easily parallelized even by a conventional parallelizing compiler. Only 5 out of 31 pre-execution regions belong to this category, which emphasizes that the loops we deal with in this study contain complex dependence chains, so it is not easy to extract thread-level parallelism from these loops using traditional loop parallelization methods. However, as shown in the second bar of Figure 5.6, additional 18 pre-execution regions become parallel after program slicing and store removal are applied. This shows most loop-carried dependences exist in the code that has nothing to do with the memory address computations, and our compiler optimization techniques effectively identify those unnecessary codes and remove them from the generated helper threads. Therefore, those *cache-missing kernels* that are left after program slicing and store removal have abundant thread-level parallelism, which cannot be uncovered by conventional parallel processing or thread-level speculation techniques. Finally, the third bar in Figure 5.6 denotes the remaining 8 pre-execution regions that are serial even after program slicing and store removal. We further break this bar down into three categories. Out of 8 pre-execution regions, 4 employ the SERIAL scheme since our compiler was able to convert all blocking loads into prefetches, and thus these loops are not parallelized, preserving the correctness of the helper thread. On the other hand, 3 of the serial pre-execution regions are parallelized anyways because they still contain blocking loads after prefetch conversion

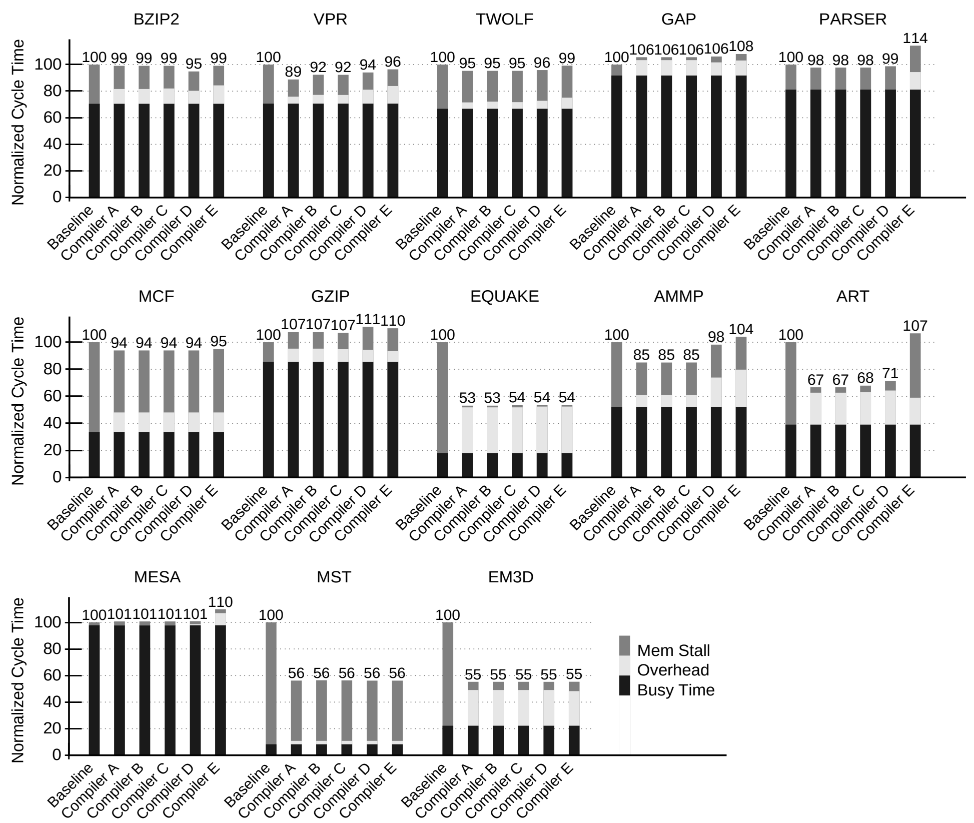


Figure 5.7: Normalized execution time broken down into the "Busy Time," "Overhead," and "Mem Stall" components. Groups of bars labeled "Compiler A" to "Compiler E" report performance for the 5 compilers from Table 4.1.

optimization. However, for these 3 cases, only the loop termination condition is broken, and the helper threads still generate correct addresses. Hence the cache miss coverage is not reduced by loop parallelization, and the `kill` instruction guarantees to terminate those over-executing helper threads after the main thread exits the pre-execution region. Finally, the remaining 1 pre-execution region is from GZIP, which is already explained.

5.3 Results of Reduced Compilers

Having evaluated our aggressive compiler in the previous section, we now evaluate our 4 reduced compiler frameworks. First, Section 5.3.1 compares Compiler A, our most aggressive compiler, and Compiler B, which removes Unravel, and discusses the impact of eliminating program slicing on performance. Section 5.3.2 evaluates the performance of Compiler C where prefetch conversion is done in the SUIF in the absence of the program slicing information. Then Sections 5.3.3 and 5.3.4 evaluate Compilers D and E, respectively, and discuss the performance of pre-execution without using offline profiles. The performance results for all these evaluations appear in Figure 5.7 across the 13 benchmarks. (Note, we use the software synchronization mechanism for the experiments.)

5.3.1 Impact of C Compiler's Code Removal

By comparing the execution time of Compilers A and B in Figure 5.7, we see eliminating program slicing in our aggressive compiler degrades the performance of only 1 benchmark, *i.e.*, VPR, and the other 12 benchmarks are not affected at all. Compiler B provides a speedup of 17.3%, on average, across all 13 benchmarks, which is almost the same as that of Compiler A, 17.6%. This means the dead-code elimination in the backend code optimizer is as effective as Unravel's program slicing in removing noncritical code. Only for those applications like VPR where pre-execution regions span multiple procedures, the powerful code analyses in Unravel trim down more code in helper threads. The result here is not surprising given the experiments of Figure 5.3 (contribution of the optimization algorithms) in Section 5.2.3. In Figure 5.3, comparing the "Parallel" and "Slicing" bars isolates the impact of program slicing, and we can see that program slicing does not affect the performance of pre-execution except for VPR; most of the code removed by program slicing is also removed during the backend compilation.

While performance is probably the most important metric to determine the merit of a technique, we believe the instruction count of helper threads also provides very useful information in that it helps to estimate the power consumption and hardware resource requirement of helper threads. Figure 5.8 reports three bars for each benchmark: the dynamic instruction count *committed* in the main thread, labeled "Main Thread," and the one in the helper threads from two different compilers, labeled "Compiler A" and "Compiler B," respectively. To focus on the code to compute the delinquent loads, we only count instructions within the pre-execution regions and do not account for other instructions outside the pre-execution regions. Note, the only difference between Compilers A and B is that the latter does not have a separate program slicing step; every-

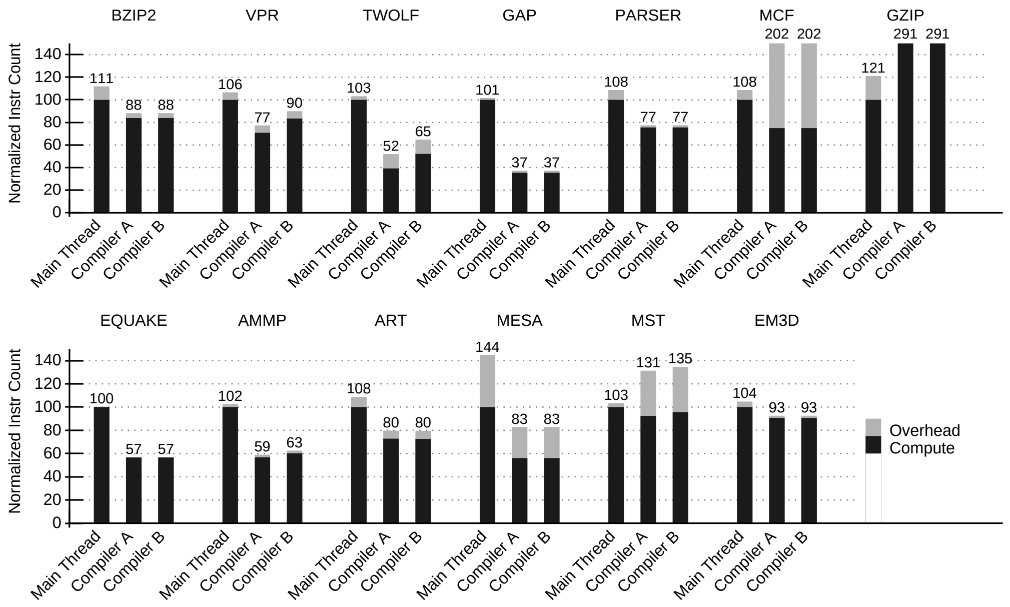


Figure 5.8: Comparison of dynamic instruction counts between Compilers A and B. The "Main Thread" bars report the dynamic instruction counts for the main thread. The "Compiler A" and "Compiler B" bars report the dynamic instruction counts for the helper threads generated by Compilers A and B, respectively. Each bar is broken down into the "Compute" and "Overhead" components.

thing else is exactly the same such as the identified delinquent loads, corresponding pre-execution regions, thread initiation scheme for each pre-execution region, and live-in variables to be passed from the main thread to the helper threads. Hence the instruction count for the main thread, "Main Thread" bar, is same for both Compilers A and B. In Figure 5.8, each bar is broken down into two components, "Overhead" and "Compute." The "Overhead" component is the instruction count associated with pre-execution, and not a part of the original program, *e.g.*, forking helper threads, passing live-ins, and thread synchronization. To avoid confusion, we ignore instructions associated with busy waiting and only consider instructions to check the relative distance between threads or availability of helper threads. The "Compute" component represents the instructions from the original program. All bars are normalized to the instruction count of the "Baseline".

Figure 5.8 shows both Compilers A and B effectively remove noncritical computation instructions from the helper threads (see the "Compute" component) except for GZIP. In GZIP, due to incorrect loop parallelization, the helper threads follow incorrect paths, thereby executing instructions that should not have been executed (see Section 5.2.6). For the remaining 12 benchmarks, Compilers A and B remove 32.9% and 30.2% of noncritical computation instructions in the main thread, respectively. Taking into account the overhead instructions as well, helper threads still execute much fewer instructions than the main thread for 10 benchmarks. In MCF and MST, the helper threads execute more instructions than the main thread. This is mainly because the backbone thread incurs high overhead to pass arguments and fork the rib threads.

Comparing the "Compiler A" and "Compiler B" bars in Figure 5.8, we see no difference between the two compilers for 9 benchmarks. This implies the backend compiler's dead-code elimination removes the same code that is removed by Unravel. Therefore, even though the generated output files at the source-code level by the two compiler frameworks are different, the final binaries after the backend compilation are same.[7] For VPR, TWOLF, AMMP, and MST, however, Unravel is not redundant to dead-code elimination and the dynamic instruction count for Compiler A is smaller than that of Compiler B; for those 4 benchmarks, Compiler A eliminates 8.1% more dynamic instructions than Compiler B. We identify two reasons that explain the difference between Unravel's program slicing and dead-code elimination. First, for many cases, the dead-code elimination analysis in *gcc* compiler is not as effective as that of Unravel, still leaving many instructions in the binary that are successfully identified as dead code and removed by Unravel. This occurs especially when the code contains pointer accesses. Second, in the existence of inter-procedure calls, Unravel performs an extraordinary job to identify the dead code compared to the backend compiler because Unravel performs inter-procedure analysis over the entire program. When a procedure call is encountered, Unravel separates the dependent arguments from the independent arguments, and cuts down dependences in the callee function associated with those function parameters. On the other hand, the scope of backend code optimizations is often limited within a procedure boundary, and thus the backend compiler conservatively assumes all arguments communicated across the procedure boundary are live. Therefore, Compiler B can miss some opportunities to remove

[7]This leaves some possibility that the generated binaries can be sometimes different depending on the backend compiler. However, since dead-code elimination is one of the basic optimizations supported by most compilers, we believe such a scenario is not likely to happen.

dead code whenever pre-execution regions contain procedure calls. In spite of the effectiveness of Unravel-based program slicing for certain circumstances, it is unclear whether we should keep Unravel in our compiler frameworks. Even for the 4 cases where Unravel is not redundant, only 1 case provides performance improvement while the performance remains the same for the other 3 cases. The answer will become clearer depending on whether we can successfully perform prefetch conversion via static compiler algorithms without using program slicing information, which we discuss in the next section.

5.3.2 Impact of SUIF-Based Prefetch Converter

In Compiler B, although the program slicing step is eliminated, prefetch conversion is still driven by Unravel; in other words, the same set of delinquent loads are converted into prefetches in both Compilers A and B. To completely remove Unravel from our compiler frameworks, we implement static compiler algorithms, described in Section 4.2.2, in Compiler C to identify the candidate loads for prefetch conversion. Comparing the "Compiler B" bars against the "Compiler C" bars in Figure 5.7, we see our SUIF-based algorithms provide almost the same performance as the Unravel-driven prefetch conversion; the performance of 11 benchmarks remains the same, and even for the two benchmarks with some differences, *i.e.*, EQUAKE and ART, the change in the execution time is hardly noticeable. On average, Compiler C provides 17.1% overall speedup across all 13 benchmarks, whereas Compiler B just slightly outperforms it, providing 17.3% speedup. This proves our simple algorithms, shown in Eqs. 4.3 and 4.4 in Section 4.2.2, are sufficient to drive prefetch conversion and largely equivalent to Unravel's complicated analysis for program slicing. Thus we can safely remove Unravel from our compiler framework completely without much degradation to the performance.

5.3.3 Impact of Eliminating Cache-Miss Profiles

In Compiler D, we remove the offline memory profiles and rely on simple compiler algorithms, described in Section 4.2.3, to statically identify delinquent loads in a program. In Figure 5.7, comparing the "Compiler C" and "Compiler D" bars, we can see eliminating the cache-miss profiles has some negative impact on performance, reducing the overall speedup from 17.1% for Compiler C to 15.0% for Compiler D. One reason for this performance degradation is that our static compiler algorithms falsely identify too many delinquent loads.[8] The number of pre-execution regions identified across the 13 benchmarks has doubled from 31 loops for Compilers A, B, and C to 62 loops for Compiler D even though we cancel the second pass in our pre-execution region selection algorithm that picks the remaining short inner-most loops (see Section 4.2.3). Since pre-execution inevitably incurs some runtime overhead, targeting too many pre-execution regions may be undesirable, especially when those selected regions do not contain many *actual* delinquent loads. Moreover, in the absence of the cache-miss profiles, we cannot apply our filtering mechanism to discard loops that account for less than 3% of the total cache misses as in Section 4.3.2. However, despite such drawbacks of eliminating the cache-miss profiles, the performance is not degraded significantly for most benchmarks; the only noticeable case is AMMP where the execution time is increased by 15.3% when going from Compilers C to D. We believe this is partly because most of our important pre-execution regions are still selected, and those regions already account for a large portion of the execution time in each benchmark. Therefore, even though we falsely identify many of new pre-execution regions, they often incur tiny amounts of overhead and have little impact on the overall execution time.

While static compiler algorithms often end up identifying too many pre-execution regions, they also identify too many delinquent loads within each pre-execution region. As is already discussed in Section 3.1.1, identifying the right delinquent loads that indeed incur the majority of critical cache misses is important to produce high-quality helper threads. The drawback of identifying too many delinquent loads is twofold. First, since the helper threads cover more loads, less code is removed by program slicing, thereby slowing down the execution of the helper threads and in turn reducing the benefit from pre-execution. Second, since more code remains in the helper threads, fewer loads are converted into prefetches, increasing the frequency of blocking in the helper threads and also reducing the speed of the helper threads. In fact, several pre-execution regions

[8]However, we observe most of the delinquent loads identified by the cache-miss profiles are still successfully identified by our reduced algorithms.

that are transformed with the Serial scheme in Compilers A, B, and C are pre-executed using either the DoAll or DoAcross schemes in Compiler D due to insufficient prefetch conversion. With the increased runtime overhead for thread management, we lose much of the performance gain we got previously. This phenomenon explains the performance degradation in AMMP and ART.

Surprisingly, the performance of BZIP2 is improved, providing 4% additional gain with Compiler D. There are two reasons for this performance improvement. First, BZIP2 contains many small loops that account for less than 3% of the total cache misses and are discarded in Compilers A, B, and C. Pre-executing some loops in that category helps to boost the performance. On the other hand, there is one inner-most loop that accounts for a large portion of the cache misses but does not get any benefit from pre-execution. Even worse, pre-executing that loop degrades the performance of BZIP2 since the loop iterates a small number of times. In Compiler D, however, since we do not perform the second pass of our pre-execution region selection algorithm to include such small inner-most loops, the cumbersome loop is not selected and it slightly helps to improve the performance.

5.3.4 Impact of Eliminating Loop Profiles

Lastly, we evaluate Compiler E, in which the loop-trip count profiles are eliminated. Comparing the "Compiler D" and "Compiler E" bars in Figure 5.7, we see eliminating the loop profiles degrades the performance of 9 benchmarks, reducing the overall speedup from 15.0% to 7.7%, on average, across all 13 benchmarks. The execution time of ART is most significantly affected; when going from Compilers D to E, the execution time is increased by 50.7%, resulting in 7% slowdown compared to the "Baseline." In ART, many important loops within the simulation region iterate thousands of times and they are mostly inner-most loops. In the absence of the loop-trip count profiles, our loop selection algorithm in Compiler E (see Section 4.2.4) always assumes the inner-most loops iterate fewer than 25 iterations, so it looks for the next-outer loops for pre-execution regions. Since one of the inner-most loops with huge trip count in ART is selected at the next-outer loop level in

Compiler E, it results in generating run-away helper threads due to lack of synchronization inside the inner-most loop body. Other than this case, our simple heuristics for estimating the amount of loop work is correct for the majority of cases; as mentioned earlier, for 83% of the inner-most loops in our benchmarks, our assumption about loop-trip count is correct.

To better understand the reason for performance degradation in Compiler E, we investigate each benchmark by closely looking at the identified pre-execution regions. In BZIP2, VPR, TWOLF, PARSER, AMMP, and MESA, the main reason for the performance degradation turns out to be the increased runtime overhead due to too many falsely identified pre-execution regions. Since Compiler E runs through the second pass in our pre-execution region selection algorithm to ensure pre-execution of all important inner-most loops as described in Section 4.2.4, many pre-execution regions that contain falsely identified delinquent loads are selected and partly responsible for the reduced gains. Compiler E identifies 115 pre-execution regions across all 13 benchmarks, which is significantly greater than that of Compiler D, 62 pre-execution regions.

5.4 Summary

In this chapter, we present various experimental results for our SUIF-based compiler frameworks. These results provide several valuable insights into compiler-based pre-execution. First, the performance result of our aggressive compiler demonstrates that compiler-based pre-execution is indeed a very promising data prefetching technique, providing 17.6% speedup across 13 benchmarks that are popularly used in the research community. We also show our helper threads successfully cover a large portion of the cache misses for the majority of benchmarks. Second, we set up experiments to identify the contribution of each individual compiler algorithm to the performance improvement. We show speculative loop parallelization and prefetch conversion are crucial to achieve good speedup with pre-execution. Moreover, we show our thread initiation scheme selection algorithm always chooses the best scheme to initiate the helper threads, and we provide detailed analyses on the rationales behind the accuracy of our algorithm. Third, the performance comparison between the hardware and software mechanisms for thread synchronization denotes the synchronization overhead has a negative impact on pre-execution effectiveness, and thus it should be lowered. However, we also show there exist tradeoffs between the two mechanisms, *i.e.*, overhead vs. introduction of new hardware support. Fourth, we perform manual inspection to examine how program slicing and store removal help to parallelize our pre-execution regions and show that our compiler-based pre-execution provides a new way to extract thread-level parallelism, especially to exploit memory-level parallelism in a program. Finally, from our detailed evaluation of the reduced compilers, we demonstrate we can sustain the performance gain from the aggressive compiler even without Unravel by replacing program slicing with the compiler's backend code optimizer and Unravel-driven prefetch conversion with simple SUIF-based compiler algorithms. We also eliminate the cache-miss profiles and the loop-trip count profiles, but better algorithms are required to properly identify delinquent loads and to estimate the loop work via static analyses, thereby reducing falsely identified targets and the runtime overhead associated with pre-execution.

Chapter 6

Pre-Execution Optimization Module in the Intel Research Compiler Infrastructure

We have demonstrated the algorithms and design of the SUIF-based compiler frameworks to construct effective prefetching helper threads. Our experimental results on an SMT processor simulator show that compiler-based pre-execution is indeed a promising prefetching technique to tolerate the ever-increasing memory latency. Beyond the simulation-based evaluation of a technique, we believe it to be very important to evaluate prefetching helper threads in a real physical system and identify critical issues to achieve a significant wall-clock speedup on real silicon. To perform such physical-system-based evaluation of compiler-based pre-execution, we first need a compiler to construct helper threads. We work with a group of people at Intel and develop an optimization module in the Intel compiler to generate helper threads, targeting the Intel Pentium 4 processor with Hyper-Threading Technology that can support two logical processors simultaneously.

This chapter describes the pre-execution optimization module in the Intel compiler and discusses the related issues to enable pre-execution in the physical system. While the optimization module in the Intel compiler and our SUIF-based compiler frameworks have many similarities, they are also different in some aspects due to the unique characteristics of the physical system. Section 6.1 examines some critical issues that cause differences in designing a compiler for pre-execution. Section 6.2 presents the compiler algorithms in the Intel compiler to enable pre-execution and compares them with the algorithms in the SUIF-based compiler frameworks. In Section 6.3, we introduce user-level library routines, called EmonLite, to monitor the program behavior at runtime with low overhead. Such a light-weight performance monitoring mechanism is necessary to enable fine-grain dynamic throttling of helper threads, thereby maximizing the performance gain from pre-execution. Finally, Section 6.4 discusses the implementation issues to build the compiler optimization module for pre-execution in the Intel compiler.

Table 6.1: Hardware resource management in the Intel's hyper-threaded processor.

Shared	Trace cache, L1 D-cache, L2 cache, Execution units, Global history array Allocator, Uop retirement logic, Microcode ROM, IA-32 instruction decode Instruction scheduler, Instruction fetch logic, DTLB
Duplicated	Per logical processor architectural state, ITLB, Instruction pointers Rename logic, Streaming buffers, Branch history buffer, Return stack buffer
Partitioned	Uop queue, Memory instruction queue, Reorder buffer, General instruction queue

6.1 Unique Aspects of a Physical System

Before presenting the compiler algorithms in the Intel compiler that are developed to enable pre-execution, let us first examine some unique aspects of the physical system that make differences in the design of a pre-execution compiler. We discuss three issues regarding hardware resource management, thread synchronization mechanism, and available hardware contexts in the processor, and see how such issues cause differences between the SUIF-based compiler frameworks and the optimization module in the Intel compiler.

6.1.1 Hardware Resource Management

Unlike the research SMT processor proposed in [75] where most microarchitectural structures are replicated or shared among multiple hardware contexts, the hardware resources in the Intel's hyper-threaded processor are managed differently. As detailed in [47], a hyper-threaded processor dynamically operates in one of the two modes: the ST (Single Threading) mode and the MT (Multi-Threading) mode. In the ST mode, all the on-chip resources are given to a single application thread, whereas, in the MT mode, these resources are shared or partitioned between the two logical processors. As shown in Table 6.1, structures like caches and execution units are shared between the two logical processors, very much like the resource sharing on the research SMT processor. In addition, structures like the ITLB and return stack buffer are replicated for each logical processor. On the other hand, structures like the reorder buffer are evenly hard-partitioned to prevent one logical processor from taking up the whole resources.

Such differences in the hardware implementation affect the scheduling of helper threads since the runtime overhead of pre-execution is much different for the two platforms. We assume the hardware resource sharing in the research SMT processor is near-optimal, though not perfect, and thus running a helper thread does not significantly affect the performance of the main thread. However, due to the hardware partitioning of certain critical resources in the MT mode, running a helper thread in the hyper-threaded processor often prevents the main thread from acquiring enough hardware resources and even degrades the application performance when the helper thread does not help. Therefore, invocation of a helper thread should be done very judiciously, and we must perform dynamic transitions between the ST mode and the MT mode to minimize the performance degradation due to the resource contentions. This is a strong motivation to propose a dynamic scheme to initiate helper threads and develop certain performance monitoring tool so that the helper threads can adapt to the dynamic program behavior.

6.1.2 Thread Synchronization Mechanism and Cost

The thread synchronization mechanism is also different between the research SMT processor and the Intel's hyper-threaded processor. In our SMT processor simulator, none of the structures is hard-partitioned, and thus there is no need for an explicit mode transition when invoking or suspending a thread. Therefore, we assume the processor is always running in the multithreading mode, and the thread synchronization between the main thread and the helper threads incurs a very low overhead; there is only a 3-cycle latency for pipeline flush when a thread is suspended, and the `resume` instruction restarts fetching from the corresponding thread immediately. On the other hand, the thread synchronization in our physical-system experimentation involves the mode transition between the ST and MT modes so that the main thread can fully utilize the entire processor resources when the helper thread is not running. Hence the overhead of thread synchronization is much larger; it takes thousands of cycles to invoke and suspend threads in the Intel's hyper-threaded processor. Considering a single prefetch reduces at most a few hundred cycles by converting a memory access into a cache hit, we cannot afford such fine-grain thread synchronization as our SUIF-based compilers support. Therefore, we propose a new thread initiation scheme, called StripMining, in the Intel compiler to support rather coarse-grain synchronization between the main thread and the helper thread.

In our experiments on the Intel's hyper-threaded processor, we evaluate two mechanisms to invoke and suspend the helper threads. First, we use the Win32 API, *i.e.*, `SetEvent()` and `WaitForSingleObject()`, provided by the Windows OS for thread management. When a thread calls `WaitForSingleObject()`, the Windows scheduler waits until the CPU utilization of the corresponding logical processor falls down below certain percentage. Only then does the OS deschedule the suspended thread and trigger a mode change from the MT mode to the ST mode. The latency between the moment when the thread calls for suspension and the occurrence of the mode transition is non-deterministic, and is a few tens of thousand CPU clock cycles.[1] On the other hand, when an active thread calls `SetEvent()` while another thread sleeps on `WaitForSingleObject()`, the OS initiates the mode transition from the ST mode to the MT mode and wakes up the dormant thread. We observe it takes a similar range of cycles to complete this mode transition.

To lower the thread switching and synchronization overhead, we also evaluate a hardware mechanism that prototypes user-level thread synchronization instructions similar to the lockbox primitives described in [77]. Note, this hardware mechanism is actually implemented in real silicon as an experimental feature. Using the hardware synchronization mechanism, a thread issues certain instructions to suspend itself and directly cause the MT to ST mode transition. Conversely, another thread can execute an instruction to wake up the suspended thread and cause the ST to MT mode transition. Using these direct hardware synchronization primitives, we measure the thread suspension takes approximately 1,500 cycles, which achieves one order of magnitude reduction as compared to the cost of the OS API. Moreover, this hardware mechanism is entirely transparent to the OS, and its latency is very deterministic.

[1]We observe the latency varies from 10K to 30K cycles for the processor that we use, but we do not have any statistical data regarding how much the mode transition latency fluctuates.

6.1.3 Available Hardware Contexts

The SMT processor simulator used to evaluate the SUIF-based compiler frameworks in Chapter 4 assumes a maximum of 4 threads can run simultaneously by sharing the processor resources. To exploit these spare hardware contexts, we developed parallel thread initiation schemes like DoAll and DoAcross that require multiple hardware contexts to run the helper threads. Moreover, as shown in Section 5.2.3, speculative loop parallelization plays a very important role to improve the performance of our selected benchmarks using pre-execution, and it uncovers abundant thread-level parallelism with the support from program slicing and store removal. However, the current Intel Pentium 4 processor with Hyper-Threading Technology supports two logical processors, so only 1 hardware context is available for helper threads, making it infeasible to apply parallel schemes. Therefore, the optimization opportunities for the compiler module in the Intel compiler is significantly reduced, and we generate a single helper thread for each pre-execution region.

6.2 Compiler Algorithms for the Intel Compiler

In this section, we describe the compiler algorithms for the pre-execution optimization module that we built into the Intel compiler. As is done for the SUIF-based aggressive compiler in Section 4.1, we group those algorithms into 4 categories. Section 6.2.1 describes how we identify the target delinquent loads and proper pre-execution regions, and Section 6.2.2 presents algorithms to improve the effectiveness of compiler-based pre-execution. Then Section 6.2.3 illustrates the need for performance monitoring and dynamic thread throttling to reduce the runtime overhead of pre-execution. Finally, Section 6.2.4 discusses issues regarding how to ensure pre-execution does not disrupt the correctness of the main computation.

6.2.1 Identifying Targets

As in our SUIF-based aggressive compiler (see Section 4.1.1), we use offline memory profiles to identify delinquent loads in a workload and loop-trip count profiles to choose pre-execution regions at the right loop-nesting level. In addition, we clone the target code region and create a separate code for helper threads so that we can further optimize the code.

Delinquent Load Identification

The first step in the helper thread construction is to identify the top delinquent loads that significantly degrade the performance of a workload. We run the workload on the Intel VTune^{TM} performance analyzer [79] and collect the clock cycle and L2 miss profiles. The VTune supports a convenient graphical user interface and provides summary profiles for each process, module, procedure, source line, and instruction. During the profile run, the hardware Performance Monitoring Counters (PMCs) are incremented whenever the specified performance events occur, and with certain frequency, the VTune samples those performance events for each static instruction in the workload. The VTune profiling run uses the same input set as is used for the later performance measurements, but we find different input sets do not significantly change the set of identified delinquent loads for our benchmarks, shown in Table 7.2. In addition, the application program is compiled with the same compiler optimization options as the "baseline" binary except that the "-Zi" flag is additionally used to insert debugging information, which does not affect the performance or behavior of the workload at all. Furthermore, the profile information is collected across the entire program execution.[2] After reading in the VTune profiles, the compiler backend optimizer correlates the data with the program's intermediate representation using the source line numbers. By analyzing the profile information, our compiler module calculates the cycle cost, or memory stall time, associated with each static load in the program and chooses the delinquent loads that account for a large portion of the entire execution time so that helper threads can be constructed to pre-execute those loads.

Pre-Execution Region Selection

Once the target delinquent loads are identified, our compiler module forms a region within which the helper thread is constructed. As in our SUIF-based compilers, we also choose loops for the pre-

[2]As to be explained later in Section 7.1.3, we measure the wall-clock time of the "entire" program execution to evaluate the performance of pre-execution for our physical-system-based study.

execution regions. When selecting pre-execution regions, the compiler module takes into account several issues to maximize the performance gain with pre-execution. As shown in Section 6.1.2, the cost of thread synchronization in a physical system, which involves the mode transition between the ST and MT modes, is in the range of thousands of cycles. Thus a key criterion in selecting a proper loop candidate for the pre-execution region is to minimize the runtime overhead associated with the thread management. One goal is to minimize the number of helper thread invocations, which can be accomplished by ensuring the trip count of the outer loop that encompasses the currently-processed loop is small. Therefore, one condition in our loop selection algorithm is that we keep searching for the next-outer loop until the next-outer loop's trip count is less than twice the trip count of the currently-processed loop. However, in our SUIF-based compiler frameworks, this issue was not considered at all since invoking a helper thread in our SMT processor simulator occurs immediately without incurring any overhead. On the other hand, another complementary goal is to ensure that the helper thread, once invoked, runs for an adequate amount of time in order to amortize the thread activation cost. Hence it is desirable to choose a loop that iterates a large number of times, or contains enough work. Therefore, the second condition in our loop selection algorithm is to keep searching for the next-outer loop until the loop-trip count exceeds a threshold value, currently one thousand iterations. Recall that the threshold value in the SUIF-based compilers was only 25 iterations for inner-most loops so that the helper threads have enough time to get ahead of the main thread. Note, the analysis starts from the inner-most loop that contains the delinquent loads and proceeds toward the outer-most loop. When the analysis reaches the outer-most loop within the procedure boundary, the search ends and the loop is selected. In other words, we do not allow our pre-execution regions to span across procedure boundaries. The loop-trip count information required for the pre-execution region selection algorithm is collected via the VTune's instruction count profiles. We observe this simple algorithm is effective, especially in the presence of the aggressive function inlining performed by the Intel compiler.

Code Cloning

As in the SUIF-based compilers, the compiler module in the Intel compiler also performs code cloning to construct a separate code for helper threads. As is already explained in Section 3.1.3, the separation between the main thread and helper thread codes ensures the transformation of the helper thread code does not affect the correct execution of the main thread as long as all side effects are removed from the helper thread. Hence it allows our compiler to optimize the helper thread code very aggressively, which would have been impossible if the compiler had not generated a separate code for helper threads.

We use multi-entry threading in the Intel compiler [74] to generate separate codes for helper threads. Unlike the SUIF-based compilers that create new subroutines for the pre-execution regions, the compiler module in the Intel compiler generates a threaded entry and a threaded return for each helper thread, and keeps all newly generated codes inlined within the corresponding original routine that contains the pre-execution region. Since the targeted-loop code and the helper thread code are within the same procedure boundary, all local variables are visible to both codes. Therefore, variables necessary to the helper threads, but not changed within the pre-execution region, do not have to be passed explicitly. Moreover, this method also provides later compilation steps with more opportunities to optimize the original code and the newly generated helper thread code together.

6.2.2 Improving Effectiveness of Pre-Execution

To produce effective prefetching helper threads, our pre-execution optimization module in the Intel compiler performs program slicing to remove noncritical code in the helper threads. Unlike the SUIF-based compilers, however, we do not perform prefetch conversion to reduce blocking in the helper threads because prefetch instructions are often dropped in case of memory system congestion in the Pentium 4 processor. In addition, we do not support any synchronization between the main thread and the helper thread due to high thread synchronization cost in the physical system. Due to lack of synchronization, helper threads can easily run too far ahead of the main thread and cause cache thrashing. To avoid such an undesirable situation, we propose three schemes to initiate helper threads in Section 6.2.2, which effectively work as the thread synchronization mechanisms.

Program Slicing

Our compiler module performs program slicing to identify instructions to be executed in the helper threads. This is necessary to filter out noncritical code from the pre-execution region, and thus provide a speed advantage to the helper thread over the main thread. Our program slicing is similar to the backward slicing presented in [87] and it consists of three steps, *i.e.*, building a dependence graph, slicing, and code pinning, as explained below.

Building Dependence Graph. To enable slicing, our compiler module builds a graph that captures both data and control dependences in the program code. The effectiveness of slicing relies on the compiler's ability to accurately identify true dependences and disambiguate memory references. Thus we invoke the memory disambiguation module in the Intel compiler, which disambiguates pointers to dynamically allocated objects [28]. Note, the analysis is not limited within the procedure boundary, but spans across multiple procedures. When building the dependence graph, a series of array dependence tests is also performed [23] so that each element in an array is disambiguated, reducing falsely identified dependences. Furthermore, each field in a structure is disambiguated as well.

Slicing. To perform slicing, our compiler module chooses the identified delinquent loads as the *slice criteria* [40, 82] based on the VTune memory profiles. Then the remaining slicing step is similar to the slicing in the SUIF-based compiler frameworks as in Section 4.1.2. Within the selected loop, the compiler module starts from each slice criterion and traverses the dependence edges backwards. Only the statements that affect the address computation of the delinquent loads, including the change of control flow, are selected as a slice. Slicing analysis terminates at the boundary of the predetermined pre-execution region so that the slice, which depends on the code outside the pre-execution region, is not included. To preserve the correctness of the main computation, all the stores to heap objects or global variables are removed from the slice. Once slicing is completed for all the individual delinquent loads, the extracted slices within the same loop boundary are merged to form a single thread, thereby reducing the thread invocations.

Code Pinning. Due to store removal during the slicing step, the extracted slice has no side effects and thus it is dead code by definition. Since the generated helper threads are scheduled and optimized by the remaining phases in the Intel compiler, passes such as global scalar optimizations using partial redundancy elimination will remove all codes with no side effects including the constructed helper threads. To avoid the removal of the helper thread codes in later compilation phases, we perform code pinning that is similar to what was shown in Section 4.1.2 for the SUIF-based compilers. Our compiler module searches for all delinquent loads that are leaf nodes in the dependence graph, on which no further instructions depend. Then it converts all such delinquent loads into the *volatile-assign* instructions in the Intel compiler, thereby preventing the subsequent compilation phases from removing the delinquent loads in the helper threads.

Prefetch Conversion

To further improve the performance of pre-execution, it is important to remove blocking in helper threads. Especially, since the current Intel Pentium 4 processor with Hyper-Threading Technology supports two logical processors, only one hardware context is available for the helper thread while the other logical processor is used to run the main thread. Since a single helper thread should get ahead of the main thread, the prefetch conversion optimization will be more crucial in the Intel compiler than in our SUIF-based compilers, which assumes the target SMT processor supports a maximum of 4 hardware contexts, thereby allowing multiple helper threads to run in parallel. Though nonblocking and nonfaulting, the prefetch instruction currently supported in the Pentium 4 processor is dropped when the memory subsystem is congested and normal load requests need to be serviced. Hence prefetches issued by the helper threads are often nullified; in addition to losing chances to improve the program performance through prefetching, we even degrade the performance due to the hardware resource contentions in the hyper-threaded processor. Therefore, we decide not to implement the prefetch conversion step in our current compiler module. Note, however, prefetch conversion can be easily enabled in the Intel compiler; we can simply convert those delinquent loads that are leaf nodes in the dependence graph into nonblocking prefetches.

Helper Thread Initiation

Unlike the research SMT processor used for the performance evaluation of our SUIF-based compilers, there is only one hardware context available for running helper threads in the hyper-threaded processor. Hence we do not implement any optimization pass to parallelize the pre-execution regions such as the DoAll and DoAcross schemes shown in Section 4.1.2. Instead, we simply rely on the Serial scheme to initiate a single helper thread for each pre-execution region. However, as mentioned earlier, we do not support any synchronization between the main thread and the helper thread due to the large thread synchronization overhead in the physical system. Thus the Serial scheme may produce a run-away helper thread if the targeted loop iterates a large number of times. To provide certain synchronization support, we introduce a derivative of the Serial scheme, called StripMining, in which the targeted loop is partitioned into several small chunks and the helper threads are invoked frequently. Moreover, to minimize the ineffectiveness of the StripMining scheme, we propose yet another thread initiation scheme, called Dynamic StripMining. Below, we illustrate some interesting tradeoffs between these three helper thread initiation schemes used in our physical experimentation.

1. **Serial.** We evaluate the Serial scheme, which was also used in our SUIF-based compiler frameworks (see Section 4.1.2). In this scheme, a helper thread is activated at the entrance point of the targeted loop and runs through all loop iterations without any further intermediate synchronization between the main thread and the helper thread. In other words, the inter-thread synchronization only occurs *once for every instance of the targeted loop.* Using the Serial scheme minimizes the invocation of helper threads and thus it can be useful especially when the thread synchronization cost is high as in the Intel's hyper-threaded processor-based system. However, due to lack of intermediate thread synchronization throughout the

entire loop execution, the helper thread can either run too far ahead of the main thread and pollute the cache, or run behind the main thread and waste computation resources which could have been used more effectively by the main thread.

2. **StripMining.** To avoid run-away or run-behind helper threads, it is necessary to perform inter-thread synchronization at a granularity finer than the size of the targeted loop instance. This motivates a new thread initiation scheme, called StripMining, in which a helper thread is invoked *once for every few iterations of the targeted loop.* The number of loop iterations between two consecutive helper thread invocations is the *sampling period.* In this scheme, once a helper thread is activated, it executes either for the number of loop iterations equal to the size of the sampling period, or until it reaches the termination of the targeted loop.

   Since the StripMining scheme invokes helper threads more frequently than the Serial scheme, the effectiveness is more sensitive to the thread synchronization cost of the target system. Furthermore, this scheme requires additional code to be instrumented in the targeted loop to check how far the main thread or the helper thread has executed within the sampling period iterations. In effect, this approach relies upon the sampling period as the synchronization boundary to frequently cross-check relative progress between the main thread and the helper thread. The size of the sampling period binds the distance by which a helper thread can run ahead of or behind the main thread. Therefore, the effectiveness of this approach also depends on the choice of the sampling period.

3. **Dynamic StripMining.** As to be illustrated later in Figures 6.2 and 6.3, a program's behavior varies at different chronological phases. Consequently, helper threads may not always be beneficial, *e.g.*, the main thread sometimes incurs very few cache misses even in a pre-execution region, and thus running a helper thread is useless. Even when the main thread suffers from many long-latency cache misses, the effectiveness of helper threads still depends on a variety of resource-related issues such as the availability of execution units, occupancy of the reorder buffer, memory bus utilization, or fill buffer (*i.e.*, Miss Status Holding Register, or MSHR) usage.

To avoid activation of helper threads when they are not helpful, we evaluate a dynamic thread initiation scheme, called Dynamic StripMining, which is a derivative of the StripMining scheme. In this scheme, rather than invoking a helper thread for every sample instance, the main thread dynamically decides whether or not to invoke a helper thread for a particular sample period. In fact, this is a dynamic throttling mechanism that monitors the pre-execution effectiveness at runtime and applies judicious control on both activation and termination of the helper thread. To evaluate the performance impact of dynamic throttling of helper threads, we assume a hypothetical perfect throttling mechanism. In other words, we do not actually implement any mechanism that monitors the dynamic program behavior and throttles the activation of helper threads based on the runtime profiles. Instead, we postprocess the cycle and L2 miss profiles collected by actually running benchmarks with and without helper threads, and estimate the potential performance improvement with a perfect throttling mechanism. More details will be presented in Section 7.1.5.

6.2.3 Minimizing Ineffectiveness of Pre-Execution

Due to hard-partitioning of some critical resources and the high thread synchronization cost, overhead is a more critical issue in a physical system with the hyper-threaded processor. Combined with those issues in the physical system, invoking helper threads for the time phases with small cache misses may be unhelpful to the main thread and thus have a negative impact on the program's performance. Our Dynamic StripMining scheme is developed to avoid performance degradation due to unhelpful helper threads. To achieve that goal, it is crucial for us to be able to monitor the dynamic program behavior. Such a performance monitoring mechanism should be light-weight so that it does not adversely affect performance. In addition, the performance profiles must be easily correlated with the program semantics and in turn, with the corresponding helper threads. To satisfy these conditions, we develop user-level library routines for light-weight performance monitoring, called EmonLite. These library routines can be easily integrated into the program code to collect various performance-related information at runtime for certain program regions.

Section 6.3 provides more details about the design, implementation, and example usage of the EmonLite.

6.2.4 Preserving Program Correctness

Like most other pre-execution techniques, the compiler module in the Intel compiler also removes all store instructions to the heap memory region and global variables as well as system calls within the pre-execution regions. Store and system call removal is performed during the slicing step as shown in Section 6.2.2. Note, however, both our compiler module and the hyper-threaded processor neither assume nor implement any support for handling exceptions signaled by helper threads.[3] Fortunately, the helper threads constructed for our benchmarks do not incur any exception signals. This is mainly because we do not parallelize the pre-execution regions as in the SUIF-based compilers due to lack of available hardware contexts for helper threads in the hyper-threaded processor, thereby preserving the data and control flow of the targeted loop in the helper threads. In addition, within our pre-execution regions, memory address computations for the delinquent loads do not include any store operation to the shared memory region, and thus store removal does not disrupt the correctness of the helper threads.

6.3 EmonLite: User-Level Library Routines for Runtime Performance Monitoring

To permit helper threads to adapt to dynamic program behavior, thus reducing the runtime overhead of pre-execution, we need a light-weight mechanism to monitor dynamic events such as cache misses at a very fine granularity. Some existing tools such as *pixie* [66] instrument the program code to use a cache simulator to *simulate* the dynamic behavior of a program. However, this methodology often fails to accurately reflect the actual program behavior on a physical system and usually slows down the program execution. To remedy these issues, we introduce EmonLite, a library of user-level routines which performs light-weight profiling through the direct use of the

[3]Exception handling is not taken care of since we evaluate the compiler module for helper threading as an experimental feature in the Intel compiler. However, for productization of such a compiler optimization module, we need certain mechanisms for exception handling so that a faulting helper thread never disrupt the main computation.

performance monitoring events supported in the Intel processors. Provided as a library of compiler intrinsics, the EmonLite allows a compiler to instrument at any location of the program code to directly read from the hardware PMCs. Therefore, event statistics such as clock cycles or L2 misses can be collected for a selected code region at a very fine granularity with high accuracy and low overhead. Note, the Intel processors support performance monitoring of over 100 different microarchitectural events associated with the branch predictor, trace cache, memory, bus, and instruction events. Compiler-based instrumentation via EmonLite enables collection of the chronology of certain instruction's dynamic events for a wide range of workloads. Such profiling infrastructure can be leveraged to support dynamic optimizations such as dynamic throttling of both helper thread activation and termination.

6.3.1 EmonLite vs. Intel VTune Performance Analyzer

Before providing the implementation details of the EmonLite, let us compare tradeoffs between the Intel VTune performance analyzer and EmonLite, and discuss why we need yet another performance monitoring tool. The VTune profiles various performance monitoring events over the entire program execution or predefined time duration. It provides profile data ranging in scope from process, to module, procedure, source line, and even assembly instruction level. However, the VTune collects only a summary of sampling profiles, so it is difficult to extract a dynamic chronological behavior of a program from the VTune's profiles. In contrast, the EmonLite provides relatively fine-grain chronology of the performance monitoring events, and thus it enables analysis of time-varying behavior of the workload at a fine granularity and allows dynamic adaptation and optimization. In addition, the EmonLite is a library of user-level routines and can be directly placed in the program source code to discriminately monitor the dynamic microarchitectural behaviors for the judiciously selected code regions. Lastly, while the VTune's sample-based profiling relies on the buffer overflow of the PMCs to trigger an event exception handler registered at the OS, the EmonLite routines read the counter values directly from the PMCs by executing four assembly instructions. Consequently, the EmonLite is extremely light-weight; it rarely slows down the user program provided that the profiling sampling interval, *i.e.*, how frequently the PMCs are read, is reasonably sized.

6.3.2 Components of EmonLite

The EmonLite library provides two compiler intrinsics that can be easily inserted into a user program code. One of the routines, called `emonlite_begin()`, initializes and programs a set of EMON-related internal processor registers to specify the performance monitoring events, for which profiles are collected. This library routine is inserted at the beginning of the user program and is executed only once for the entire program execution. The other intrinsic, called `emonlite_sample()`, reads the counter values from the PMCs and is inserted in the user code of interest. Since the PMC provides performance event information for each logical processor, in order to ensure accuracy of profiling, the target application is pinned to a specific logical processor via the OS affinity API, such as `SetThreadAffinityMask()` in the Win32 API.

6.3.3 Implementation Issues

Automatic instrumentation of the EmonLite library routines is also implemented in the Intel pre-production compiler. For a selected code region, the following steps are taken to generate the chronology of performance monitoring events.

Step 1. Same as the previously described compiler analysis phases, delinquent loads are identified first, then appropriate loops surrounding the delinquent loads are selected.

Step 2. The compiler inserts the instrumentation code into the user program. It searches for the `main()` function of the user program[4] and inserts `emonlite_begin()` to initialize the EmonLite and set up corresponding PMCs.

[4]Our compiler module for EmonLite code instrumentation assumes programs written in C language only and thus we can always find the `main()` function in the program code. Note, however, our compiler module for helper thread optimization is not limited to a specific language since all analyses and optimizations are done on the Intel compiler's common intermediate representation.

```
while (arcin) {
  /* emonlite_sample() */
  if (!(num_iter++ % PROFILE_PERIOD)) {
    cur_val = readpmc(16);
    L2miss[num_sample++] = cur_val - prev_val;
    prev_val = cur_val;
  }

  tail = arcin->tail;
  if (tail->time + arcin->org_cost > latest) {
    arcin = (arc_t *) tail->mark;
    continue;
  }
  ...
}
```

Figure 6.1: Example of the EmonLite code instrumentation: `price_out_impl()` of MCF.

Step 3. For each loop identified in Step 1, the compiler inserts necessary code along with `emonlite_sample()` to read the PMC values for every sampling period.

Figure 6.1 shows how the instrumentation code is inserted into a heavily cache-missing loop in `price_out_impl()` of MCF to collect L2 miss profiles. By varying the profiling interval, PROFILE_PERIOD, the granularity and sensitivity of profiling can be easily adjusted.

6.3.4 Example Usage: Chronology of L2 Miss Events

Figure 6.2 provides the chronology of the L2 miss events for the same loop in Figure 6.1. The graph illustrates the dynamic behavior of the L2 cache at a very fine granularity, *i.e.*, 10 loop iterations per sample (around 2K cycles on average). In the figure, it is obvious that there exist time fragments where the main thread incurs small or even no L2 misses. If a helper thread were launched for such periods with few cache misses, it could potentially degrade the performance during this time interval due to hardware resource contention. This observation has been the motivation to develop certain mechanisms to control helper threads dynamically, which was further discussed in Section 6.2.2.

Figure 6.3 illustrates another example of chronology at a much coarser granularity for the same loop as Figure 6.2. The number of L2 misses is collected for every 100,000 iterations, whereas each profiling period takes about 14M cycles on average. This figure shows the EmonLite profiles for the entire execution of the MCF benchmark, and a phase behavior of L2 misses for the program

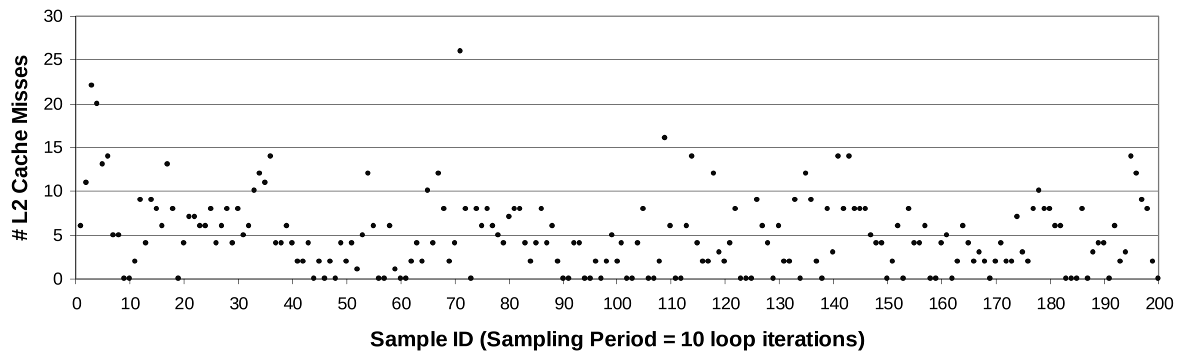


Figure 6.2: Chronology of the L2 miss events at a fine granularity: `price_out_impl()` of MCF

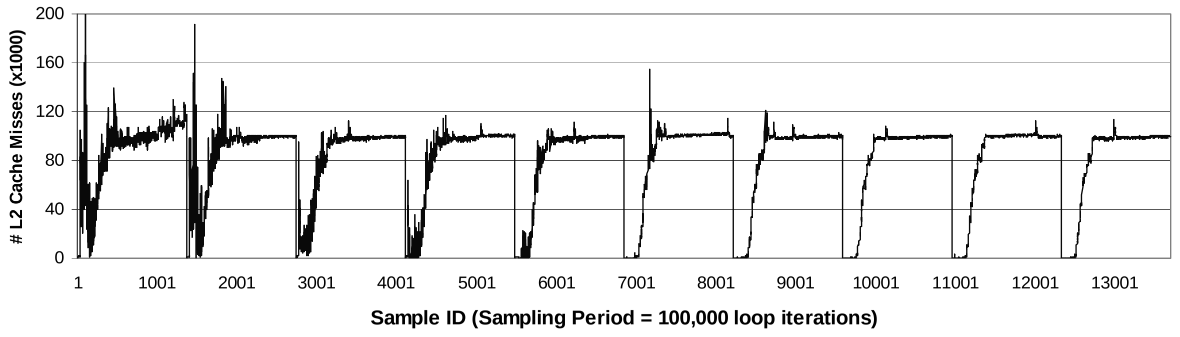


Figure 6.3: Coarse-grain phase behavior of the L2 miss events: `price_out_impl()` of MCF

is clearly evident from the figure.

6.4 Implementation

Having presented several algorithms to generate effective helper threads targeting the Intel Pentium 4 processor with Hyper-Threading Technology, we now discuss implementation issues of the compiler optimization module in the Intel compiler in Section 6.4.1, live-in identification and passing in Section 6.4.2, and code generation in Section 6.4.3.

6.4.1 Prototype Compiler

The compiler module is implemented as a middle-end optimization pass in the Intel compiler and thus it is able to utilize several product-quality analyses such as array dependence analysis and global scalar optimization.[5] These analyses, invoked after the pre-execution optimization pass, help

[5]The Intel compiler has a common intermediate representation, called IL0, for C, C++, and FORTRAN95 [23], and our compiler module residing in the IL0 optimization phase is shared for both IA-32 and Itanium Processor

to further optimize the generated helper thread code. Prior to the helper threading optimization phase, the Intel compiler invokes interprocedural analysis phase so that many of the user-defined procedures are inlined, providing much optimization opportunities across procedure boundaries. The per-load basis cache-miss and cycle-count summary profiles are fed into the Intel compiler in the form of the VTune's `tb5` files [79], and they are correlated to the IL0 statements via source line number and function name to identify the delinquent loads to be covered by helper threads. Finally, the IA-32 backend compiler generates binaries in which helper thread codes are attached.

6.4.2 Live-in Passing

Once a helper thread is formed, the Intel compiler identifies the live-in variables to be passed from the main thread to the helper thread. Through analysis, the variables used in the selected loop are divided into two groups: those that have upwards-exposed reads and those that are privatizable. The variables in the former group are selected as live-ins and will be explicitly passed through global variables that are declared solely for the use of helper threads. Since the helper thread code is inlined into the same routine with the main thread code using multi-entry threading in the Intel compiler, local variables declared for the main thread are also visible to the helper thread. Therefore, we do not insert code to pass the *read-only* variables to the helper thread.

6.4.3 Code Generation

After the compiler analysis phases for pre-execution, the constructed helper threads are attached to the application program as a separate code. In addition, the codes to create, schedule, and invoke the helper threads are inserted as well. For each pre-execution region, we generate two kinds of helper threads that implement the SERIAL scheme and the STRIPMINING scheme shown in Section 6.2.2. In the hyper-threaded processor-based system, it is essential to reduce the thread switching overhead. We adopt a software-based thread recycling mechanism similar to what was

Family (IPF) backend compilers. Hence the compiler optimization module for pre-execution can be applicable for a number of source languages and target platforms. However, we only evaluate our compiler module for benchmarks written in C and generate binaries targeting the IA-32 architecture with Hyper-Threading Technology.

described in Section 4.1.3 for our SUIF-based compilers. The compiler creates only one OS thread using the OS API, `CreateThread()`, at the beginning of the program and recycles the thread to target multiple pre-execution regions.

Chapter 7

Evaluation in a Physical System with the Intel's Hyper-Threaded Processor

We believe it to be greatly valuable to experiment with a proposed research idea in a real physical system in order to verify whether the idea can indeed provide wall-clock speedup, and to uncover useful insights that cannot be learned from simulation-based evaluation. In this chapter, we present experimental results of compiler-based pre-execution on real silicon and discuss our findings from those experiments. The helper threads are generated using the compiler optimization module in the Intel compiler presented in Chapter 6, and the performance of selected workloads with and without helper threads is measured on a machine with an Intel Pentium 4 processor with Hyper-Threading Technology. Section 7.1 presents the experimental methodology and results, and Section 7.2 discusses the findings and insights from our experience with helper threads on real silicon.

7.1 Experimental Evaluation

In this section, we first present the methodology to perform physical experimentation on the Intel's hyper-threaded processor-based system; Section 7.1.1 provides the physical system configuration used for the experiments, Section 7.1.2 introduces the evaluated benchmarks, and Section 7.1.3 defines the baseline configuration so that we can compute the speedup with pre-execution. Then we present the experimental results; Section 7.1.4 evaluates prefetching helper threads using two static thread initiation schemes, *i.e.*, SERIAL and STRIPMINING, Section 7.1.5 estimates the potential of dynamic throttling of helper threads based on the hypothetical DYNAMIC STRIPMINING scheme, and Section 7.1.6 examines dynamic program behavior for various monitoring granularities and emphasizes the need for light-weight thread synchronization mechanisms.

7.1.1 System Configuration

Table 7.1 presents the system configuration used for our experiments. The system contains a single 2.66GHz Intel Pentium 4 processor with Hyper-Threading Technology [30, 47], which supports two

Table 7.1: Physical system configuration.

CPU	2.66GHz Intel Pentium 4 with Hyper-Threading Technology
L1 Trace cache	12K micro-ops, 8-way set associative, 6 micro-ops per line
L1 Data cache	16KB, 4-way set associative, 64B line size, write-through, 2/4-cycle for Int/FP
L2 Unified cache	1MB, 8-way set associative, 64B line size, 7-cycle access latency
DTLB	64 entries, fully associative, map 4K page
Load buffer	48 entries
Store buffer	24 entries
Reorder buffer	128 entries
OS	Windows XP Professional, Service Pack 1

logical processors simultaneously. For memory subsystem, the processor uses a 12K micro-op trace cache and a 16KB data cache at the first level (L1), and a 1MB unified cache at the second level (L2). In addition, the processor implements a hardware stride prefetcher and a sector prefetcher for data prefetching. The reorder buffer has 128 entries; in the MT mode, this structure is hard-partitioned, so each logical processor gets 64 entries. Finally, we use the Windows XP as the OS to resemble the computing environment of usual end users. While we evaluate the OS API for thread synchronization, our compiler module for pre-execution does not depend on a specific OS.

7.1.2 Benchmarks

To decide the benchmarks used for our experiments, we first run the entire SPEC CPU2000 benchmark suite [29] on the VTune performance analyzer and collect the clock cycle and L2 miss profiles. Then those applications that have significant number of cycles attributed to the L2 misses are selected; they are MCF and BZIP2 from the SPEC CINT2000 suite, and ART from the SPEC CFP2000 suite. The reference input sets are used for both profile run and the actual performance measurement. In addition, we also pick MST and EM3D from the Olden benchmark suite [12] for the same reason. The remaining applications in the Olden benchmark suite are not included since they contain structures that perform tree traversal, which our current pre-execution compiler

Table 7.2: Benchmark characteristics.

Suite	Name	Input
SPEC CINT2000	MCF	reference
	BZIP2	reference
SPEC CFP2000	ART	reference
Olden	MST	3000 nodes
	EM3D	40000 nodes

module cannot analyze. Table 7.2 summarizes the benchmarks that are evaluated in our study and their input sets.

Tolerating memory latency has long been tackled by both microarchitecture techniques and advanced compiler optimizations. As an example, the Intel Pentium 4 processor employs a hardware stride prefetcher that monitors the address stream of load instructions and captures stride access patterns. In addition, the production compiler usually performs various optimizations to reduce the number of cache misses such as cache-conscious code or data layout and access optimizations [17]. Our objective is to tackle those cache misses that remain even after these hardware and compiler techniques are applied, and to provide additional speedup using helper threads. The benchmarks are compiled with the best compiler options "-O3 -Qipo -QxW" (however, we do not perform *profile-guided optimization*) in the most recent version of the Intel production compiler. Then the VTune is used to identify the candidate loads for pre-execution. Figure 7.1 shows the percentage of the L2 misses associated with the targeted delinquent loads over the total L2 misses. It also shows the percentage of the exposed memory stall cycles due to the cache misses over the entire execution time. For every benchmark in the figure, we observe fewer than five static loads contribute to a large fraction of the total L2 misses, *i.e.*, 83.5% for the top five delinquent loads on average. The percentage of memory stall cycles indicates the upper bound on the performance improvement that is achievable with perfect data prefetching for the targeted delinquent loads.

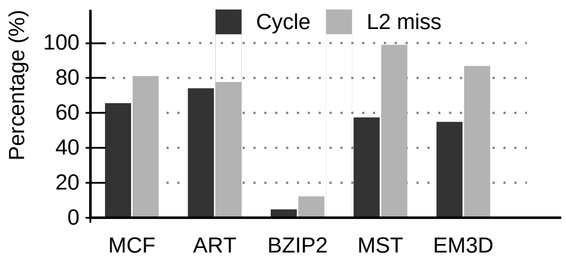


Figure 7.1: VTune profiles: cycles and L2 misses associated with the identified delinquent loads

7.1.3 Baseline

Before presenting experimental results with helper threads, we need to define our *baseline* configuration. In a hyper-threaded processor, the Windows OS periodically reschedules a user thread on different logical processors. This involves the overhead of OS job scheduling and possibly incurs more L1 and L2 misses if multiple physical processors are employed within a single system. On the other hand, in the context of helper threading, a user thread and its helper thread, as two OS threads, can potentially compete with each other to be scheduled on the same *logical* processor. In addition, on a multiprocessor configuration with multiple physical processors, an application thread and its helper thread can also be scheduled to run on different *physical* processors. In this case, without a shared cache among different physical processors, the prefetches from a helper thread will not be beneficial to the application thread. In order to avoid such undesirable situations, the compiler adds a call to the Win32 API, `SetThreadAffinityMask()`, to manage thread affinity at the beginning of the application thread to pin the main thread to a particular logical processor. This is our baseline configuration to evaluate the performance of pre-execution. Similarly, the helper thread, when created, is pinned to the other logical processor within the same physical processor. The speedup of pre-execution is computed by dividing the execution time with helper threads by that of the baseline, whereas we measure the wall-clock time for the entire program execution for both cases.

Figure 7.2 shows the effect of thread pinning on the performance of benchmarks without pre-execution. Each bar represents the execution time without thread pinning, which is normalized to the execution time with thread pinning. Clearly, thread pinning slightly improves the single

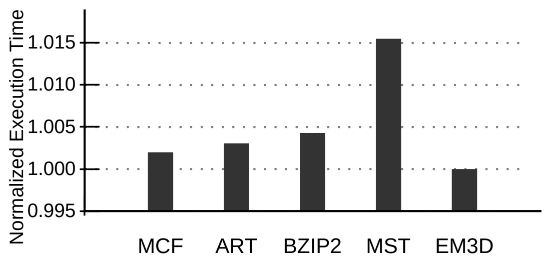


Figure 7.2: Normalized execution time without thread pinning. The execution time without helper threads, but with thread pinning, is our baseline for the actual performance measurements.

thread performance in the hyper-threaded processor. This is because thread pinning eliminates the overhead associated with OS job scheduling. Throughout the rest of this chapter, we use the execution time *with* thread pinning as the baseline reference performance numbers.

7.1.4 Evaluation of Static Thread Initiation Schemes

Statistics for StripMining Scheme

To evaluate the StripMining scheme, the sampling period should be determined a priori. The compiler instruments each targeted loop with the EmonLite library routines to profile the chronology of cycles and L2 misses. The sampling period is adjusted such that each sample takes between 100K and 200K cycles on average. Recall that the Windows API-based thread synchronization mechanisms cost between 10K and 30K cycles, whereas the prototype hardware-based synchronization mechanism takes about 1,500 cycles. Consequently, with the Win32 API, the thread synchronization overhead has a significant impact on performance. Table 7.3 lists the procedure name that contains the targeted loop, the sampling period in loop iterations, and the number of samples over the entire program execution for each selected loop, which in turn denotes the total number of helper thread invocations for the pre-execution region. In each benchmark except MCF, a loop that accounts for the largest fraction of the memory stall time is selected. In MCF, two loops that severely suffer from cache misses are chosen.

Table 7.3: Statistics for the StripMining scheme.

Application	Procedure Name	Sampling Period	Number of Samples
MCF	`refresh_potential`	100	2,422,827
MCF	`price_out_impl`	1,000	1,370,258
ART	`match`	1,000	1,672,740
BZIP2	`sortIt`	1,000	118,201
MST	`BlueRule`	100	44,985
EM3D	`all_compute`	200	20,000

Speedup Results

Figure 7.3 reports the speedup results of the two static schemes, *i.e.*, the Serial and StripMining schemes. For each scheme, we compare the performance of two thread synchronization mechanisms, the heavy-weight Windows API and the light-weight hardware mechanism. The speedup is over the baseline configuration, and it is for the entire program execution, not just for the targeted loop only. For each benchmark, we show speedups in percentage for four different configurations, *i.e.*, "SO," "SH," "MO," and "MH." Each configuration name consists of a two-letter acronym. The first letter denotes the thread initiation scheme that is used to initiate the helper threads; "S" stands for Serial, whereas "M" stands for StripMining. The second letter denotes the thread synchronization mechanism to resume and suspend the helper threads; "O" stands for OS API whereas "H" stands for hardware mechanism.

First, let us examine the performance impact of the thread synchronization cost by comparing the speedup results of the OS API and hardware mechanism in Figure 7.3. For the Serial scheme, *i.e.*, "SO" vs. "SH," the light-weight hardware mechanism provides 1.8% more speedup, on average, than using the OS API. This relatively small difference is primarily because, for the Serial scheme, the helper threads run for the entire iterations of the targeted loops before the next synchronization point at the loop boundary, thus the impact of thread startup cost is much less significant, even considering the cost of the OS API. On the other hand, for the StripMining

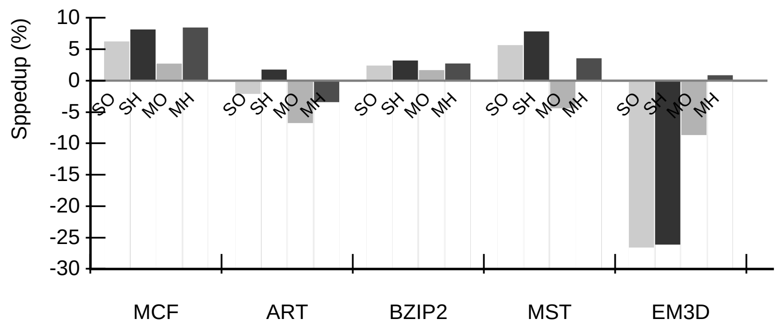


Figure 7.3: Speedup of the static thread initiation schemes. From the left to right bars, we label each bar as "SO" for the Serial scheme with the OS API, "SH" for the Serial scheme with the hardware mechanism, "MO" for the StripMining scheme with the OS API, and "MH" for the StripMining scheme with the hardware mechanism. The speedup is computed by normalizing the execution time of each configuration to that of the baseline.

scheme, the difference in performance impact by the two thread synchronization mechanisms is rather pronounced. Comparing the "MO" and "MH" bars, the hardware mechanism provides an average of 5.5% additional gain over the OS API. Since the helper threads are activated more frequently in the StripMining scheme, the effectiveness is much more sensitive to the thread synchronization cost. The heavy-weight OS API introduces significant overhead on the main thread and potentially causes the helper threads to be activated out of phase, thus resulting in ineffectual pre-execution, which not only runs behind but also takes away critical processor resources from the main thread. This explains the slowdown in "MO" for most benchmarks except for MCF, which suffers from lots of long latency cache misses, and thus running helper threads is still beneficial even with the heavy-weight thread synchronization mechanism.

Comparing the performance of the Serial and StripMining schemes in Figure 7.3, the Serial scheme performs slightly better than the StripMining scheme for the sampling period shown in Table 7.3 and the given thread synchronization cost. When using the OS API for synchronizing helper threads, *i.e.*, "SO" vs. "MO," the Serial scheme outperforms the StripMining scheme for all benchmarks except for EM3D. In EM3D, with the Serial scheme, we observe the helper thread runs away from the main thread due to lack of synchronization in the middle of the loop's execution. Consequently, the prefetched cache blocks from the run-away helper thread are evicted before use, and worse yet, evict useful cache blocks in the process, thus degrading the performance of the main thread significantly. However, using the StripMining scheme effectively prevents such cache thrashing since each sampling period effectively works as a synchronization point, providing better performance for EM3D. For the other benchmarks, since the StripMining scheme invokes helper threads more frequently, the heavy-weight overhead of calling the OS API limits the performance improvement of the main thread. On the other hand, if the light-weight hardware mechanism is employed, *i.e.*, "SH" vs. "MH," the performance with the StripMining scheme is comparable to that of the Serial scheme. Figure 7.3 shows there is little difference between the two schemes in MCF and BZIP2. In ART, however, since the targeted loop consists of only 12 instructions, the instrumentation code to track sampling period and invoke the helper threads accounts for a relatively large portion of the loop work, resulting in performance degradation with the StripMining scheme. In MST, the StripMining scheme performs worse due to both the thread synchronization cost and code instrumentation overhead. Although the hardware mechanism provides an order of magnitude reduction in overhead as compared to synchronization via the OS API, even at 1,500 cycles, the hardware synchronization still takes more than twice as long as the latency to serve a cache miss from main memory. While the benefit of the StripMining scheme is not noticeable in the figure, we expect it to be more effective with even lower thread synchronization cost.

A Case Study: Estimating Upper-Bound Performance with Helper Threads

The performance with pre-execution is determined by how fast a helper thread can run, not by how fast the main thread runs. In other words, as long as the helper thread triggers cache misses quickly and runs far ahead, the main thread simply catches up with the helper thread while enjoying cache hits. Thus, for the two loops in MCF that are pre-executed, we measure the time to execute the cache-miss kernel in several situations to estimate the upper-bound performance with helper threads. We introduce a set of experiments, from E1 to E5, where we use the Serial scheme and the hardware mechanism for thread synchronization. Note, depending on which thread initiation scheme is used and how helper threads are optimized, the experimental results and the estimated upper-bound performance with helper threads may vary. The description of each experiment is shown below; "E" stands for Experiment.

E1: We first measure how long it takes to execute the original loop code without helper threads (just the targeted loop only, not the entire program). We place the EmonLite library routines before and after the targeted loop and record the cycle counts for each loop instance. At the end, we sum up all cycle counts and get the total cycles spent on the targeted loop over the entire execution. This becomes the baseline configuration for the experiments E1 through E5.

E2: We measure how fast a helper thread executes the cache-miss kernel when all the hardware resources are given to the helper thread. First, we perform slicing on the original loop and remove noncritical code that does not affect the execution of the delinquent loads. The remaining code is the cache-miss kernel and looks exactly the same as the helper thread with the Serial scheme. Then we record the cycle counts for each loop instance and compute the total cycle counts as is done in E1. In theory, E2's cycle count is the best achievable performance by a helper thread for the Serial scheme in the absence of resource contention. Comparing the E1 and E2 bars, we can also estimate the criticality of the cache-miss kernel. In other words, if the E1 and E2 bars are close, it means the cache-miss kernel is in the critical path of the targeted loop. Thus, unless the helper thread is optimized in a different way, using the Serial scheme may not provide any speedup.

E3: This experiment is the same as E2 except that we run another thread alongside the helper thread which does nothing but execute the `pause` instructions. The purpose of this experiment is to see the impact of hardware resource partitioning on the speed of the helper thread.[1] The total cycle count is the shortest time of the helper thread in the MT mode.

[1]Unfortunately, we cannot leave the hyper-threaded processor in the MT mode while one of the logical processors is completely idle, which is exactly what we want in order to examine the impact of resource partitioning. Hence the pausing thread is the best we can do.

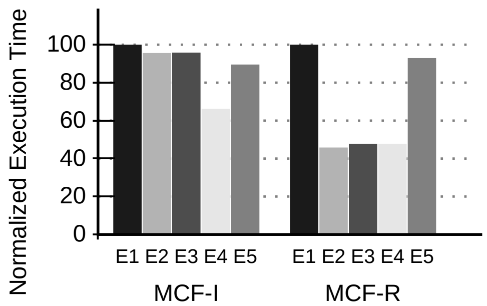


Figure 7.4: Speed of the cache-miss kernels for various situations. For these experiments, we use the SERIAL scheme with the hardware thread synchronization mechanism.

If the cycle count is much increased when going from E2 to E3, this implies the cache-miss kernel is bounded by processor resources that are hard-partitioned in the MT mode such as the reorder buffer.

E4: If the E3 cycle count is larger than that of E1, we cannot expect any speedup with helper threads using the SERIAL scheme. Therefore, we need to employ another technique to optimize and transform the helper thread, thereby achieving speedup with helper threads. As an optimization technique for E4, we examine whether the delinquent load shows stride access patterns, and if it does, we implement stride prefetching in the helper threads. Then we measure the total cycles of the modified cache-miss kernel still in the MT mode as in E3. Note, for E2 through E4, to ensure the correct execution of the program, we remove all stores in the cache-miss kernel to eliminate side effects and execute the original loop after the EMONLITE-based cycle count measurement.

E5: Lastly, we run the optimized cache-miss kernel from E4 on a helper thread while the main thread executes the original loop code in parallel. The total cycles are the actual performance that we achieve with helper threads. By comparing the performance of E4 and E5, we can see the impact of interference between the main thread and the helper thread. Possible reasons for the interference are hardware resource sharing and cache thrashing.

Figure 7.4 shows the execution time of each experiment, which is normalized to the execution time of E1. This figure provides valuable insights into how helper threads can achieve speedup in the hyper-threaded processor.

The first loop, labeled "MCF-I," is from `price_out_impl()`, which accounts for 29.2% of the total cycles in MCF. This loop is pointer-chasing and the cache miss kernel is the linked-list traversal. After slicing, the code to perform value computation is removed. First, comparing the E1 and E2 bars, one can see that the linked-list traversal is the critical path of the loop, and the execution time of the value computation is overlapped behind the linked-list traversal. Comparing the E2 and E3 bars, the execution time is not much increased even though some of the hardware resources are hard-partitioned. Since most instructions in this loop are serialized along the linked list, they do not require many reorder buffer entries and the cycles are not significantly affected by the reduced reorder buffer size in the MT mode. Comparing the E1 and E3 bars, one can notice that helper threads may not provide significant gain if the helper thread just traverses the same linked list as the main thread does. We observe that the delinquent load in this loop shows fixed stride access pattern and implement a software-based stride prefetcher in the helper thread.[2] The speed of the helper thread is significantly improved as shown in the E4 bar. Comparing the E4 and E5 bars, one can see the effect of interference between the main thread and the helper thread, and we finally achieve an 11.7% speedup for this loop.

The second loop, labeled "MCF-R," is from `refresh_potential()`, which accounts for 52.7% of the total cycles in MCF. This loop is also pointer-chasing and the extracted cache-miss kernel determines the control flow of the loop. Comparing the E1 and E2 bars, the cache-miss kernel accounts for about half the original cycles and its speed is not affected by resource partitioning as shown in E3 for the similar reason in the "MCF-I" case. We do not apply any technique to optimize the cache-miss kernel, so the E3 and E4 bars are the same. Finally, when we perform pre-execution, the performance is significantly affected by interference between the two threads, and we achieve a 7.6% speedup for this loop.

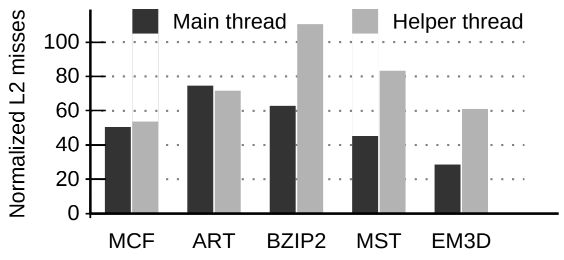


Figure 7.5: Cache-miss coverage using the SERIAL scheme with the hardware thread synchronization mechanism.

Dynamic Behavior of Helper Threads

In order to shed further insights into the impact of helper threads on program execution, we discuss the dynamic behavior of the helper threads by examining both the cache-miss coverage and cycle-count improvement.

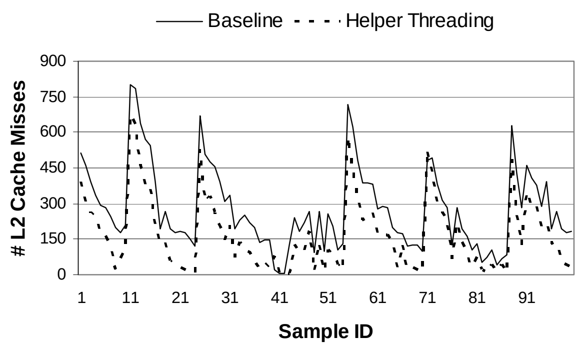
Figure 7.5 illustrates the L2 miss coverage based on the VTune profiles[3] for the SERIAL scheme with the hardware thread synchronization mechanism (configuration "SH" from Figure 7.3). In Figure 7.5, we show two bars for each benchmark; the bar at the left, labeled "Main thread," is the L2 miss count incurred by the targeted delinquent loads in the main thread, whereas the bar at the right, labeled "Helper thread," is the L2 miss count incurred by the helper thread to prefetch for those delinquent loads. Note, only cache misses due to delinquent loads within the targeted pre-execution regions are counted, and cache misses incurred outside the pre-execution regions are not shown in the figure. Each bar is normalized to the L2 misses of the baseline configuration for the same set of the delinquent loads. The data clearly indicate that helper threads significantly reduce cache misses in the main thread, ranging from 25.3% in ART to 60.4% in EM3D. In EM3D, helper threads eliminate a large portion of the L2 misses for the targeted delinquent loads. However, they also increase the number of cache misses for the non-targeted loads, which is not shown in Figure 7.5, thereby degrading the overall performance of EM3D (see Figure 7.3).

[2]We observe that this load is not perfectly prefetched by the hardware prefetcher implemented in the processor, thus leaving some room for performance improvement with pre-execution.

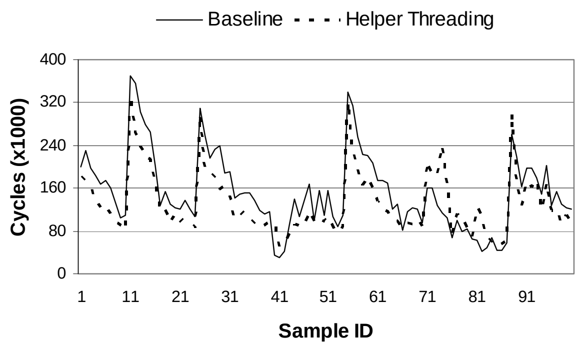
[3]Current version of the VTune performance analyzer is enabled for Hyper-Threading Technology [79], and thus it can collect profile information while threads are running on the two logical processors simultaneously.

On the other hand, this figure also reveals some inefficiency of the static thread initiation schemes. First, in all benchmarks except for BZIP2, the percentage of the cache misses covered by the helper threads, “Helper thread” bar, is not close to 100%. This indicates helper threads sometimes run behind the main thread and do not help to reduce neither the cache misses nor the cycles of the main thread. Since unhelpful helper threads may degrade the performance of the main thread due to the resource contention problem discussed in Section 6.1.1, helper threads should not be activated for certain period. Second, in ART, BZIP2, MST, and EM3D, the sum of the two bars exceeds 100%. This indicates that for certain time phases, helper threads run too far ahead of the main thread and incurs cache thrashing, thereby making already prefetched cache blocks still miss in the cache. During those time phases when the helper threads are not effective, certain dynamic throttling mechanism can be introduced to either suspend an on-going helper thread or prevent activation of the next helper thread instance in order to avoid the performance degradation.

To investigate the impact of helper threads at a much finer granularity, Figure 7.6 shows the EmonLite-based chronology of the L2 miss events and the cycle events in BZIP2. The profiles are collected over 100 samples using the StripMining scheme with the hardware thread synchronization mechanism. Each figure depicts two sets of data, one without pre-execution (solid line), and the other with pre-execution (dotted line). Comparing the patterns in Figures 7.6a and 7.6b, one can see there exists strong correlation between the L2 miss event and the cycle event, implying that those targeted loads are likely critical. However, when helper threads are applied, there are some sample phases when L2 miss reductions do not convert to similar reductions in cycle counts. For instance, between sample IDs 71 and 88, even though the number of L2 misses is reduced with helper threads, the cycle counts actually increase. It would be helpful if we could detect those time phases when the application performance is degraded with pre-execution so that helper threads do not get activated for such time phases. This observation leads us to consider certain runtime mechanisms to dynamically throttle helper threads, a topic to be discussed in the next section.



(a) L2 cache miss event



(b) Cycle event

Figure 7.6: Dynamic behavior of the performance events in BZIP2 with and without helper threads. a) Chronology of the L2 miss events. b) Chronology of the cycle events.

Table 7.4: Percentage SD statistics. Profiles are collected without launching helper threads.

Application	% SD(cycle)	% SD(L2 miss)
MCF (refresh)	44.80%	39.45%
MCF (implicit)	44.50%	50.41%
ART	17.14%	3.79%
BZIP2	61.91%	96.83%
MST	30.59%	30.86%
EM3D	46.05%	44.72%

7.1.5 Evaluation of Dynamic Throttling of Helper Threads

In this section, we explore the potential of dynamic throttling of helper threads based on the Dynamic StripMining scheme by assuming a hypothetical perfect throttling mechanism.

Quantifying Dynamic Behavior

First, let us introduce how we quantify the dynamic behavior of a program. Using the same sampling periods shown in Table 7.3, we collect the cycle and L2 miss profiles for each sample of the targeted loops. Profile information is acquired using the EmonLite code instrumentation without running helper threads. Then we compute the percentage standard deviation (SD) of the cycles and the L2 misses among all the samples as shown in Eq. 7.1, where PMC(i) is the PMC value for the i-th sample, A is the average PMC value per sample, and N is the total number of samples.

$$SD(\%) = \sqrt{\frac{\sum_i (PMC(i) - A)^2}{N}} * 100/A \tag{7.1}$$

Table 7.4 reports the percentage SD values of cycles and L2 misses for each targeted loop. A large SD value implies the corresponding performance event is more time variant. Again, we can observe some correlation between the cycle event and L2 miss event standard deviations. Or rather, if one performance monitoring event is dynamically variant, so is the other one. This is because the delinquent loads in our targeted loops are usually on the critical path, and thus the cache miss behavior directly affects the cycle-count behavior.

Performance Potential with Perfect Throttling

To estimate the performance impact of dynamic throttling of helper threads, we postprocess the cycle and L2 miss profiles acquired by actually running benchmarks with and without helper threads. For this purpose, it is essential to gauge when a helper thread improves or degrades the performance of the main thread. This can be done by comparing the cycle profiles with and without helper threads for each sample. For a limit study, an ideal scenario is to activate the helper thread only when it is beneficial and deactivate it when it degrades performance. Thus, as the first step, we collect the EmonLite profiles using the StripMining scheme for every sampling period with and without helper threads. To estimate the execution time with perfect throttling of helper threads, we do the following. Using the profiles without helper threads as the baseline configuration, we record the cycle counts *with helper threads* for those samples that provide performance improvement. For the remaining samples that exhibit performance degradation, the increased cycle counts due to detrimental effect of helper threads are discarded and the baseline cycle counts are recorded instead. The sum of the recorded cycles projects the execution time with perfect throttling, where a perfect throttling algorithm would activate a helper thread only for those samples with speedup.

Figure 7.7 presents the total cycles for the targeted loops only, not the entire program, for 4 different configurations; the StripMining scheme with the OS API (SO), the Dynamic StripMining scheme with the OS API (DO), the StripMining scheme with the HW mechanism (SH), and the Dynamic StripMining scheme with the HW mechanism (DH), where each bar is normalized to the cycles of the baseline. It is apparent that perfect throttling would provide nontrivial speedups beyond the static scheme's performance. The figure shows, on average, 4.8% more gain with the OS API and 1.2% more gain with the hardware mechanism can be achieved. The larger additional gain with the OS API is partly due to the nondeterministic nature of the OS

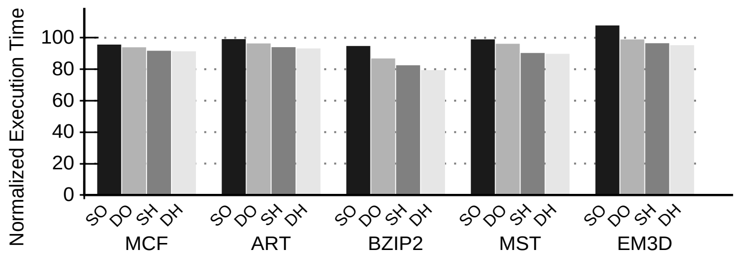


Figure 7.7: Performance comparison between the StripMining scheme and the Dynamic StripMining scheme. From left to right bars, each bar is labeled to present the thread initiation scheme and the thread synchronization mechanism: "SO" for the StripMining scheme with the OS API, "DO" for the Dynamic StripMining scheme with the OS API, "SH" for the StripMining scheme with the hardware mechanism, and "DH" for the Dynamic StripMining scheme with the hardware mechanism.

synchronization mechanism. Interestingly, there exists correlation between the SD values in Table 7.4 and the impact of dynamic throttling in Figure 7.7. For instance, BZIP2 has the largest SD value for cycle counts showing the most dynamic behavior among the benchmarks, and the amount of execution time difference without and with dynamic throttling is also the largest. This implies applications with more dynamic behavior can benefit more from dynamic throttling mechanisms. With a perfect throttling algorithm and light-weight hardware synchronization mechanism, helper threads can provide as much as 20.6% wall-clock speedup for the targeted loop in BZIP2. This performance potential of dynamic throttling serves to motivate future efforts to better optimize the helper threads and lower the thread synchronization cost.

7.1.6 Sensitivity of Program's Dynamic Behavior

In our experimental settings, the granularity of the sampling period for the StripMining scheme is limited by the cost of the thread synchronization mechanisms. Even in the worst case, the length of a sampling period cannot be shorter than the time required for resuming and suspending the helper threads. Otherwise, helper threads will be out of phase and cannot improve the main thread's performance. Consequently, the dynamic behavior of a program is exploited at a rather coarse granularity with the current thread synchronization cost. Note, even the 1,500 cycle latency of the hardware synchronization mechanism is more than twice the L2 miss latency. In this section,

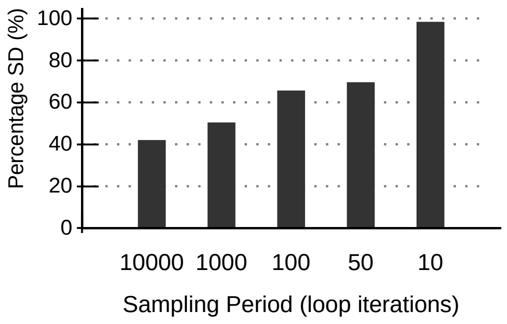


Figure 7.8: Percentage SD of L2 misses for various sampling periods.

we demonstrate the dynamic program behavior for various sampling periods, from very coarse granularity to extremely fine granularity.[4] From this experiment, we show that the time variance in the dynamic cache miss behavior becomes more pronounced as the resolution of sampling of the program execution increases, which indicates there exists much more room for exploring dynamic throttling at a finer granularity, potentially providing larger performance improvement. In turn, this motivates further hardware optimizations to reduce the thread synchronization cost.

In Figure 7.8, we vary the sampling period from 10000, to 1000, 100, 50, and 10 loop iterations for the targeted loop in `price_out_impl()` of MCF, and profile the L2 misses for each sample using the EmonLite code instrumentation. The figure presents the percentage SD for the different sampling periods. The SD values show a steady increase from 10000, to 1000, 100, and 50 iterations. Once the sampling period reaches 10 loop iterations, where each sample takes around 2K cycles on average, the cache miss behavior is highly dynamic, implying there is significant variation in L2 miss counts between different samples. Figure 7.8 shows that such dynamic cache behavior can only be captured at a very high sampling resolution.

7.2 Key Observations

This section summarizes the key insights we learned from our experiments with prefetching helper threads on real silicon. We first discuss several impediments to achieve speedup in a physical system, then present some essential mechanisms to overcome the constraints and further increase

[4]Note, we can still monitor the performance events at such fine granularity thanks to the light-weight profiling overhead of the EmonLite.

the performance gain.

7.2.1 Impediments to Speedup

From our experiments, we observe several issues that are unique to a physical system limit the performance improvement with prefetching helper threads. To achieve significant speedup on a real machine, it is crucial to address those impediments to speedup that are correlated with each other.

First, hardware resource contention in the hyper-threaded processor imposes intricate trade-offs regarding when to launch a helper thread, for how long to execute the helper thread, and how frequently to reactivate the helper thread. To perform effective pre-execution in a physical system based on hyper-threaded processors, judicious invocation of helper threads must be ensured to avoid potential performance degradation of the target workload due to resource contention between the main thread and the helper thread.

Second, program execution often exhibits dynamically varying behavior of microarchitectural events such as cache misses and hardware resource distribution between the main thread and the helper thread. As illustrated in Figure 7.6, there exist certain time phases when helper threads may not be helpful due to various reasons such as lack of cache misses to tolerate or contentions for MSHRs, cache ports, and bus bandwidth. We observe static scheduling of helper threads, solely based on compile-time analyses, has some limitations due to such dynamic events at runtime. Therefore, it is very important for the helper threads to adapt to the dynamic program behavior and computing environment at runtime and throttle the invocation of helper threads to enable more effective prefetching.

Third, to achieve even more speedup with helper threads, monitoring and adapting to the dynamic behaviors of program execution at a very fine granularity are crucial. This requires frequent communication and synchronization between the main thread and the helper thread. To support such fine-grain handshaking effectively, having very light-weight thread synchronization and switching mechanisms is critical.

7.2.2 Essential Mechanisms

To overcome those impediments presented in Section 7.2.1, certain software and hardware mechanisms are required. We need to develop better compiler algorithms to construct more efficient helper threads and schedule them judiciously to ensure timely activation and deactivation. In addition, the compiler must further optimize the helper threads to reduce resource contentions, *e.g.*, by exploiting occasional stride-prefetch pattern, as a form of strength reduction in order to accelerate helper thread execution and thus potentially minimize resource occupancy by the helper threads. Furthermore, it is crucial to employ runtime mechanisms to capture the dynamic program behavior and throttle the helper threads, thereby filtering out helper thread activations which lead to wasteful resource contention and unnecessary prefetches. Since the dynamic throttling mechanisms would be more effective with very fine-grain thread synchronization, light-weight thread synchronization support in hardware is also essential. If provided with such compile-time and runtime support, pre-execution can be a highly effective technique to deal with the ever-increasing memory latency problem in those workloads that have large working sets and thus suffer from significant cache misses, especially those misses that defy conventional prediction-based prefetching techniques or software prefetches integrated into the main thread code.

Chapter 8

Evaluation of Compiler-Based Pre-Execution in a Multiprogramming Environment

Thus far, we evaluate compiler-based pre-execution for single-threaded programs only. However, modern computing environments typically include multiprogrammed or multithreaded workloads. For instance, multiple different applications that are independent of each other, or multiple queries for a database executing as different processes can run simultaneously in an SMT processor. Furthermore, in parallel processing, a single workload is parallelized and runs as a multithreaded workload. Performing pre-execution in such multithreading environments exposes some intriguing tradeoffs regarding how to distribute the available hardware resources to computation threads, formerly called the main thread in the single-threaded workload case, and helper threads. When the main threads and the helper threads effectively share the hardware resources in the SMT processor, performing pre-execution can help to boost the overall processor throughput. On the other hand, running helper threads possibly prevents the main threads from acquiring enough hardware resources, perhaps degrading the throughput of the system. In this chapter, we evaluate compiler-based pre-execution in the context of multiprogramming in order to better understand the combination of resource management via thread scheduling and compiler-based pre-execution in SMT processors.

8.1 Multiprogramming Simulation Methodology

For this experiment, we generate helper thread code using our most aggressive SUIF-based compiler framework, known as "Compiler A" in Table 4.1. In addition, we slightly change the configuration of our SMT processor simulator so that it can support a maximum of 5 hardware contexts simultaneously; in our experiment, we assume a hypothetical computing situation in which 2 independent application threads, *i.e.*, main threads, are running and each application can invoke up to 3 helper threads for prefetching. To accommodate the requests for helper threads from the 2 main threads, a total of 8 hardware contexts would be necessary (2 for the main threads and 6 for the helper threads). However, rather than providing enough contexts to support the maximum requirement, we assume the contexts for helper threads are dynamically shared between the 2 main threads. We believe such a design point is realistic since it reduces the design complexity of the SMT processor, and it helps to better utilize its available hardware contexts. Except for the number of hardware contexts, the same processor configuration from Table 5.1 is used throughout all multiprogramming experiments.

Since only a limited number of contexts are available for pre-execution, the main threads must arbitrate for the hardware contexts before initiating pre-execution. To support arbitration, we implement a software queue to keep track of the idle hardware contexts available for helper threads. Before entering a pre-execution region, each main thread must acquire a lock to gain exclusive access to the idle context queue, and reserve the necessary number of hardware contexts depending on the thread initiation scheme. Recall that the Serial scheme attempts to reserve 1 context, while the DoAll and DoAcross schemes attempt to reserve 3 contexts. Due to lack of sufficient hardware contexts for helper threads, reservation requests may fail to acquire the requisite number of contexts. To handle such cases, we modify the helper thread code generated by our compiler to enable the parallel thread initiation schemes, which require multiple contexts, to accommodate a variable number of helper threads. To avoid the situation where a main thread keeps injecting useless instructions into the processor pipeline waiting for the release of the idle context queue lock, we use a hardware semaphore for the idle context queue lock while the helper threads still use software semaphores for communication and synchronization.

The degree of contention for hardware contexts, an important determiner of multiprogramming performance, depends on how actively individual programs employ helper threads during execution. To quantify the helper thread's activity, we introduce a new metric, called the Threading Duty Factor (TDF). An application's TDF is defined as the percentage of time the helper threads remain active when the application runs on a dedicated 4-context SMT machine in which 1 context is used to run the main thread and the other 3 contexts are used for the helper threads. A TDF value indicates the occupancy across all 3 contexts for the helper threads. For example, if a single helper thread employing the Serial scheme is always active throughout the entire execution

Table 8.1: Benchmarks for the multiprogramming experiments. The TDF values for each benchmark are listed in the third row, and the helper thread initiation schemes employed in each benchmark are listed in the last row. Benchmark groupings are indicated in the first row: Group 1 benchmarks have high-TDF values, Group 2 benchmarks have medium-TDF values, and Group 3 benchmarks have low-TDF values.

Group 1			Group 2	Group 3		
em3d	mst	181.mcf	179.art	175.vpr	256.bzip2	188.ammp
99.98%	99.64%	86.18%	63.14%	23.12%	22.22%	18.14%
DA	DX	DX	DA & SE	DA & SE	DA & SE	SE

of a program, the TDF value is 33.33%, since 2 hardware contexts are still available for helper threads. Eq. 8.1 shows how to compute the TDF value for a program:

$$TDF(\%) = \frac{\sum_{i=1}^{3} \ Execution\ time\ of\ helper\ thread\ i}{Total\ simulation\ time} * \frac{1}{3} * 100 \tag{8.1}$$

Table 8.1 reports the TDF values and thread initiation schemes for 7 benchmarks used in our experiments. We divide them into three groups according to their TDF values: Group 1 benchmarks have high-TDF values (EM3D, MST, MCF), Group 2 benchmarks have medium-TDF values (ART), and Group 3 benchmarks have low-TDF values (VPR, BZIP2, AMMP).

8.2 Evaluation Results

Figure 8.1 reports the multiprogramming results. We study 10 multiprogrammed workloads, each consisting of two applications selected from Table 8.1. For each workload, we perform three experiments. All experiments run for a fixed number of cycles, called the *scheduling interval*, which is set to 100M cycles in this study. The first two experiments, labeled “Baseline Simul” and “Pre-exec Simul,” run the two applications in a multiprogrammed workload simultaneously without and with pre-execution, respectively. In these experiments, both applications are active during the entire scheduling interval. For the experiment labeled “Pre-exec Time,” we run the two applications with pre-execution in a time-sliced manner, thus each application is active for exactly

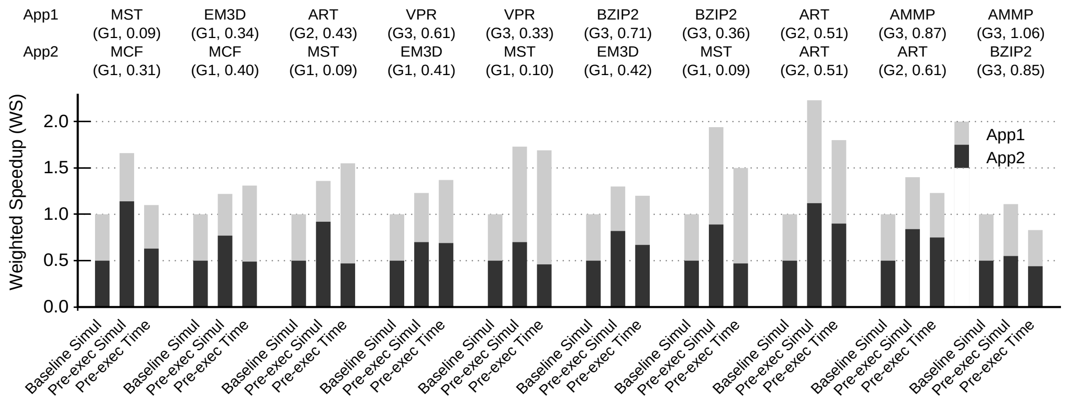


Figure 8.1: Weighted speedup of the multiprogrammed workloads. "Baseline Simul" denotes simultaneous execution without pre-execution. "Pre-exec Simul" and "Pre-exec Time" denote pre-execution with simultaneous execution and time sharing, respectively. The TDF group that each application belongs to and the baseline IPC for the application are shown in parentheses.

half the scheduling interval. Note, for "Pre-exec Simul," the 2 main threads compete with each other to acquire hardware contexts for running helper threads, whereas "Pre-exec Time" does not encounter the same problem since only one main thread runs at any given time, utilizing whatever processor resources it needs to perform pre-execution. Figure 8.1 shows the weighted speedup [67] achieved across the entire scheduling interval for each experiment and workload, computed as follows:

$$Weighted\ Speedup = \frac{1}{2}\left[\frac{IPC_{App1}}{IPC_{BaseApp1}} + \frac{IPC_{App2}}{IPC_{BaseApp2}}\right] \tag{8.2}$$

where $IPC_{BaseApp1}$ and $IPC_{BaseApp2}$ are the IPCs achieved by each application in the "Baseline Simul" experiment. Individual bars are broken down into two components to show each application's contribution to the overall weighted speedup.

Figure 8.1 provides several valuable insights. First, comparing the "Pre-exec Simul" bars against the "Baseline Simul" bars, we see pre-execution improves the weighted speedup for all 10 multiprogrammed workloads. On average, the workloads receive a 45.0% boost in the weighted speedup with pre-execution compared to the baseline. This demonstrates pre-execution has a positive impact on the performance of multiple simultaneously executing applications, even when applications must share a limited number of hardware contexts for helper threads and those launched helper threads steal some hardware resources from the computation threads.

Although performing pre-execution performs better than no pre-execution, we also want to study whether combining pre-execution with simultaneous execution is the best choice. Simultaneous execution in an SMT processor tends to boost throughput by allowing multiple applications to more effectively exploit processor resources. However, it also tends to reduce the pre-execution effectiveness because simultaneously executing applications must compete for the hardware contexts, so the application that could not acquire enough hardware contexts for helper threads often loses opportunities to benefit from helper threads' prefetching. We quantify this tradeoff by comparing the "Pre-exec Simul" and "Pre-exec Time" bars in Figure 8.1. Workloads in the "Pre-exec Simul" experiments exploit pipeline sharing via simultaneous execution of the applications, while workloads in the "Pre-exec Time" experiments exploit contentionless access to the hardware contexts for helper threads since the applications execute one at a time.

Figure 8.1 shows the tradeoff between pre-execution and simultaneous execution depends on the workload and to which TDF group its applications belong. When both applications in a workload have either medium- or low-TDF values, as in the ART-ART, AMMP-ART, and AMMP-BZIP2 workloads, then "Pre-exec Simul" always performs better than "Pre-exec Time." For these workloads, contention for the hardware contexts is low in the "Pre-exec Simul" experiments due to the modest-TDF values of individual applications. Consequently, applications exploit pipeline sharing while running simultaneously without paying a performance penalty for context contention, resulting in higher throughput compared to time slicing. In contrast, when one of the applications in the workload has a high-TDF value, pre-execution and simultaneous execution do not always combine symbiotically. High-TDF applications tend to monopolize the hardware contexts, limiting the other application's ability to perform pre-execution. Time-slicing can relieve this contention, but at the expense of sacrificing simultaneous execution. If boosting the pre-execution performance of the lower-TDF application outweighs the benefit of pipeline sharing, then "Pre-exec Time" outperforms "Pre-exec Simul." This happens in the EM3D-MCF, ART-MST, and VPR-

EM3D workloads. However, if the benefit of pipeline sharing outweighs boosting the pre-execution performance of the lower-TDF application, “Pre-exec Simul” outperforms “Pre-exec Time.” This happens in the MST-MCF, VPR-MST, BZIP2-EM3D, and BZIP2-MST workloads.

Across all 10 multiprogrammed workloads, “Pre-exec Simul” outperforms “Pre-exec Time” by 9.9% on average. Hence for our workload mixes, simultaneous execution is generally more profitable than time slicing. We also observe “Pre-exec Time” outperforms “Baseline Simul” in all workloads except one, providing a boost in the weighted speedup of 29.7% on average. This reinforces our earlier claim that pre-execution is better than no pre-execution for the workloads in Figure 8.1, even when computation threads do not exploit simultaneous execution.

Chapter 9

Related Work

Now, we discuss some of the related works and show how our research can be placed among these previous proposals. This chapter consists of the following three topics, which have been the basis for the advent of pre-execution. First, pre-execution is a novel form of multithreading technique that utilizes idle hardware contexts in single-chip multithreading processors, and thus Section 9.1 covers several multithreading-related proposals. Second, we use pre-execution, which is a general latency-tolerance technique, for data prefetching to eliminate memory stalls in a program's execution. Hence Section 9.2 examines some previous works on data prefetching by categorizing them into the prediction-based and execution-based techniques, and show how prefetching has evolved to the thread-based technique like pre-execution. Finally, Section 9.3 introduces different kinds of pre-execution techniques that are grouped based on how to construct helper threads, and it compares compiler-based pre-execution with other pre-execution techniques.

9.1 Multithreading

Multithreading has been studied for decades in both the academic research community and industry. There have been a plethora of proposals on large multiprocessor systems to enable conventional multithreading techniques such as multiprogramming and parallel processing [3, 39, 65]. In this thesis, however, we focus on multithreading techniques that make use of single-chip multithreading processors, and we examine some of the recent proposals on multithreading. In this section, we discuss speculative multithreading, which is a new type of parallel processing with speculation hardware support, and helper threading that helps the execution of the main thread indirectly by exploiting spare hardware contexts. Both multithreading techniques try to improve single program performance. Then we introduce two popular single-chip multithreading architectures, *i.e.*, SMT and CMP, and examine recent industry trends toward such processors.

9.1.1 Speculative Multithreading

Speculative multithreading, also known as Thread-Level Speculation (TLS), is one popular way to use processors that have a multithreading capability. In contrast to pre-execution where computed results are usually not integrated into the processor's shared state, all threads in TLS try to perform useful computations to contribute to the overall processor throughput. Many speculative multithreading techniques are performed on a CMP model, which is modified to handle the memory speculation effectively. This section introduces two important speculative multithreading techniques.

Thread-Level Data Speculation (TLDS) [72] introduces the hardware and compiler support for constructing and managing multiple speculative threads extracted from a single program. Although the CMP or SMT processors have capability of running multiple threads on a single chip, conventional compilers have not been so successful to parallelize nonscientific codes because it should be guaranteed that no dependence violations occur by parallelization. TLDS proposes the hardware mechanisms to detect a dependence violation between store and load instructions, and to roll back the faulting thread so that it can be restarted. In TLDS, dependence violations do not disrupt the execution of a program due to such speculation hardware support. As long as the dependence violations occur only infrequently, TLDS improves the performance of a program by running multiple threads in parallel.

The idea of Multiscalar Processors [68] is one of the early proposals regarding speculative multithreading. In the multiscalar paradigm, a compiler constructs the Control Flow Graph (CFG) and partitions the CFG into blocks of instructions, called *tasks*. These tasks are assigned to the multiple processing units in the multiscalar processor, arranged in a circular queue. The head of the queue is nonspeculative and executes the earliest task in the queue, and the following processing units execute tasks in a sequential program order. In a multiscalar processor, a special hardware structure, called the Address Resolution Buffer (ARB) [27], is used to hold the speculative memory operations, detect dependence violations of the memory references, and handle the recovery process if necessary.

9.1.2 Helper Threading

Helper threading, also known as *assisted execution* [22], is another type of multithreading where one or more helper threads run alongside the useful computation threads and *indirectly* help their executions. We roughly group various helper threading techniques into four categories, *i.e.*, latency tolerance, exception handling, fault tolerance, and implementation of hardware structures in software, and discuss them below.

Latency tolerance. One of the most popular applications of helper threading is tolerating latency in a program. This kind of helper thread is called the *runahead thread* [7, 24, 73] since the helper thread runs in front of the main computation and resolves the long-latency delinquent events early, thereby hiding the latency behind useful computations. Runahead threads can be used for many purposes such as data prefetching [19, 36, 35, 37, 41, 42, 63, 81, 84], instruction prefetching [1], branch outcome precomputation [14, 15, 25, 63, 84], I/O prefetching [13], and virtual function-call target prediction [61]. In runahead threading, a very crucial issue is how to expedite the helper threads so that they always run in front of the main computation. This thesis contributes to the development of such optimization algorithms to produce more effective helper threads, especially for optimization and transformation of helper thread codes at compile time.

Exception handling. Common hardware exceptions, when implemented by trapping, unnecessarily serialize the execution of a program. Observing the performance degradation due to such serialization, Zilles *et. al.* [86] propose to run the exception handler code on a separate thread, *i.e.*, helper thread, and allow the main thread to keep fetching instructions that are control and data independent of the faulting instruction. In addition to eliminating serialization and overlapping two kinds of threads, the branches in the post exception code are resolved earlier and the memory latency is also tolerated by triggering potential cache misses early. The performance improvement depends on the frequency of exceptions during a program execution, and the amount of independent instructions whose execution can be overlapped with the execution of the exception handler code. This idea is unique in that it decouples the computation code and exception handling code, which is not part of the original program code, and improves the main thread's performance via running a helper thread simultaneously.

Fault tolerance. As transistors scale to deep submicron level, transient faults occur more frequently. Reinhardt and Mukherjee [58] demonstrate the Simultaneous and Redundantly Threaded (SRT) processor where an identical copy of the same program runs simultaneously as an independent thread on a derivative of an SMT processor, but it runs behind the original program to check any occurrence of faults. Since the SMT nature of SRT processor enables dynamic scheduling of the hardware resources, the SRT provides a higher efficiency compared to previous commercial fault-tolerant computers, in which hardware components are fully replicated. On the other hand, the Simultaneously and Redundantly Threaded processors with Recovery (SRTR) [78] enhances the SRT processor by adding the recovery capability beyond the detection of the transient faults. While the SRT allows an instruction in the leading thread to commit even before the fault checking occurs, relying on the trailing thread to trigger the detection, in the SRTR, any leading instructions cannot commit before the check for fault occurs since a faulting instruction cannot be undone once the instruction commits. In [78], the authors propose mechanisms to compare the leading and trailing values as soon as the trailing instruction completes, thereby exploiting the time between the completion and commit of leading instructions. This type of helper threading neither contributes to the processor throughput nor improves the program performance, but ensures the correct execution of a program via running the program redundantly on a helper thread.

Implementation of hardware structures in software. One can implement complicated hardware structures in software and execute proper codes on helper threads to manage such software structures. In Simultaneous Subordinate Microthreading (SSMT) [14], subordinate threads, *i.e.*, helper threads, that are written in microcode are executed alongside the primary thread, *i.e.*, main thread, and improve the accuracy of branch prediction. In this work, a branch predictor is implemented in software and runs on microthreads. Solihin *et. al.* introduce the idea of using a User-Level Memory Thread (ULMT) [69, 70], in which a user thread runs on the memory processor and executes the correlation table, built as a software data structure, to perform correlation prefetching. This technique has some advantages such as it is applicable for wide range of address patterns even for those in irregular applications, and it is flexible in that the prefetching algorithm can be customized depending on the workloads. In addition, Master/Slave Speculative Parallelization (MSSP) [85] proposes pre-executing data values as a replacement for a hardware value predictor. In MSSP, one master thread, which is distilled from the original program code to compute critical code only, speculatively runs ahead of the multiple slave threads and provides necessary live-in values a priori. The slave threads are executed and managed as in TLS, but they run faster due to the help from the master thread. Thus, MSSP can be viewed as a combination of TLS and helper threading.

9.1.3 Single-Chip Multithreading Processors

Having presented several techniques that exploit recent single-chip multithreading processors, we now examine two of the most popular research-processor models that support multiple threads on a single chip. Those processors enable both speculative multithreading and helper threading that require light-weight communication between threads. Then we introduce some commercial processors that implement those research processors.

First, Tullsen *et. al.* propose a novel processor microarchitecture, called the Simultaneous Multithreading (SMT) processor [75], in which instructions can be fetched from multiple threads by sharing the hardware resources available in the processor. The SMT processor can be implemented on an existing single-threaded superscalar processor with some modifications. It uses the same pipeline and functional units as the superscalar processor, but some of the hardware structures such as the program counter and register file are replicated to support multiple hardware contexts. The SMT idea starts from the observation that a single program does not fully utilize the available hardware resources in a superscalar processor. Consequently, the most important characteristics of the SMT is to boost the overall processor throughput by making multiple threads effectively share the processor resources. Originally, the SMT processor is designed to execute multiple independent application threads that perform useful computations. However, many research works use the SMT processor to improve a single program performance, and pre-execution is one of the most popular applications in such efforts.

Chip Multiprocessor (CMP) [52] is another processor model that contains multiple processing units on a single chip. As integrated circuit technology becomes more advanced, billions of transistors will be integrated into a single die in the near future. There are several directions to make use of these enormous transistor budgets. One way is to implement much wider and deeply pipelined superscalar processors. However, such a superscalar processor requires very complicated control logic, like renaming registers and forwarding data values, to manage out-of-order execution. Consequently, it takes a much longer time to design and validate such complicated processors, limiting continued boost of the processor clock frequency. Moreover, the power consumption is another critical issue in such complex processor designs. Even though we have the ability to build very complex superscalar processors, many workloads lack enough Instruction-Level Parallelism (ILP) [80] that can fully utilize a superscalar processor's out-of-order execution capability. Thus a wider and deeper superscalar processor may not be able to provide significant additional gain. Having observed these problems, Olukotun *et. al.* propose another way to use billions of transistors, which is the CMP [52]. In the CMP, multiple processors exist on a single chip where each processor is a narrower but faster superscalar processor. Using the CMP, sequential workloads enjoy the benefit of a high-clock rate processor, whose performance is comparable to much more complicated but slower superscalar processors, whereas parallel workloads exploit low communication delays between multiple processing elements on a single die.

Recently, these research processor models started emerging as commercial products. Intel announced the first implementation of Hyper-Threading Technology on the Intel Xeon Processor family [47], which supports two logical processors simultaneously. On the other hand, Barroso *et. al.* introduce the design of the Piranha processor [8]. Piranha is a CMP that consists of 8 Alpha processor cores where each processor core issues instructions in order. In addition, IBM announced the SMT implementation in IBM POWER5 [34], which contains dual processor cores and each core supports two virtual processors. With the emergence of such commercial processors that support multithreading, we expect even more proposals in the future to investigate ways to exploit these processors. We believe our compiler-based pre-execution is one good starting point since it requires little modification to the existing processor hardware, and thus can be easily applied to and evaluated on real silicon. Our work in [35] is the first to experiment with helper threads in a real physical system, and more discussions on physical experimentation will be provided in Section 9.3.4.

9.2 Data Prefetching

As in Section 2.3, we divide data prefetching techniques into two groups: prediction-based prefetching and execution-based prefetching. Execution-based prefetching techniques are further classified into software prefetching, hardware-based prefetching, and thread-based prefetching techniques. This section examines some of the previous proposals on the prefetching techniques and discusses how they are related to pre-execution.

9.2.1 Prediction-Based Prefetching

Chen and Baer [16] propose a hardware-based data prefetching technique which keeps track of the memory-access patterns in a Reference Prediction Table (RPT). The RPT entry consists of 4 fields: *tag* holds the PC of the load/store instructions, *prev_addr* records the last effective address of the entry, *stride* is the difference between two consecutive addresses, and *state* indicates how further prefetch should be generated. They also propose three prefetching schemes to issue a prefetch one iteration ahead (basic), one memory latency time ahead (lookahead), and by considering different loop levels (correlated). For 10 applications from the SPEC92 benchmark suite, their hardware technique successfully reduces the memory stall time. However, in those benchmarks, only an average of 14.3% data references show irregular access patterns and the remaining 85.7% of data references have either zero or constant stride.

Jouppi [33] introduces *stream buffers* as a method to improve direct-mapped cache performance. Compared to the stride prefetchers, stream buffers are designed to prefetch sequential streams of cache lines, independent of the program context. However, Jouppi's design of stream buffers is unable to detect streams that contain non-unit strides. Thus Palacharla and Kessler [56] extend the stream buffer mechanism so that it can also detect non-unit strides without having direct access to the program context.

Joseph and Grunwald [32] propose the design of the Markov prefetcher, which is a hardware-based prefetcher performing *correlation-based prefetching.* The Markov prefetcher is distinguished from Chen and Baer's hardware stride prefetcher [16] by prefetching *multiple reference predictions* from the memory subsystem. Using the Markov model, they monitor the miss-address reference stream and correlate currently missed reference address with future reference addresses. They report an average 54% reduction in memory stall time for various commercial benchmarks.

While prediction-based hardware prefetchers perform well for benchmarks that exhibit regular memory-access patterns, they are not effective for irregular references that are hard to predict. Thus people have investigated various execution-based prefetching techniques to handle irregular memory behavior that are discussed below.

9.2.2 Execution-Based Prefetching

To overcome prediction-based prefetching techniques' inability to handle irregular memory-access patterns, there have been many execution-based prefetching techniques where selected instruction sequence is actually executed to generate accurate prefetches for the delinquent loads. We discuss some important proposals on software prefetching and hardware-based prefetching techniques. Thread-based prefetching techniques are thoroughly covered in Section 9.3 where we discuss various pre-execution techniques.

Software Prefetching

Callahan *et. al* [11] present a compiler algorithm for inserting nonblocking prefetch instructions into the program source code. The initial algorithm prefetches all array references in inner loops one iteration ahead. However, they recognize this scheme issues too many prefetches and introduce a more intelligent scheme based on dependence vectors and overflow iterations. Since they perform simulation at a fairly abstract level and the prefetch overhead is estimated rather than presented, it is hard to evaluate the actual performance impact of their scheme. Their sophisticated scheme is not automated by a compiler and the overflow iterations are calculated by hand. Mowry *et. al.* [49] propose a compiler algorithm to insert prefetch instructions into the code that operates on dense matrices, and they implement the algorithm in the SUIF compiler infrastructure. Their algorithm identifies the memory references that are likely to miss in the cache by locality analysis and covers only those loads to reduce the overhead with software prefetching. For a collection of scientific applications, they show significant speedup with their prefetching technique.

While the above two works target array accesses in scientific workloads, Luk and Mowry [43] propose compiler algorithms to prefetch for recursive data structures found in pointer-based applications. Luk and Mowry introduce three compiler algorithms, *i.e.*, greedy prefetching, history-pointer prefetching (also known as *jump-pointer prefetching*), and data-linearization prefetching to remap heap objects, and automate the greedy prefetching algorithm in SUIF. As pointed out earlier, in software prefetching, the integration of the prefetch instructions into the main program code forces prefetching to be strictly tied to the progress of the main thread. Also, poor computation capability of software prefetching limits its use for more complicated address calculation patterns that also include control flow.

Hardware-Based Prefetching

Among many execution-based hardware prefetching techniques, we examine two proposals here. Roth *et. al.* [60] introduce a dynamic scheme that captures the access patterns of the linked data structures to generate future memory reference addresses. Their technique exploits the dependence relationships between the loads that produce addresses and loads that consume these addresses in the pointer-chasing traversals. Once the producer-consumer pairs are identified, they generate an internal representation for the associated structure and its traversal pattern. The constructed representation code is executed on a prefetch engine to generate prefetches for load instructions in the linked data structures. Their technique achieves speedups of up to 25% for applications in the Olden benchmark suite [12], which consists of various pointer-intensive programs.

Another example that uses a hardware prefetch engine to generate prefetches for future memory references is *Multi-Chain Prefetching* by Kohout *et. al* [38]. In this work, they perform offline analyses to extract traversal information for the various program structures such as loops or tree traversal. Then the extracted *descriptor* information is fed into the prefetch engine and the necessary code is instrumented in the original program code to pass appropriate values to the prefetch engine at runtime and initiate prefetching. Their prefetch engine can prefetch multiple independent pointer chains simultaneously, thereby exploiting inter-chain memory parallelism in workloads.

While both techniques are effective to handle pointer-chasing traversal patterns that are usually found in the memory references of the linked data structures, they introduce a new hardware structure, *i.e.*, prefetch engine, which can be viewed as a small memory coprocessor to solely perform data prefetching. With the advent of single-chip multiprocessors where multiple hardware contexts are tightly coupled together, many people propose thread-based prefetching techniques that utilize the general-purpose hardware contexts to run prefetching threads. Those thread-based prefetching techniques are discussed in more detail in the next section.

9.3 Pre-Execution

In this section, we examine some important related works on pre-execution. As shown in Figure 1.1, we classify previous works on pre-execution into three groups based on when and how helper threads are constructed.[1] Section 9.3.1 discusses four prior works on compiler-based pre-execution that analyzes either the program source code or the intermediate representation of a program to construct helper threads. Section 9.3.2 examines four linker-based pre-execution techniques where helper threads are constructed at link time by analyzing program binaries using post-pass binary analysis tools. Section 9.3.3 covers some hardware-based pre-execution techniques where

[1]We exclude dynamic optimizer-based approach since we could not find any previous works in this category.

special hardware structures analyze instruction traces to extract helper threads at runtime. Finally, Section 9.3.4 introduces two prior works that experiment with prefetching helper threads in a real physical system.

9.3.1 Compiler-Based Pre-Execution

In our prior work [36], we propose various compiler algorithms to optimize helper threads and show the design of a compiler framework to construct effective helper threads in the form of C source code. Our compiler framework is implemented in the SUIF compiler infrastructure with the help from an offline memory profiler and a program slicer. To the best of our knowledge, this is the first work to fully automate a source-to-source compiler that generates prefetching helper threads. To build the compiler framework, we adopt some of the conventional compiler algorithms, *e.g.*, program slicing [9, 44] and loop parallelization [21, 54, 55], but use them in a different way, *i.e.*, to optimize prefetching helper threads. While the helper threads in [36] only perform data prefetching, we believe the compiler algorithms and compiler framework can be also applied to other latency-tolerating purposes such as early resolution of branch outcomes.

Our follow-up work [37] extends the work in [36] and introduces the design of a few reduced compiler frameworks for pre-execution. We replace the external tools and profiling steps in our aggressive compiler, presented in [36], with static compiler analyses in order to reduce the complexity of the compiler design. In addition to demonstrating the compiler algorithms and design, we also conduct various experiments on a SimpleScalar-based SMT processor simulator and report interesting results that prove compiler-based pre-execution effectively eliminates the cache misses for a variety of workloads and significantly improves the processor performance. In this thesis, Chapters 4 and 5 are based on these two prior works of ours, *i.e.*, [36, 37].

While the above two works on compiler-based pre-execution make use of the SUIF compiler infrastructure to generate helper threads, our recent prior work [35] introduces the design and algorithms of a pre-execution optimization module in the Intel compiler. In this work, helper threads are optimized and generated in the form of the Intel compiler's intermediate representation and the backend compiler directly produces binaries targeting the Intel's IA-32 architecture. The constructed helper threads are attached to the binary and launched at runtime when the main thread encounters the selected pre-execution regions. Chapter 6 provides more details about this pre-execution compiler module in the Intel compiler.

We introduce yet another previous work by Luk [42], which is the first to construct helper threads and perform pre-execution at the program's source-code level. Compared to the above three compiler-based pre-execution works [35, 36, 37], Luk's work is different in the following three aspects. First, while helper threads in [35, 36, 37] are generated using fully automatic compiler frameworks, the helper thread construction in [42] is done by manual analysis of a program. Moreover, all optimization algorithms are applied on the intermediate representation of a program in [35, 36, 37] while Luk's work directly analyzes the program C source code. Finally, all our prior works perform code cloning to generate a separate code for helper threads; [36, 37] create separate subroutines for helper threads and [35] uses multi-entry threading in the Intel compiler. On the other hand, in Luk's work, the main thread and the helper threads execute the same code. Although it is guaranteed that helper threads compute the delinquent load addresses accurately, such code sharing between the main thread and the helper threads does not allow the programmer or compiler to optimize the code aggressively as is done in our compiler frameworks [35, 36, 37]. Consequently, helper threads may not acquire enough speed advantage over the main thread.

9.3.2 Linker-Based Pre-Execution

While compiler-based pre-execution analyzes the program in the form of source code or intermediate representation at compile time, Liao *et. al.* [41] introduce the design of a post-pass binary tool, which analyzes a program binary and constructs helper threads at the binary level. The generated helper threads are attached to the original program binary. Their work is the first to automate the entire process of constructing helper threads at the binary level. In this paper, they implement algorithms to generate the basic triggers and the chaining triggers introduced in the Speculative Precomputation paper [20] and perform various experiments using the research Itanium processor model.

Execution-Based Prediction (EBP) [84] is another pre-execution technique where helper threads are extracted from the binary. In this work, speculative slices, or so-called *backward slices* [87], are extracted using binary analysis. While most other pre-execution works focus on the use of helper threads for data prefetching, EBP targets both cache-missing loads and frequently mispredicted branch instances. They first identify the Performance Degrading Events (PDEs) and construct speculative slices manually. Like ours, their slices are speculative, and thus this technique requires speculation hardware support to preserve the program correctness. In addition to slice construction and execution, this work also shows a way to correlate the precomputed branch outcome to the corresponding branch instance.

Roth and Sohi's Data-Driven Multithreading (DDMT) [63] prioritizes the critical instructions in a program and executes them on speculative threads, called Data-Driven Threads (DDTs). DDTs are data driven, and thus the dynamic instructions are not necessarily contiguous as in the original program. The DDT is extracted from program traces using an offline binary analysis tool. All the instructions executed in DDTs are used for triggering the critical events such as cache misses and branches. Therefore, the basic idea behind DDMT is very similar to that of backward slicing [87]. There is a unique contribution in this work; while the results computed in helper threads are usually discarded in most pre-execution techniques, they integrate the results of speculatively executed instructions into the main computation using a technique called *register integration* [62] so that helper threads actually perform useful computations other than data prefetching or branch outcome resolution.

Collins *et. al.* propose the idea of Speculative Precomputation (SP) [20]. Although the same authors publish the follow-up paper, Dynamic Speculative Precomputation (DSP) [19], immediately after the SP work, these two papers are quite different in two aspects: identification of delinquent loads and construction of precomputation slices. First, to identify the delinquent loads, SP performs offline memory profiling. For construction of speculative threads, SP also uses offline trace analysis and the constructed helper threads are appended to the program binary. On the other hand, in DSP, every step of pre-execution is fully automated using special hardware structures, and both delinquent load identification and helper thread construction are done at runtime.

9.3.3 Hardware-Based Pre-Execution

Dynamic Speculative Precomputation (DSP) [19] is one of the most advanced hardware-based pre-execution techniques. In DSP, the necessary steps for pre-execution, *i.e.*, identification of the delinquent loads, construction of the *precomputation slices* (p-slices), and initiation of the helper threads, are all implemented in hardware. To perform pre-execution, the important cache-missing loads are identified at runtime using the Delinquent Load Identification Table. Then a p-slice is constructed using a hardware structure called the Retired Instruction Buffer (RIB). They use an RIB size of 512 entries, and thus the scope of their analysis is inherently limited to at most 512 post-retirement instructions. The constructed p-slice is stored in the Slice Cache (SC) and the corresponding trigger instruction for each p-slice is marked. Later in the program execution when a trigger instruction is detected, the speculative threads fetch their instructions from the SC. This paper also introduces several techniques to effectively run p-slices such as the CHAINING scheme.

Sundaramoorthy *et. al.* propose the Slipstream Processor [73]. In this processor model, two types of threads exist: advanced stream (A-stream) and redundant stream (R-stream). The A-stream is a speculative and shortened version of the original program, thus running ahead of the full program and providing control and dataflow outcomes. It also performs data prefetching for the R-stream so that the R-stream can speedup. In A-stream, dynamic instances of ineffectual instructions are bypassed if such bypassing does not hurt the correct progress of the A-stream. On the other hand, the R-stream, a trailing program, validates the execution of the A-stream to ensure the correct execution of the program. The A-stream is constructed using the Instruction-Removal predictor (IR-predictor), which is based on a conventional trace predictor. One potential drawback is the complete processor model requires nontrivial hardware support.

In addition, there are more works that make use of special hardware structures to construct helper threads and control pre-execution. Annavaram *et. al.* introduce a novel data prefetching technique, called Dependence Graph Precomputation (DGP) [6]. In DGP, the Dependence Graph Generator analyzes instructions in the instruction fetch queue to construct the dependence graph. Runahead Processing is proposed by Dundas and Mudge [24], which is one of the early works in pre-execution research. In this proposal, a helper thread is not constructed explicitly. Instead, upon detection of an L1 miss, the processor *pre-executes* subsequent instructions while the cache miss is serviced so that the memory access and useful computation can be overlapped. Akkary and Driscoll propose a Dynamic Multithreading Processor [4]. In this processor model, helper threads are automatically constructed using hardware at procedure or loop boundaries, and executed speculatively on a simultaneous multithreading pipeline. While all such hardware-based pre-execution techniques rely on special hardware support to construct and run helper threads, our work is fully controlled by compiler-generated helper threads with little modifications to the existing SMT processors.

9.3.4 Physical Experimentation with Prefetching Helper Threads

Many research ideas are evaluated in a simulation-based environment to prove the proposed technique is promising. In addition, by probing different processor components that are modeled in the simulator, simulation-based experiments often provide various information about the behavior of benchmarks while the technique is applied. Although simulation-based evaluation is valuable, an eventual goal of any proposed research idea is to be applied to a real machine and possibly, be productized. Thus it is very important to evaluate a research idea in a real physical system and uncover insights into the technique that are hard to learn from simulation-based evaluation.

To our knowledge, only two prior works evaluate prefetching helper threads in a real machine. The first work on such physical-system-based evaluation of helper threads is our work in [35]. We build a compiler optimization module in the Intel compiler to construct prefetching helper threads and conduct experiments in the Intel Pentium 4 processor with Hyper-Threading Technology. Our experimental results show that prefetching helper threads indeed provide wall-clock speedup on real silicon for a number of real-world benchmarks. In addition, we observe some of the impediments in a physical system need to be addressed, in order to achieve even more speedup with helper threads. In this thesis, Chapter 7 is based on the physical experiments in [35].

While helper threads in [35] target the Intel's IA-32 architecture, Wang *et. al.* evaluate prefetching helper threads on an experimental Itanium 2 machine [81]. In this work, they introduce Virtual Multithreading (VMT), which is a novel form of switch-on-event user-level multithreading and is capable of fly-weight multiplexing of event-driven thread executions on a single processor without any OS intervention. The concept of VMT is prototyped on an Itanium 2 processor using the existing Processor Abstraction Layer firmware mechanism without any extra hardware support. On a 4-way MP physical system equipped with the VMT-enabled Itanium 2 processors, they show that helper threading via the VMT mechanism can achieve significant performance gains for a diverse set of real-world workloads ranging from single-threaded workstation benchmarks to heavily multithreaded large scale decision support systems using the IBM DB2 Universal Database [31].

Chapter 10

Conclusion

10.1 Summary of Contributions and Implications of the Research

We believe this dissertation has several valuable contributions in the areas of optimizing compilation techniques as well as experiments in both the simulation-based evaluation environment and real physical system. Those contributions are summarized below.

First, we propose several compiler optimization algorithms to construct effective helper threads. To be effective, it is very important for helper threads to run in front of the main computation and trigger cache misses early. In order to expedite the execution of the helper threads, we develop various optimization techniques such as program slicing to remove noncritical code in the helper threads, prefetch conversion to eliminate blocking in the helper threads, and speculative loop parallelization to overlap even more memory accesses simultaneously. Those compiler algorithms are implemented to build compiler frameworks that automatically produce prefetching helper threads. In this thesis, we introduce the design of compiler frameworks that include the offline profiling tools to collect target information, compiler optimization modules, and code generators. Those compiler optimization algorithms are implemented in the SUIF compiler infrastructure to generate C source code to which helper threads are attached. In addition, we also propose alternative ways to design much simpler compiler frameworks by replacing some critical components in our aggressive compiler with static compiler algorithms.

Second, having developed compiler frameworks to construct helper threads, we evaluate the effectiveness of our compilers for pre-execution. We conduct various experiments to evaluate the helper threads generated by our SUIF-based compiler frameworks on a SimpleScalar-based SMT processor simulator. We show that compiler-based pre-execution is indeed a very promising latency-tolerance technique and significantly improves the single-thread performance for various benchmarks. Our experimental results clearly show all three optimization algorithms, *i.e.*, program slicing, prefetch conversion, and loop parallelization, contribute to the performance improvement, and choosing the right thread initiation scheme is important as well. We also observe that speculative loop parallelization rarely disrupts the correct execution of the cache-miss kernels, and thus exploits memory-level parallelism that resides in a workload.

Third, while simulation-based evaluation of a research idea is a good first step, applying such idea to a real physical system is very important in that we can verify whether the performance on a simulator is achievable and also we can uncover valuable insights that are hardly learned by simulation-based experiments. To enable such physical-system evaluation, we help a group of people at Intel to develop the compiler optimization module in the Intel compiler to generate effective helper threads. We also provide the experimental methodology to evaluate the prefetching helper threads on an Intel Pentium 4 processor with Hyper-Threading Technology. Our experiments not only include the real processor, but the production-quality compiler infrastructure, real OS, and real-world benchmarks as well.

Fourth, from our experience with helper threads, we identify several impediments to speedup in real silicon. First, some critical hardware structures in hyper-threaded processors are hard-partitioned in the MT mode so that invoking a helper thread possibly degrades the performance of the useful computation thread when the helper thread does not help. Moreover, the thread synchronization in a physical system costs thousands of cycles, which limits the fine-grain thread management. To overcome those impediments, we observe having certain runtime mechanisms is crucial so that we can monitor the program behavior using various hardware PMCs and the helper threads can adapt to the dynamic behavior. Such dynamic throttling of helper threads can effectively eliminate the helper thread invocations that are not helpful to improve the performance of the main computation, thereby providing even more speedup on real silicon.

Finally, we apply compiler-based pre-execution in a multiprogramming environment and examine how helper threads can boost the overall throughput of selected multiprogrammed workloads. By using our aggressive SUIF-based compiler to construct helper threads, we introduce a hardware mechanism to arbitrate multiple main threads for limited number of hardware contexts for helper threads. Our experimental results show, when performing pre-execution, sharing of processor resources via simultaneous multithreading provides more throughput than running multiprogrammed workloads in time-sliced manner. Moreover, we observe the performance boost with helper threads depends on how long and how many hardware contexts each useful computation thread needs at runtime. To further investigate the behavior of helper threads in a multiprogramming environment, more work needs to be done and it is discussed in the next section.

10.2 Future Direction

We believe our work has opened up several directions for many interesting research ideas. We introduce two potential future works that are not only limited to compiler-based helper thread optimization and construction, but also include various kinds of hardware and system-level support, in order to further improve single-thread performance or boost the overall system throughput in a multiprogramming environment.

From our experiments with helper threads in a physical system, we learned in order to overcome the hardware resource contention problem and achieve further speedup, the helper threads must adapt to the dynamic behavior of a program by throttling their activation and deactivation at a very fine granularity. In addition, certain important information is unknown at compile time, for instance, the target platform on which the generated binary will run, and the input set, which is used to run the workload. Depending on the configuration of the target platform, *e.g.*, cache size or clock frequency, or the input parameters for a program run, the set of delinquent loads and selected pre-execution regions may vary. To take into account all these dynamic and unknown factors associated with program execution, we can consider a new form of dynamic helper threading where different versions of helper threads that target the same code region are constructed at compile time and the most optimal helper threads are selected at runtime with support from performance monitoring and helper thread selection algorithms. We believe this idea opens up two big research issues. The first one is how to construct several versions of helper threads that are differently optimized based on the various factors mentioned above. This requires more systematic identification of the dynamic factors that affect the performance of pre-execution, categorization of different situations and corresponding optimization methods, and development of the compiler optimization algorithms. Secondly, we need to build certain runtime mechanisms that monitor the dynamic behavior of the workload and profile necessary hardware performance monitoring counters with very low overhead, correlate the profile results with the helper thread performance, and make intelligent decisions to choose the optimal helper thread. Since the program behavior can constantly vary across the entire execution, different helper threads can be selected for different time phases.

In Chapter 8, we evaluate compiler-based pre-execution in a multiprogramming environment and show that arbitration of hardware resources for multiple main threads is crucial to achieve a good performance improvement with helper threads. However, we view it as only an initial step and feel that more work needs to be done to better understand the behavior and usefulness of helper threads in a multiprogramming environment. For the experiments in Chapter 8, we assumed certain number of hardware contexts are dedicated to run only main threads or helper threads, thereby emulating some sort of hard-partitioning of available hardware contexts. Rather than limiting the usage of a context to either main thread or helper thread, we believe it to be more realistic and interesting to assume competition between the main threads and helper threads for the spare hardware contexts. The realization of such a computing environment requires various support. Having lighter-weight thread synchronization mechanisms is especially crucial and certain runtime mechanism in either hardware or the OS is necessary to monitor the throughput of the whole workloads and the effect of launching helper threads instead of running useful computation threads. In addition to performance monitoring, some sort of algorithms regarding how to arbitrate the main threads and the helper threads for the available hardware contexts in a system also need to be developed.

BIBLIOGRAPHY

[1] T. Aamodt, P. Marcuello, P. Chow, P. Hammarlund, and H. Wang. Prescient Instruction Prefetch. In *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation (in conjunction with Micro 35)*, pages 3–10, Istanbul, Turkey, November 2002.

[2] S. G. Abraham, R. A. Sugumar, B. R. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, Austin, TX, December 1993. IEEE/ACM.

[3] A. Agarwal, B.-H. Lim, D. Krantz, and J. Kubiatowicz. April: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, WA, June 1990. ACM.

[4] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 226–236, Dallas, TX, November 1998. IEEE/ACM.

[5] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? SRC Technical Note 1997-016a, Digital, July 1997.

[6] M. Annavaram, J. M. Patel, and E. S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 52–61, Goteborg, Sweden, June 2001. ACM.

[7] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically Allocating Processor Resources between Nearby and Distant ILP. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 26–37, Goteborg, Sweden, June 2001. ACM.

[8] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multipro-

cessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, Vancouver, Canada, June 2000. ACM.

[9] D. Binkley and K. Gallagher. *Program Slicing*, volume 62. Academic Press, San Diego, CA, 1996.

[10] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.

[11] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991. ACM.

[12] M. C. Carlisle. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines, PhD Thesis. Department of computer science, Princeton University, June 1996.

[13] F. Chang and G. A. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 1–14, New Orleans, LA, February 1999. ACM.

[14] R. S. Chappell, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 186–195, Atlanta, GA, May 1999. ACM.

[15] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. Difficult-Path Branch Prediction Using Subordinate Microthreads. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 307–317, Anchorage, AK, May 2002. ACM.

[16] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[17] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, May 1999. ACM.

[18] R. F. Cmelik and D. Keppel. Shade: A Fast Instruction Set Simulator for Execution Profiling. TR 93-12, Sun Microsystems, July 1993.

[19] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic Speculative Precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 306–317, Austin, TX, December 2001. IEEE/ACM.

[20] 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. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, Goteborg, Sweden, June 2001. ACM.

[21] R. Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, University Park, PA, August 1986. IEEE.

[22] M. Dubois and Y. H. Song. Assisted Execution. CENG Technical Report 98-25, University of Southern California, October 1998.

[23] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An Overview of the Intel IA-64 Compiler. *Intel Technology Journal*, 3(4), November 1999.

[24] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. In *Proceedings of the 11th International Conference on Supercomputing*, pages 68–75, Vienna, Austria, July 1997. ACM.

[25] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proceedings of the 31st*

International Symposium on Microarchitecture, pages 59–68, Dallas, TX, December 1998. IEEE/ACM.

[26] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages*, 9(3):319–349, July 1987.

[27] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. In *Transactions on Computers, 45(5)*, pages 552–571. IEEE, May 1996.

[28] R. Ghiya, D. Lavery, and D. Sehr. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 47–58, Snowbird, UT, June 2001. ACM.

[29] J. L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[30] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal (Issue on Pentium 4 Processor)*, 4(1), February 2001.

[31] IBM. IBM DB2 Product Family. http://www.ibm.com/software/data/db2.

[32] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 252–263, Denver, CO, June 1997. ACM.

[33] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990. ACM.

[34] R. Kalla, B. Sinharoy, and J. Tendler. POWER5: IBM's Next Generation POWER Microprocessor. In *A Symposium on High Performance Chips (Hot Chips 16)*, Palo Alto, CA, August 2003. IEEE.

[35] 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 Hyper-Threaded Processors. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, pages 27–38, Palo Alto, CA, March 2004. IEEE.

[36] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–170, San Jose, CA, October 2002. ACM.

[37] D. Kim and D. Yeung. A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code. *ACM Transactions on Computer Systems*, 22(3):326–379, August 2004.

[38] N. Kohout, S. Choi, D. Kim, and D. Yeung. Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes. In *Proceedings of the 10th Annual International Conference on Parallel Architectures and Compilation Techniques*, pages 268–279, Barcelona, Spain, September 2001. IEEE.

[39] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[40] S. S. Liao, A. Diwan, R. Bosch, A. Ghuloum, and M. S. Lam. SUIF Explorer: an Interactive and Interprocedural Parallelizer. In *Proceedings of the 7th International Symposium on Principles and Practice of Parallel Programming*, pages 37–48, Atlanta, GA, May 1999. ACM.

[41] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 117–128, Berlin, Germany, June 2002. ACM.

[42] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, Goteborg, Sweden, June 2001. ACM.

[43] C.-K. Luk and T. C. Mowry. Automatic Compiler-Inserted Prefetching for Pointer-Based Applications. *IEEE Transactions on Computers (Special Issue on Cache Memory and Related Problems)*, 48(2):134–141, February 1999.

[44] J. R. Lyle and D. W. Binkley. Program slicing in the presence of pointers. In *Proceedings of the 3rd Annual Software Engineering Research Forum*, pages 255–260, Orlando, FL, November 1993.

[45] J. R. Lyle, D. R. Wallace, J. R. Graham, K. B. Gallagher, J. P. Poole, and D. W. Binkley. Unravel: A CASE Tool to Assist Evaluation of High Integrity Software. NISTIR 5691, National Institute of Standards and Technology, August 1995.

[46] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *Proceedings of EuroPar '99*, pages 716–726, Toulouse, France, August 1999. Springer-Verlag.

[47] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. In *Intel Technology Journal (Issues on Hyper-Threading)*, volume 6, February 2002.

[48] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-Processors: An Implementation of Operation-Based Prediction. In *Proceedings of the 15th International Conference on Supercomputing*, pages 321–334, Sorrento, Italy, June 2001. ACM.

[49] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, Massachusetts, October 1992. ACM.

[50] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110, Anchorage, AK, May 2002. ACM.

[51] Sun Microsystems. Niagara: A Torrent of Threads. http://www.aceshardware.com/read.jsp?id=65000292. April 2004.

[52] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the 7th International Symposium on Architectural Support for Parallel Languages and Operating Systems*, pages 2–11, Cambridge, MA, October 1996. ACM.

[53] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache Miss Heuristics and Preloading Techniques for General-Purpose Programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 243–248, Ann Arbor, MI, November 1995. IEEE/ACM.

[54] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[55] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[56] S. Palacharla and R. E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, April 1994. ACM.

[57] V.-M. Panait, A. Sasturkar, and W.-F. Wong. Static Identification of Delinquent Loads. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, pages 303–314, Palo Alto, CA, March 2004. IEEE.

[58] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, Vancouver, Canada, June 2000. ACM.

[59] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995.

[60] A. Roth, A. Moshovos, and G. S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, San Jose, CA, October 1998. ACM.

[61] A. Roth, A. Moshovos, and G. S. Sohi. Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In *Proceedings of the 13th Annual International Conference on Supercomputing*, pages 356–364, Rhodes, Greece, June 1999. ACM.

[62] A. Roth and G. S. Sohi. Register Integration: A Simple and Efficient Implementation of Squash Reuse. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 223–234, Monterey, CA, December 2000. IEEE/ACM.

[63] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Conference on High-Performance Computer Architecture*, pages 191–202, Nuebo Leone, Mexico, January 2001. IEEE.

[64] A. Roth and G. S. Sohi. A Quantitative Framework for Automated Pre-Execution Thread Selection. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 430–441, Istanbul, Turkey, November 2002. IEEE/ACM.

[65] B. Smith. *The Architecture of HEP.* MIT Press, Cambridge, MA, 1985.

[66] M. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[67] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proceedings of the 9th International Conference on Architectural Support for*

Programming Languages and Operating Systems, pages 234–244, Cambridge, MA, November 2000. ACM.

[68] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995. ACM.

[69] Y. Solihin, J. Lee, and J. Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, Anchorage, AK, May 2002. ACM.

[70] Y. Solihin, J. Lee, and J. Torrellas. Correlation Prefetching with a User-Level Memory Thread. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):563–580, June 2003.

[71] Standard Performance Evaluation Corporation. SPEC CPU2000 V1.2. http://www.specbench.org/osg/cpu2000/.

[72] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, Las Vegas, NV, February 1998. IEEE.

[73] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–202, Cambridge, MA, November 2000. ACM.

[74] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su. Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal (Issue on Hyper-threading)*, 6(1), February 2002.

[75] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995. ACM.

[76] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, Philadelphia, PA, May 1996. ACM.

[77] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 54–58, Orlando, FL, January 1999. IEEE.

[78] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery Using Simultaneous Multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98, Anchorage, AK, May 2002. ACM.

[79] Intel Corporation. VTune Performance Analyzer. http://developer.intel.com/software/products/VTune/index.html.

[80] D. W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, CA, April 1991. ACM.

[81] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, and J. P. Shen. Helper Threads via Virtual Multithreading on An Experimental Itanium 2 Machine. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct 2004. ACM.

[82] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, San Diego, CA, March 1981. ACM.

[83] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[84] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, Goteborg, Sweden, June 2001. ACM.

[85] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 85–96, Istanbul, Turkey, November 2002. IEEE/ACM.

[86] C. B. Zilles, J. S. Emer, and G. S. Sohi. The Use of Multithreading for Exception Handling. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 219–229, Haifa, Israel, November 1999. IEEE/ACM.

[87] C. B. Zilles and G. S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, Vancouver, Canada, June 2000. ACM.