Modeling and Optimization of Speculative Threads

by

Tor M. Aamodt

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy,
Department of Electrical and Computer Engineering,
University of Toronto

Abstract

Modeling and Optimization of Speculative Threads

Tor M. Aamodt
Doctor of Philosophy
Department of Electrical and Computer Engineering
University of Toronto
2006

This dissertation proposes a framework for modeling the control flow behavior of a program and the application of this framework to the optimization of speculative threads used for instruction and data prefetch. A novel form of helper threading, *prescient instruction prefetch*, is introduced in which helper threads are initiated when the main thread encounters a spawn point and prefetch instructions starting at a distant target point. The target identifies a code region tending to incur I-cache misses that the main thread is likely to execute soon, even though intervening control flow may be unpredictable. The framework is also applied to the compile time optimization of *simple p-threads*, which improve performance by reducing data cache misses.

The optimization of speculative threads is enabled by modeling program behavior as a Markov chain based on profile statistics. Execution paths are considered stochastic outcomes, and program behavior is summarized via path expression mappings. Mappings for computing reaching, and posteriori probability; path length mean, and variance; and expected path footprint are presented. These are used with Tarjan's fast path algorithm to efficiently estimate the benefit of spawn-target pair selections.

Two implementation techniques for prescient instruction prefetch—*direct pre-execution*, and *finite state machine recall*—are proposed and evaluated. Further, a hardware mechanism for reducing resource contention in direct pre-execution called the *YAT-bit* is proposed and evaluated. Finally, a hardware mechanism, called the *safe-store*, for enabling the inclusion of stores in helper threads is evaluated and extended. Average speedups of 10.0% to 22% (depending upon memory latency) on a set of SPEC CPU 2000 benchmarks that suffer significant I-cache misses are shown on a research Itanium® SMT processor with next line and streaming I-prefetch mechanisms that incurs latencies representative of next generation processors. Prescient instruction prefetch is found to be competitive against even the most aggressive research hardware instruction prefetch technique: *fetch directed instruction prefetch.*

The application of the modeling framework to data prefetch helper threads yields results comparable with simulation based helper thread optimization techniques while remaining amenable to implementation within an optimizing compiler.

Acknowledgements

I would like to thank my thesis supervisor Professor Paul Chow for his guidance over the past several years and for encouraging me to pursue my own directions in my research. I am very thankful to the members of my Ph.D. thesis committee, Professor Todd C. Mowry, Professor Andreas Moshovos, Professor Greg Steffan, and Professor Ian F. Blake for their invaluable feedback on this dissertation.

I would also like to thank Dr. Hong Wang and Dr. John P. Shen for the opportunity to work with them and other members of Intel Research during the initial phase of this research at Intel Corporation's Microarchitecture Research Lab in Santa Clara California. Other individuals from Intel Santa Clara that contributed to a collegial atmosphere during my internship include Perry H. Wang, Shih-Wei Steve Liao, Murali Annavaram, Ed Grochowski, Trung A. Diep, Ryan Rakvic, Mauricio Breternitz Jr., and Gerolf Hoflehner. Similarly, Srikanth Srinivasan, Jared Stark, Konrad Lai, and Shih-Lien Lu from Intel MRL-uAL in Oregon; Edward (Ned) Brekelbaum from Intel MRL-uAL in Austin; Pedro Marcuello and Antonio Gonzalez from Intel Barcelona; and Per Hammarlund from Intel DPG in Oregon made working at Intel an enjoyable and educational experience.

My peers in the Department of Electrical and Computer Engineering at the University of Toronto have made the time spent in graduate school a pleasure. In particular, I would like to thank Amy Wang, Andy Ye, David Tam, Guy Lemieux, Imad A. Ferzli, Jason Anderson, Kostas Pagiamtzis, Lesley Shannon, Marcus van Ierssel, Mark Stoodley, Mathew Zaleski, Mazen Saghir, Navid Azizi, Patrick Doyle, Paul Kundarewich, Peter Jamieson, Reza Azimi, Sirish Raj Pande, Valavan Manohararajah, Vincent Charles Gaudet, and Warren Gross. Since I've been at UofT so long, I might was well apologize in advance to those whose names I've surely missed! Thanks also to Eugenia Distefano and Peter Pereira for their always lightning fast IT support.

I've also enjoyed discussions with various people at computer architecture conferences including Amir Roth, Craig B. Zilles, Jamison D. Collins, Guri Sohi, and Dean M. Tullsen.

My family deserves a very special thanks. Foremost, my wife Dayna has been a constant source of encouragement and inspiration. My parents and in-laws have been very supportive of my desire to obtain my doctorate and I am thankful. I am thankful for my mother's consistent encouragement. My father's study of electrical engineering while I was young, combined with the TRS 80 Color Computer he got me soon after he graduated, surely played a part in the subsequent path of my academic studies. My mother-in-law and father-in-law have helped my wife and myself out in immeasurable number of ways over the past several years and I am very grateful. My wife's grandmother (Bubby) and my grandmother (Nana) will be happy to know I can finally relax on my weekends as would my wife's dearly departed grandfather (Zaidy). My grandfather (Grandpa) implanted the goal of pursuing a Ph.D. from a very early age and even suggested computer engineering back in the days of the Intel 486 (when I was thinking more of becoming an architect that designs buildings rather than computers—I eventually discovered the wisdom of his suggestion while pursuing my undergraduate degree in the Engineering Science program at UofT and thinking hard about what I wanted to study in graduate school).

Financial support for this research was provided by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and by an internship with Intel Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation proposes and evaluates a novel framework for modeling the dynamic control flow behavior of computer programs, and applies this framework to modeling and optimizing the performance impact of speculative threads. Speculative threads are a promising approach to improving microprocessor performance on applications exhibiting complex control flow and/or memory access patterns. They improve performance by increasing the amount of useful computational work completed per clock cycle. The earliest research on speculative threads proposed dividing the computation of a single-threaded application into tasks that could be executed concurrently–a technique known variously as *Multiscalar* [SBV95], *thread-level speculation,* [SM98, SCZM00], or *speculative multithreading* [MGT98].

Recently, several forms of speculative *helper threads* have been proposed for reducing the impact of challenging microarchitectural performance bottlenecks in modern microprocessors that have resisted hardware only solutions. While speculative multithreading improves performance by overlapping the execution of instructions from distinct tasks, helper threads overlap memory accesses and/or precompute branch outcomes to reduce disruptions to the flow of instructions through the microprocessor. Instead of focusing on improved *instruction level parallelism* – the latter being the focus of earlier studies on thread-level speculation, helper threads that prefetch instructions or data can be viewed as a technique for enhancing *memory-level parallelism* [Gle98]. Memory level parallelism is enhanced when the latency of cache misses is overlapped with computation or other cache misses [CFA04].

This chapter describes the motivation and background for this dissertation, the methodology employed, the contributions this dissertation makes, and finally summarizes the dissertation's organization.

1.1 Motivation

Microprocessors form the heart of today's computing systems. They range in size from relatively small embedded processors for signal processing applications such as anti-lock brakes, to the largest and most power-hungry microprocessors used in parallel computing systems for business and scientific computing. Improvements in microprocessor performance per unit cost lead directly to improvements in application responsiveness—i.e., the time it takes to finish a single task—and throughput—i.e., the number of independent tasks that can be completed per unit time. Just as significantly, these performance improvements also enable improvements in the *quality* of software applications by making the use of more sophisticated software features feasible. For example, increasing microprocessor performance has enabled the use of more sophisticated algorithms in applications such as word processors, which can use additional performance to add features such as speech recognition and background grammar/spell checking. While individual improvements in performance are incremental, the accumulated improvements have a dramatic effect on the use of computing by continually enabling applications that would not have been practical earlier. The recent emergence of cell phones with integrated *personal digital assistant* (PDA) and digital cameras is an example of a new application that has relied upon improvements in computing performance to meet the small form factor required for consumers to find such a device useful. Another recent example is the "iPod" music player from Apple Computer [Inc05].

Underlying the historical trend of increasing performance per unit cost are advances in semiconductor technology that have yielded exponential improvements in both transistor switching frequencies, and the number of transistors that can be integrated on a single chip (which now typically doubles every 18-20 months—a trend sometimes referred to as "Moore's Law" [Moo65]). Developing a microprocessor that makes the best use of the logic gates and wires of a given process technology to deliver maximum microprocessor performance is one of the primary goals of the field of computer architecture. While there are other goals of importance in designing a computing system—such as maintaining application compatibility with prior microprocessors, increasing memory capacity, reducing power consumption, improving fault tolerance, and supporting secure computing to protect digital content—performance (or performance per unit cost) has traditionally been, and continues to be one of the main factors considered when buying a computing system.

Performance can be quantified as the reciprocal of an application's total execution time. Execution time in turn can be expressed using the "iron law" of computer performance, which expresses total execution time as a product of the average number of clock cycles it takes to execute an instruction, the time it takes to complete one clock cycle, and the total number of instructions executed:

$$\text{Execution Time} = (\text{Cycles Per Instruction}) \cdot (\text{Cycle Time}) \cdot (\text{Total Instructions Executed})$$

This formula applies to any instruction set and implementation technology. In recent years it has become customery express the relationship as:

$$\text{Execution Time} = \frac{(\text{Total Instructions Executed})}{(\text{Instructions Per Cycle}) \cdot (\text{Clock Frequency})}$$

Where *instructions per cycle* (IPC) is used rather than CPI because modern microprocessors are designed to execute multiple instructions per cycle.

Reducing total instructions executed to improve performance is usually accomplished by careful instruction set design and through the use of an optimizing compiler. Increasing clock frequency is achieved by reducing semiconductor technology feature sizes. In any one process technology generation, detailed processor design considerations may significantly influence clock frequency (an excellent example of this is provided in Papworth et al.'s summary of the performance tuning effort for the Pentium Pro microprocessor from Intel Corporation [Pap96]). Even though the impact of such design decisions on cycle time is dwarfed by the exponential growth in clock frequency over many years and successive technology generations, these design decisions are very important to the competitiveness of any given product generation. Processor microarchitecture design impacts clock frequency by placing constraints on the number of gates connected back-to-back between pipeline latches. Typically it is possible to improve IPC by introducing a more complex hardware algorithm. However, such improvements in IPC may be offset by an increase the critical delay path from the output of one set of latches into the next by increasing the number of levels of back-to-back logic on the critical delay path. Historically, this has lead to a form of competition between microprocessor designs embracing a "speed demon" approach (focus on higher clock frequency), versus a "brainiac" approach (focus on higher IPC) [Gwe93].

This dissertation describes techniques for increasing the average number of instructions executed per clock cycle without impacting cycle time.

Two trends in contemporary microprocessor design motivate the techniques presented in this dissertation. The first is the growing availablity of single-chip multithreaded hardware. This growth is spurred by an abundance of hardware resources with increasing transistor densities, along with the relative ease (in terms of design complexity and verification costs) of obtaining improved throughput by supporting additional concurrent threads on a single chip. The second, and more fundamental, is a trend towards increasingly significant performance losses due to cache misses and branch mispredictions. Traditionally, performance bottlenecks such as cache misses and branch mispredictions have been addressed by hardware and/or compiler optimization techniques. However, while these approaches have been very effective thus far, they do not completely eliminate some performance degrading events. The techniques explored in this dissertation build on recent efforts that help attack those performance degrading events that traditional techniques are unable to adequately cope with as semiconductor technology continues to scale.

1.2 Background

This section provides a short introduction to the topic of microprocessor design at the microarchitecture level. At the top level of a microprocessor design is the *instruction set architecture* (ISA) interface between hardware and software. The programmer creates software using either a high-level language, which is transformed to machine code using a compiler, or by directly coding instructions at the assembly level. These instructions are executed by the processor and transform the program's inputs to its outputs in some useful way. To ensure software compatibility between successive generations of microprocessors, the ISA is held fixed (with the occasional addition of new instructions). To enhance performance, the hardware organization that implements the instruction set architecture may be varied. This hardware organization is called the microarchitecture of the processor.

The next section provides fundamental background on single-threaded, microarchitectures and can be skipped by the knowledgeable reader. Section 1.2.2 gives an overview of multi-threaded architectures and describes the terminology used in this dissertation.

1.2.1 Fundamental Microarchitecture and Program Behavior

To enhance the impact of the microarchitecture on performance it is important to consider the behavior of "typical" programs. To obtain greater average application performance for a particular semiconductor technology generation, cycle time can be optimized while employing microarchitecture optimizations to extract instruction-level parallelism. *Instruction level parallelism* (ILP) refers to the amount of parallelism among the dynamic sequence of instructions in a single thread of control. While from the programmer's perspective a thread may *appear* to execute sequentially, one instruction at a time, in an order determined by the control flow constructs of the programming language, the hardware microarchitecture may in fact schedule multiple instructions to execute at the same time provided they are independent of each other.

Hardware optimizations used to dynamically extract ILP from software exploit regularities in the behavior of typical programs. What is "typical" tends to evolve over time and hence significant effort is devoted to gathering and refining collections of benchmark programs thought to be representative of programs that will run on future microprocessors. One such example is the SPEC benchmark suite, which has been updated roughly every four years [Sta]. Benchmark suites such as SPEC enable objective comparison of competing computer systems, and encourage microprocessor designers to optimize the performance for real programs. For many microarchitecture techniques used in modern microprocessors, programs *can* be written that defeat the optimization and actually slow down when the technique is employed. The key is that the behavior exhibited by such programs is *unlikely* in typical programs. In typical programs found in SPEC and other bencharks it has been observed that parallelism among instructions from different basic blocks, correlated branch outcomes, and locality in the sequence of memory addresses generated by the program are common. It has been shown that these particular features can be exploited by hardware structures that adaptively learn the relevant structure of a program and use this information to significantly improve performance. Below we briefly explore the general ways in which these and other features of program behavior are used in existing microprocessor designs. The techniques described later in this dissertation focus on exploiting regularity in the control flow paths executed by a program at a level of granularity not exploited by hardware-only microarchitecture techniques.

Figure 1.1 shows a block diagram of the microarchitecture of a typical current generation out-of-order superscalar microprocessor. Instruction execution is *pipelined* so that an instruction

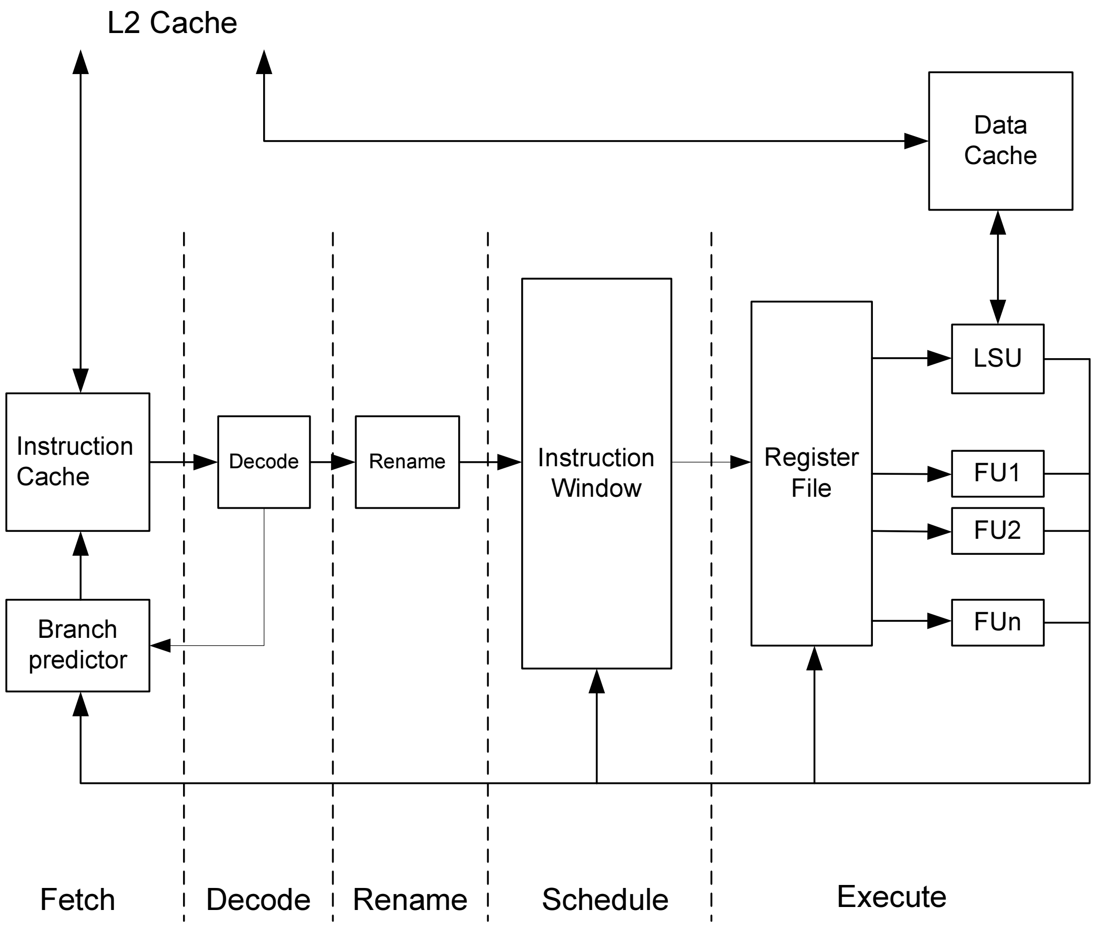


Figure 1.1: Superscalar processor

may begin processing before the immediately preceding instructions in the instruction stream have finished processing. Each clock cycle, the fetch stage of the processor brings a new group of instructions into the pipeline from memory. The fetch stage includes a *branch predictor* and *branch target buffer* (BTB) that together attempt to guess the direction ("taken", or "not-taken") and target address of branches as they first enter the pipeline.

Once a set of instructions have been fetched, they are decoded to determine their impact on computation. Then the operands specified in the instruction are renamed and some information about the instruction is placed in an instruction reorder buffer.

Register renaming is a mechanism by which each definition of a programmer-visible architectural register (i.e., those registers specified in the ISA) that is encountered in the dynamic instruction stream is given a unique physical storage location. Renaming serves two important purposes: First, it exposes parallelism obscured when the limited number of registers available in a given instruction set architecture forces software to reuse registers. Without register renaming, the process of issuing instructions into execution units must wait whenever an instruction defines a register that may be used by an earlier instruction that has not completed execution. Second, register renaming provides a unique tag that may be used both to signal that the result of a computation is ready to be used by those instructions that depend upon it, and to route that result to the hardware units responsible for executing dependent instructions.

The *reorder buffer* is a FIFO queue used to support precise exceptions and to recover from branch mispredictions.

After register renaming, instructions are placed into an instruction scheduling window where they wait for their source operands to become ready so that they may be selected for execution (in some microarchitectures, the scheduling window also serves as the reorder-buffer). After instructions have completed execution they wait in the instruction window until all prior instructions have completed execution. Finally, instructions are retired from the processor in the same order they entered into the pipeline. This in-order retirement ensures both that the results and side-effects of speculatively executed instructions following a mispredicted branch are discarded without changing the architected program state, and also that the processor can stop when an exception is detected and later restart from the same location after the operating system has processed the exception as though no interruption had occurred. In-order retirement of instructions eliminates the need to stop and save the precise details of the entire pipeline state so that it can be restored later.

Pipelining improves performance by reducing cycle time while exploiting predictability in the control flow of a program and independence between the various stages of instruction processing to divide up the work of executing a single instruction over many cycles and partially overlap an instruction's execution with that of other instructions. For example, provided all instructions are encoded using the same number of bits, the fetching of an instruction can proceed in parallel with the decoding of the previous instruction. As long as the branch predictor is correct, the work done in parallel by the fetch and decode stages is useful. However, if the branch predictor makes a mistake, the fetched instructions are discarded and the opportunity to exploit this parallelism is lost.

The particular pipeline organization in Figure 1.1 is specialized to typical application behavior in several other respects. For example, the use of an instruction window and dynamic instruction scheduling improves performance by exploiting the parallelism typically found between nearby instructions in the dynamic instruction sequence of a program. This parallelism is an inherent property of the program itself, and the processor's ability to extract it is also related to the predictability of program control flow. Typically, in non-numeric applications, it is necessary to examine instructions from multiple basic blocks at the same time to find independent instructions to schedule for simultaneous execution. A *basic block* is a sequence of instructions that is executed as long as the first instruction in the sequence is executed (in other words, a basic block starts with an instruction that is the target of a branch or with the "fall-through" instruction immediately following a "not-taken" branch; it ends with the next branch instruction or the next occurance of an instruction that is immediately before the target instruction of some branch). Without executing a branch that terminates a basic block, it is not known with complete certainty what the subsequent basic block will be. Again, this control dependence is tackled by employing a branch predictor. Branch prediction relies on the fact that for typical programs, the outcome (direction and target address) of branch instructions tends to be correlated with the earlier behavior of the same branch and/or the behavior of other branches.

The processor interfaces to memory through caches that keep frequently accessed instructions and data in a small storage location with fast access times. The first-level cache is backed up by a second level, and sometimes third level of successively slower but larger caches that also maintain copies of instructions and data that are accessed less frequently than those resident in the first-level cache. Load and store instructions are placed in a structure called the load/store queue (also known as a memory order buffer) that schedules access to the memory hierarchy and ensures memory requests satisfy data dependencies through memory and the memory consistency model. Typically, first-level caches are *indexed* using the virtual address and *tagged* using the physical address provided by a *translation lookaside buffer* (TLB). Upon a first-level data cache miss, a request is sent to the next higher level of the memory hierarchy and, if the request is the result of a load instruction, any instruction that depends upon the value produced by the load must wait for this access to complete. The front-end of the processor accesses instructions in the instruction cache after consulting the branch predictor and BTB to index the cache, and the instruction TLB to perform the tag comparison to detect a hit or miss condition. Upon an instruction cache miss, the fetch unit stalls and instructions are no longer fed into the beginning of the pipeline while instructions already in the pipeline continue processing.

Caches exploit locality in the stream of memory addresses generated by typical programs. This locality exists because programs typically perform many operations on data before they are finished with it, and many algorithms involve looping through a sequence of instructions

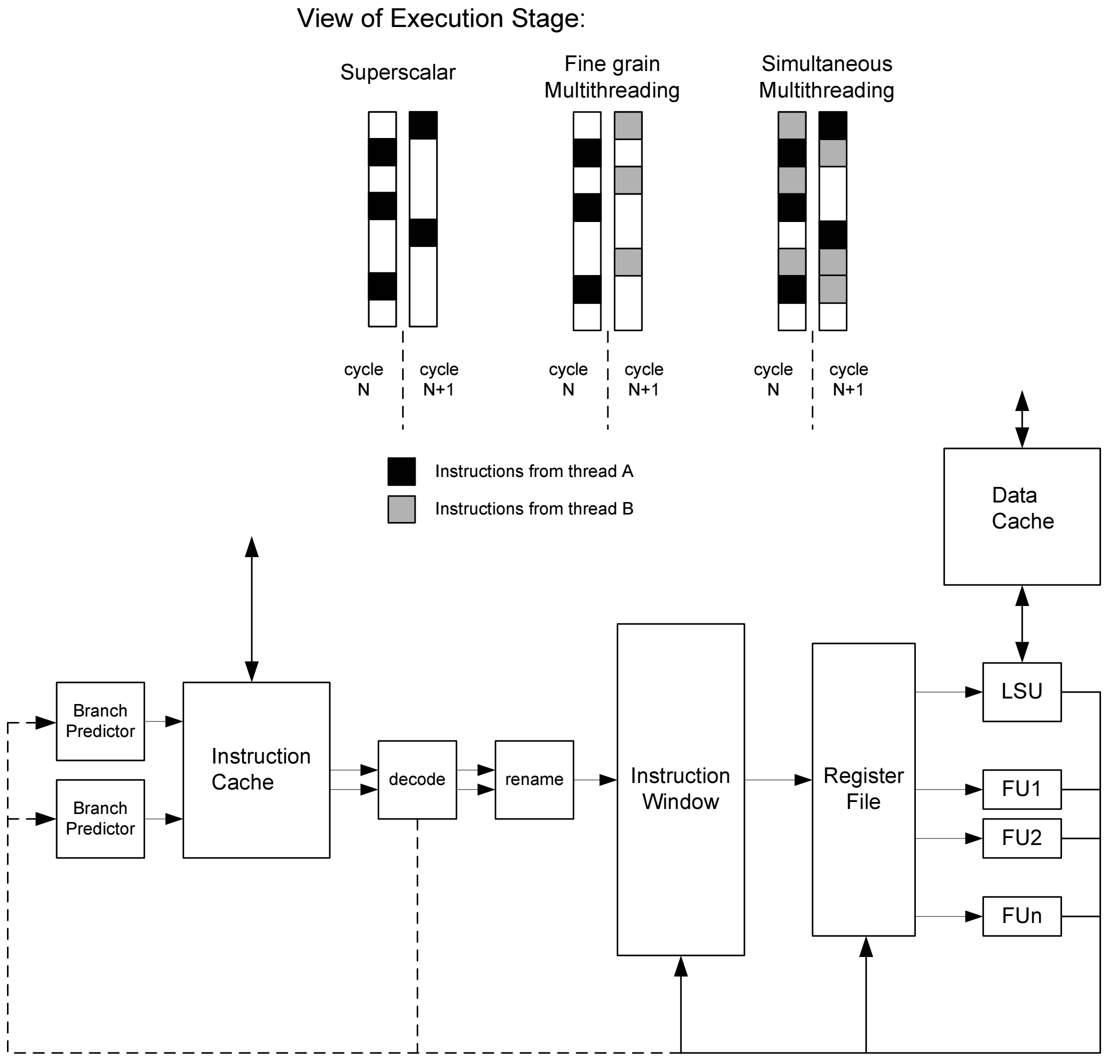


Figure 1.2: Superscalar processor with simultaneous multithreading support

repeatedly. However data and instruction cache misses still persist despite this locality. To improve the effectiveness of the memory hierarchy, modern processors employ a technique called prefetching. *Prefetching* is a process in which instructions or data are brought into the cache before the processor actually needs them.

1.2.2 Multithreading Microarchitectures and Program Behavior

The basic superscalar design in Figure 1.1 can be extended to allow multiple threads of control, or instruction streams, to share most of the pipeline's hardware resources. This is called *simultaneous multithreading* (SMT) and is illustrated in Figure 1.2. SMT has been implemented on Intel's Pentium 4 microprocessor under the brand name of 'Hyper Threading', and IBM has announced SMT support in the Power 5, which IBM has disclosed is also a single chip multiprocessor (CMP)—i.e., has multiple microprocessor cores on a single silicon die. The Alpha EV8 microprocessor (development of which was cancelled in 2001) was also designed with an SMT hardware organization [SFKS02]. In an SMT processor the hardware thread contexts on a single processor core typically share the first level of the cache hierarchy. If the threads running on these SMT thread contexts are unrelated, then instructions and data compete for space within the cache hierarchy, potentially leading to higher cache miss rates for both threads. The effect on job throughput is offset by the positive benefit of multiplexing work from both threads on the execution core. In particular, as the issue width of microprocessors are increasing at the same time that memory latencies are scaling up, it is getting more and more difficult to find ready instructions to schedule in a given cycle from a single thread despite the fact that instruction window sizes are also increasing. By allowing ready instructions from independent threads of computation to share execution resources these additional issue slots can be used. In the top portion of Figure 1.2 (labelled "View of Execution Stage"), the issuing of instructions to execution units each cycle is contrasted with normal superscalar, and fine grain multithreading (such as implemented on the Tera Computer System [ACC$^+$90]). For example in Figure 1.2 the superscalar processor issues three instructions from thread 'A' on clock cycle 'N' and two instructions from thread 'A' on clock cycle 'N+1'. Fine grain multithreading increases throughput by allowing instructions from a different thread to be scheduled each clock cycle. In the example in Figure 1.2 on clock cycle 'N+1' the processor supporting fine grain multithreading issues three instructions from thread 'B' instead of the two instructions from thread 'A' thus increasing instruction throughput per cycle by one half over the superscalar design (for an improvement of 20%). Simultaneous multithreading allows a potentially far greater sharing of execution resources within a given cycle. In the example in Figure 1.2, the processor supporting SMT issues three instructions from thread 'A' and three instructions from thread 'B' on the first cycle and roughly doubles instruction throughput compared to the superscalar processor.

SMT allows hardware resources, such as instruction scheduling window entries and issue ports to be allocated dynamically to the thread that can make the best use of them, rather than statically partitioning resources among thread contexts. Of the two structures mentioned (instruction scheduling window entries and issue ports) the latter has been shown to be the more important hardware resource to dynamically partition for sharing among application threads [RR03]. On the other hand, the sharing of caches and branch predictor table entries can lead to destructive interference between concurrently running threads. One of the key hardware attributes enabling helper threads such as those studied in this dissertation to improve performance is the cache interaction between concurrently running threads on an SMT processor. The benefit of sharing space in the cache is that the helper thread can be designed to bring instructions and/or data into the caches just before the main thread requires them.

It has been observed [ZS00] that a small fraction of all static loads and branches are responsible for the majority of data cache misses and branch misprediction events that foil existing hardware techniques such as stride prefetching [PK94] and branch history based branch prediction mechanisms [Smi81, LS84, MH86, YP91]. The load address or branch outcome stream of these instructions does not show sufficient locality that predictions can be made based merely upon past behavior. Load instructions that exhibit this behavior have been described as *delinquent loads* [CWT$^+$01].

Recently, proposals have been put forth to use multithreading resources to reduce the impact of such performance bottlenecks [DS98, CSK$^+$99]. One of the earliest proposals included simulating larger branch history based predictors in software than can be built in hardware economically, augmented with sophisticated mechanisms for integrating the prediction results into the conventional superscalar pipeline [CSK$^+$99]. Later studies suggested the alternative approach of isolating the subset of instructions that will produce the hard to predict outcome, called the computation *slice* [Wei81], and executing this slice of instructions early enough that the results can be used to eliminate or reduce the impact of the associated performance degrading events. Studies of the slices leading to branch mispredictions and cache misses have shown that these are both limited in size [ZS00], and contain significant locality in the sequence of operations contained within the slice from one instance to the next [AMC01]. Subsequent studies have demonstrated the effectiveness of these techniques under realistic implementation constraints [MPB01, CWT$^+$01, CTWS01, ZS01, Luk01, APD01]. Recent work has begun to consider optimization of data prefetch helper threads [RS02, LWW$^+$02] and the implementation of helper threads on real hardware systems [KLW$^+$04, WCW$^+$04].

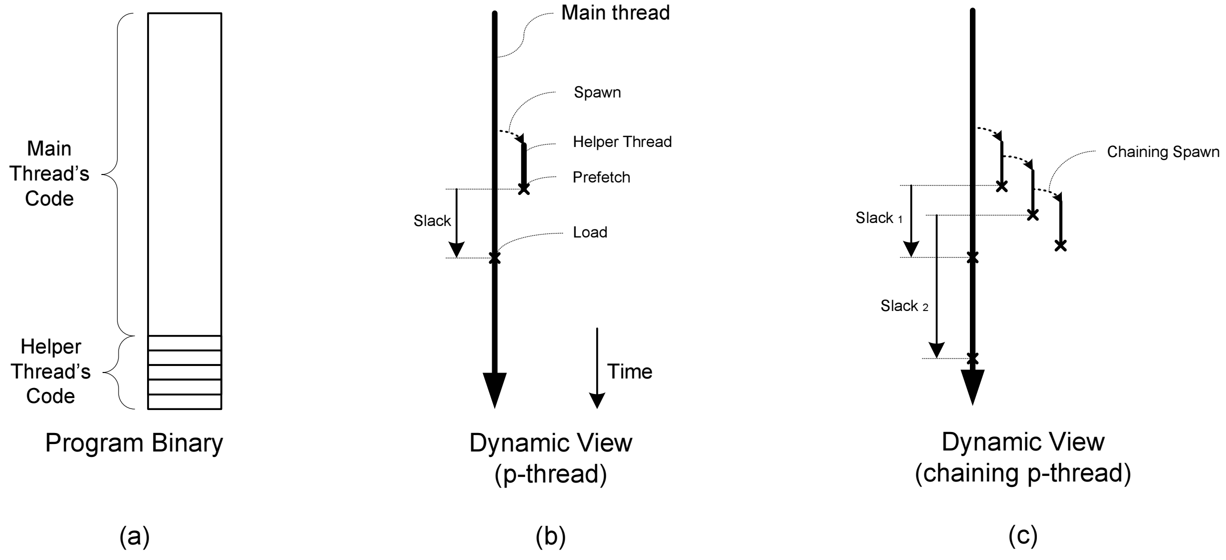
1.2.3 Helper Threads

A programmer seeking to reduce application runtime by exploiting multiprocessing hardware can manually partition an application into threads when developing the source code. In some cases this partitioning actually eases the software design process itself, for example by decoupling development of unrelated tasks, but often the goal is to improve performance of a single application. In the latter case, a careful consideration of dependencies within the program is required. When manual partitioning is successful it can lead to dramatic improvements in performance. As the process of parallelizing software and the required dependence analysis is well understood for certain application domains—primarily numerically intensive applications—support for automatic parallelization has been incorporated into some modern compilers.

However, there are many important applications for which it is difficult to achieve significant performance improvements by statically partitioning the computation into threads using existing techniques. To tackle the challenges posed by such applications, there has been extensive research on dynamic hardware-based program parallelization that has shown significant potential [SBV95, SM98, KT98, MGT98, AD98, MG02]. With the possible exception of SUN Microsystem's MAJC processor [TCC$^+$00, Hal99], commercial microprocessors do not yet employ such techniques.

In contrast to the parallelization techniques described above, a *helper thread* performs computation that is run simultaneously with the application's *main* thread. The goal of the helper threads explored so far is to eliminate microarchitectural performance bottlenecks such as cache misses and branch mispredictions in the main thread.

Figure 1.3 illustrates two types of helper threads that have been investigated previously [CWT$^+$01, ZS01, LWW$^+$02]. Figure 1.3(a) shows a static view of the program and highlights the location of helper thread instructions relative to the main thread's instructions in the binary image. When the program executes, helper threads are invoked if the main thread reaches a spawn instruction and there are thread contexts available—see Figure 1.3(b). When a helper thread is spawned the registers in the spare thread context are initialized to the corresponding values in the main thread. The spawn instruction then initializes the program counter of the spare thread context and activates it causing the helper thread to run. In this dissertation the hardware model includes a native spawn instruction that executes in a single cycle. While dedicated hardware support for such spawn instructions has not been announced in any commercial processor thus far, recent studies have shown that microcode implementations of closely related spawn instruction



Part (a) illustrates a modified program in which helper thread code has been attached to the program binary. In part (b) the thick arrow represents the dynamic instruction stream of the main thread and the shorter thick line represents the dynamic instruction stream of a simple helper thread that generates a single data prefetch. Part (c) illustrates a 'chaining' helper thread that spawns another helper thread in addition to generating a data prefetch.

Figure 1.3: Data prefetch helper threads.

functionality incurring multiple clock cycles of latency [KLW$^+$04, WCW$^+$04], combined with appropriate helper threads, can yield significant benefit on real applications even on systems composed of single core processors running a multithreaded workload [WCW$^+$04]. Referring again to Figure 1.3, the helper thread exits when it executes a special thread exit instruction.

As illustrated in Figure 1.3(c), a *chaining* helper thread contains a spawn instruction that triggers the creation of a new helper thread. Typically, the spawn instruction is assumed to trigger a new invocation of the same thread code containing the spawn. This form of chaining helper thread has been shown to be effective for tackling data cache misses inside of loops. Chaining improves performance by increasing both the number of helper thread instructions executed in parallel (ILP) and the number of outstanding prefetches accessing the memory system (MLP). On in-order machines, it may also help to prevent a cache miss triggered in one helper thread invocation from stalling the execution of a future helper thread invocation.

While each helper thread instance runs for a short duration of time, there will be many helper thread instances during the runtime of an application and hence helper threads may consume valuable execution resources that potentially could have been used for other purposes. Furthermore, the effectiveness of each helper thread is based on how well its computation anticipates future performance degrading events in the main thread.

1.2.4 Program Behavior Exploited by Helper Threads

There are several factors that govern the performance impact of helper threads. These include the microarchitecture behavior of the helper thread's code upon the helper thread's own execution (i.e., "ignoring" the existence of the main thread), the resources available to run helper threads, and the relationship of the helper thread's behavior relative to the behavior of the main thread.

An important property of the helper thread's execution is the number of instructions that must be executed to generate each prefetch. An important resource is the number of spare thread contexts available to run helper threads. However, it is the relationship of the helper thread's behavior relative to the behavior of the main thread that determines whether a prefetch or branch prediction is timely and necessary. This relationship can be quantified in several ways which will be described in Chapter 2.

The main contribution of this dissertation is a novel framework for modeling the impact of speculative threads used to improve microprocessor performance. While prior work has examined the impact of data dependencies on speculative threads [RS02, LWW$^+$02, ZCSM02, ZCSM04], the novel aspect of this new framework is its focus on statistically quantifying the impact of control flow on the impact of speculative threads. While knowledge of data dependencies is critical to identifying parallel sub-computation, state-of-the-art microprocessors follow a single path of control flow that inherently limits the rate at which instructions can pass through a finite window, dataflow-driven, out-of-order execution core. Hence, at a higher level of granularity than the number of instructions that can fit within a modern processor's scheduling window, control flow determines how long it takes for program execution to get from one place to another, as well as how likely that event is to take place. This scale also defines the typical scale, in terms of numbers of dynamic instructions, of speculative threads. Thus, to model the impact of speculative threads it is beneficial to study ways of modeling control flow variation.

As already noted, the correlation between individual branch outcomes—a form of control flow behavior—is already exploited by modern branch predictors. The aspects of control flow this dissertation focuses on extend beyond individual branch outcomes to the control flow structure of the program. In particular, this work considers the statistics of program paths weighted by their frequency of occurrence and uses this information to model the impact of speculative threads. Prior work by Marcuello and Gonzalez noted the potential to use one such aspect—the notion of reaching probability—for improving the performance of speculative multithreading [MG02]. This dissertation takes this notion as a starting point, and advances beyond it by developing a unified theoretical framework incorporating many other important statistical quantities, shows how these quantities can be evaluated efficiently using path expressions, and then applies this novel framework to the optimization of helper threads.

To evaluate the overall framework, it is applied to optimizing the selection of two types of helper threads. The framework proposed in this paper helps estimate helper thread performance impact in a manner compatible with the generation of helper thread code within a standard profile driven optimizing compiler.

While the framework naturally fits in the context of profile-driven compiler optimization, the notion of quantifying the impact of speculative threads to optimize their selection is not inherently limited to this setting. For instance, it may be feasible to implement this type of analysis and optimization methodology in hardware or in a dynamic compilation framework within a runtime system. The advantage of such a system is that it could be used to adapt helper threads to changes in program behavior.

1.3 Dissertation Summary

This section gives an overview of this dissertation, outlining its goals, methodology, contributions and organization.

1.3.1 Research Goals

The primary objective of this research was the development and detailed evaluation of novel techniques for improving the performance impact of speculative threads. The impact of speculative threads can be enhanced by optimizing the selection and composition of the speculative threads themselves. To do so it is desirable to have an accurate framework for quickly estimating—without the use of detailed simulation—the impact of tradeoffs in the construction of speculative threads. Such tradeoffs exist, for example, because there are a limited number of thread contexts that must be shared by application main threads and speculative threads, and speculative threads consume execution resources that could also be applied to directly execute instructions from one of the main threads.

A secondary objective was the development and detailed evaluation of novel implementation techniques and hardware support mechanisms both to facilitate the optimization process and to enhance the impact of the resulting speculative threads. For practical reasons the scope of inquiry is limited to speculative helper threads for instruction and data prefetch.

1.3.2 Methodology

The simulation results presented in Chapters 2 to 4 were collected using a simulator based upon SMTSIM [Tul96] extended to run the Intel Itanium® instruction set. Data in these chapters was collected while the author was on an internship with the Microarchitecture Research Lab of Intel Corporation in Santa Clara. This SMTSIM-based simulator is a highly detailed execution driven timing simulator that models an in-order Itanium processor with similar hardware organization to the Itanium 2 from Intel Corporation. Simultaneous multithreading is modeled by assuming instructions from multiple thread contexts can issue in-order to a set of shared function units. Instruction fetching is restricted to bring in instructions from at most two threads in any cycle, independent of how many thread contexts are supported. Memory latencies and pipeline depths are increased in the simulator model to be representative of a future generation processor operating at a higher clock frequency. Additional details of this simulation infrastructure are described later on in the dissertation along with descriptions of the techniques developed in this dissertation.

The simulation results presented in Chapter 5 are based upon a modified version of SimpleScalar [BA97] that supports SMT and some of the new hardware mechanisms discussed later in this dissertation. As well, a compiler pass was developed that couples an implementation of the modeling framework in Chapter 2 with the CodeSurfer program slicing tool (licensed for academic use) via the latter's Win32 DLL API and via textfiles with the SUIF compiler infrastructure [HAA^{+}96]. The output produced by this combination is ANSI C that is then compiled using the SimpleScalar compiler for the PISA instruction set. Additional details are described in Chapter 5.

1.3.3 Contributions

This dissertation makes the following contributions:

1. It proposes and evaluates a novel control flow modeling framework that accurately predicts the following statistical quantities between two static instructions within an application thread: (a) the reaching probability, (b) the posteriori probability, (c) the mean path length, (d) the path length variance, and (e) the path instruction footprint.

2. It evaluates this framework on the task of optimizing the use of helper threads for prescient instruction prefetch. A simple "greedy" algorithm is employed along with a heuristic function that quantifies the expected impact of a particular helper thread selection on performance.

3. It proposes and evaluates two implementation techniques for realistically enabling instruction prefetch helper threads: *Direct Pre-execution* and *Finite State Machine Recall*.

4. It proposes and evaluates a hardware mechanism, the *YAT-bit*, for filtering the instructions executed during Direct Pre-execution.

5. It extends and evaluates a proposal for supporting store instructions in speculative threads called the *safe-store*.

6. It proposes and evaluates a way of using counted instruction prefetch operations to reduce the resource consumption of helper threads.

7. Finally, it proposes extensions to the framework to enable it to optimize the use of helper threads for data prefetch.

1.3.4 Organization

The rest of this thesis is organized as follows: Chapter 2 introduces a rigorous framework for modeling speculative threads. Chapter 3 describes how to apply this framework to optimize prescient instruction prefetch helper threads. Chapter 4 describes hardware support mechanisms and implementation techniques for implementing prescient instruction prefetch helper threads. Chapter 5 describes the application of the modeling framework to the optimization of data prefetch helper threads. Chapter 6 describes related work, and Chapter 7 concludes. Appendix A documents two of the longer derivations that underlie the contributions made in Chapter 2.

Chapter 2

Modeling Framework

In this chapter we describe the modeling framework developed for this dissertation. Building on this foundation, an optimization algorithm for generating instruction prefetch helper threads is presented in Chapter 3, and similarly, an optimization algorithm for generating data prefetch helper threads is presented in Chapter 5. This framework is designed to model short term variations in control flow that exist within a program phase, over periods longer than represented by the number of instructions in-flight within a microprocessor. A high-level view of the framework is shown in Figure 2.1. This framework is applicable within a profile-guided optimizing compiler that augments an application with helper thread code and enables the application to launch these helper threads on a microprocessor with appropriate hardware support.

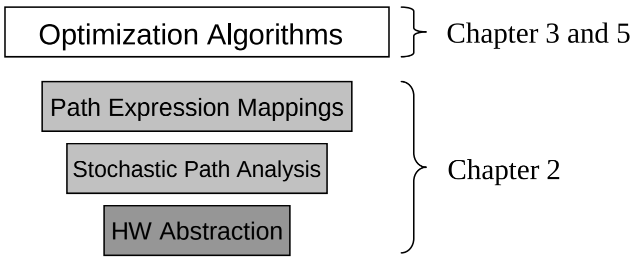
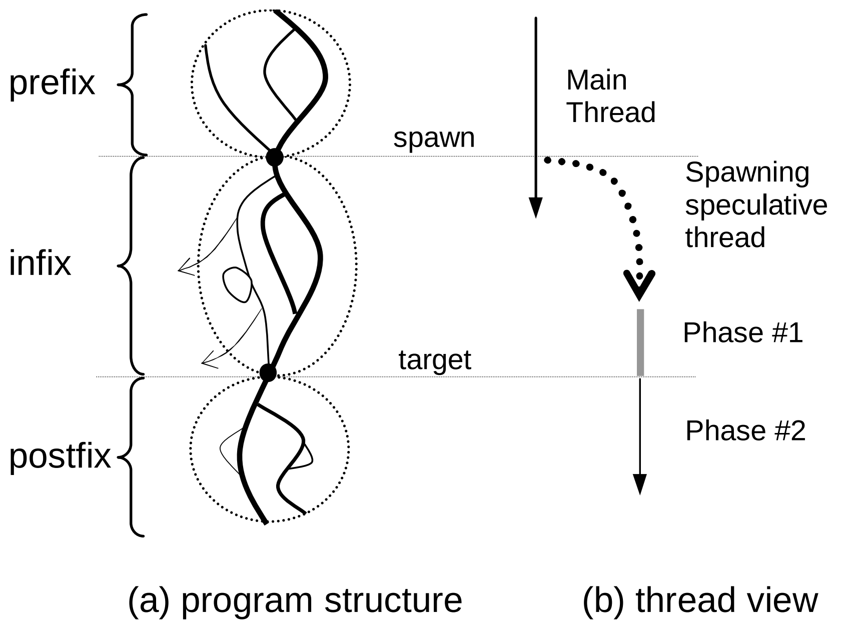


Figure 2.1: Modeling and Optimization Framework

The rest of this chapter is organized as follows: In Section 2.1 we define a generic speculative thread, and then consider a specific example from which we abstract key properties impacting the performance of speculative threads. This abstraction captures important properties of several types of helper thread as well as some important aspects of speculative multithreading. Then



Part (a) shows a control-flow graph with spawn and target points highlighted. It is known *a priori* that having the speculative thread mimic the behavior of the main thread in the region labeled postfix is beneficial to performance. Part (b) illustrates two phases of speculative thread execution: Phase #1 live-in precomputation; Phase #2 postfix precomputation and generation some form of "results" for main thread. Note that distance along the vertical axis in Part (b) corresponds to the number of dynamic instructions executed by the main thread (i.e., it does not directly measure time).

Figure 2.2: Speculative Threads.

we consider several metrics for quantifying these properties. In Section 2.2 we show how to compute these metrics by considering the set of all control flow paths between two instructions within a program. In Section 2.3 we provide some experimental evidence showing how accurately the framework models the properties. Finally, Section 2.4 summarizes the contributions of this chapter.

2.1 Generic Properties of Speculative Threads

The objective of a speculative thread modeled by the framework set forth in this dissertation is to "mimic" the behavior of the main thread in some, as yet unspecified way (the examples that will be considered later relate to instruction and data prefetch). For concreteness, we consider a generic speculative thread to consist of five components: a prefix region in the main thread, a spawn point in the main thread, an infix precomputation slice, a target point, and a postfix region. These are illustrated in Figure 2.2, and described below.

1. The *spawn point* indicates the location within the main thread at which the speculative thread is launched. The spawn point may be encoded in the program binary as a special "spawn instruction", or it may simply be a program address recognized by the hardware using some other mechanism. Regardless of how it is implemented, the action taken upon reaching the spawn point is to attempt to start a new speculative thread.

2. The *target point* indicates the starting location within the main thread that the speculative thread mimics.

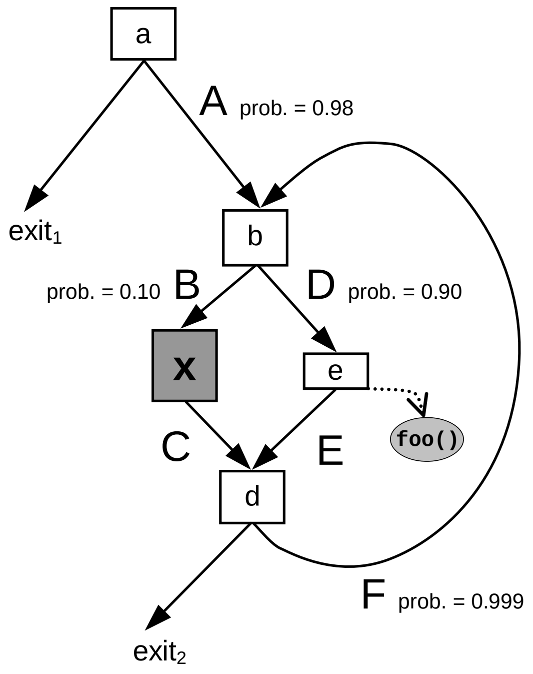
3. The *postfix region* are the instructions following the target that the speculative thread mimics.

4. The *infix precomputation slice* are the instructions between the spawn and the target that must be executed by the speculative thread to accurately mimic the behavior of the inputs to the postfix region. It may be possible to partially predict the result of the infix precomputation slice using hardware mechanisms such as value prediction [LS96].

5. The *prefix region* are those instructions preceding the spawn-point in the main thread. The instruction sequence leading up to the spawn-point may be used to determine whether to spawn a speculative thread.

The benefit of launching the speculative thread depends upon the difference in behavior of the speculative thread compared with the main thread. For example, at the spawn point it may not be certain that the main thread will subsequently execute the target instruction and postfix region. In the event the target and postfix region are "skipped" there may be little benefit in launching the speculative thread as the speculative thread is supposed to do work on behalf of the main thread by mimicking the latter's behavior. Below we consider a specific example in which the speculative thread prefetches instructions in the postfix region on behalf of the main thread—a type of helper thread called *prescient instruction prefetch* in Chapter 3. In the following example, the speculative thread optimization problem is described in terms of the problem of selecting the location of the spawn and target points.

***Example:* Spawn-Target Selection Tradeoffs**



Control flow graph fragment with edge profile information.
Block e calls subroutine `foo()`.

Figure 2.3: Example: Optimizing an Instruction Prefetch Helper Thread.

Figure 2.3 depicts a control-flow graph fragment. Nodes represent basic blocks, and edges represent potential control flow transfer between two basic blocks. The shaded block labeled x is known from cache profiling to suffer many instruction cache misses. Each edge is labeled with the probability that the main program will make the respective control flow transfer unless the value is exactly one. Block e calls subroutine `foo()`. The questions of interest are: (1) which locations are the best places to spawn a helper thread to prefetch x, and (2) what should the target be?

Note that, starting from any location, the program is very likely to reach x, because every iteration the probability of exiting the loop (i.e., from block d) is much smaller than the probability of transitioning to x (i.e., from node b). Even though any choice of spawn is roughly as "good" as any other in this particular sense, as we show next, not all spawn points are necessarily as effective as others.

For instance, consider the impact the size of subroutine `foo()` has on spawn-target pair selection. If `foo()`, together with blocks b, d, e, and x fit inside the instruction cache, then initiating a prefetch of block x starting at block a is effective, because the loop will likely iterate several times before the main thread branches from `b` to access `x`. On the other hand, if `foo()` together with its callees (if any) require more instruction storage than the cache capacity, then there is no point in spawning a prefetch thread to target `x` from *any* block. The reason is that instructions prefetched by the helper thread will almost certainly be evicted by an intervening call to `foo()` before the main thread can access them due to the low probability of branch `b` transitioning to `x`. A better target in this situation would be block `b`, because evaluating the branch at the end of block `b` would cause the helper thread to prefetch `x` only when it is about to be executed by the main thread. A good set of spawn points targeting `b` in this case might include the beginning of blocks `a`, and `d`.

In the latter case, if we can choose only one spawn location from `a` or `d`, due to resource limitations, the more profitable choice is `d`, because `a` would only cover misses at `x` in the event control goes directly from `a` to `b` to `x`, while `d` would cover the cache misses at `x` in all other cases.

As this example demonstrates, it is important to accurately predict the run time properties of helper threads when selecting spawn-target pairs. Returning to the perspective of a generic speculative thread such as that illustrated in Figure 2.2 the performance impact of a generic speculative thread can be optimized along the following three dimensions:

speculativeness	When the main thread reaches the spawn point it is not certain whether the target and postfix region will be reached. For a given target point the degree of speculativeness can be varied by selecting the spawn point carefully.
coverage	For a given target point, the choice of spawn point may impact the number of times a dynamic instance of the target point is preceded by an instance of the spawn point. An instance of the target is covered by an instance of the spawn if the spawn occurs before the target.
timeliness	The results computed by the speculative thread should be produced early enough that the main thread can use them when needed, but not so early that they tie up finite storage structures used to store the results, preventing these storage structures from being used to hold other important values.

To tackle the challenges to effective use of speculative threads illustrated in the above example, we propose optimizing different forms of speculative threads along the dimensions enumerated above. Section 2.2 formulates an analytical framework that aids in rigorously quantifying the speculativeness, coverage, and timeliness of speculative threads. This framework is applicable to forms of thread speculation other than the one illustrated by the example above. For instance, in Chapter 5 we show how simple data prefetch helper threads can be optimized using this framework. The postfix region for these helper threads consists of only the target delinquent load instruction. In Chapter 5 we also show how the framework can be used to optimize the selection of data prefetch helper threads.

2.2 Analytical Framework

In this section an analytical framework—represented by the bottom three layers of Figure 2.1—is introduced that models program behavior as a Markov chain. This framework provides a means of computing a set of key statistical quantities characterizing the tradeoffs illustrated by the example in the previous section. In particular, a novel technique for transforming the computation of these statistical quantities to use Tarjan's path expression algorithm [Tar81b] is described, thus leveraging the latter's efficiency. The resulting framework is used as the foundation for a simple spawn-target pair selection algorithm for prescient instruction prefetch described in Section 3.2.2 (see Figure 3.4), and for data prefetch helper threads in Chapter 5.

2.2.1 A Statistical Model of Program Execution

For a control flow graph, where nodes represent basic blocks, and edges represent transitions between basic blocks, we model the intra-procedural program execution as a discrete Markov chain [HD77]. A Markov chain is defined by a set of states and transitions. The basic blocks in a procedure represent the states of the Markov chain, and transitions are defined by the probabilities of branch outcomes in the control flow graph[1]. These probabilities can be gleaned from traditional edge profiling [CMH91], which measures the frequency that one block flows to another. For inter-procedural control flow, we incorporate the effect of procedure calls (when necessary) by computing summaries for subroutines, or sets of mutually recursive functions. This is equivalent to restricting transitions into and out of procedures so that a callee must return to its caller. As it is based upon using traditional edge profile data, this model ignores correlation

[1]In Chapter 5, the model is extended to incorporate pairs of branch outcomes. In a similar fashion it is possible to extend the framework to incorporate even longer histories of branch outcomes.

between branch outcomes. Given the above interpretations, the graph in Figure 2.3 (ignoring the subroutine call to foo()) represents a Markov chain of the form just described.

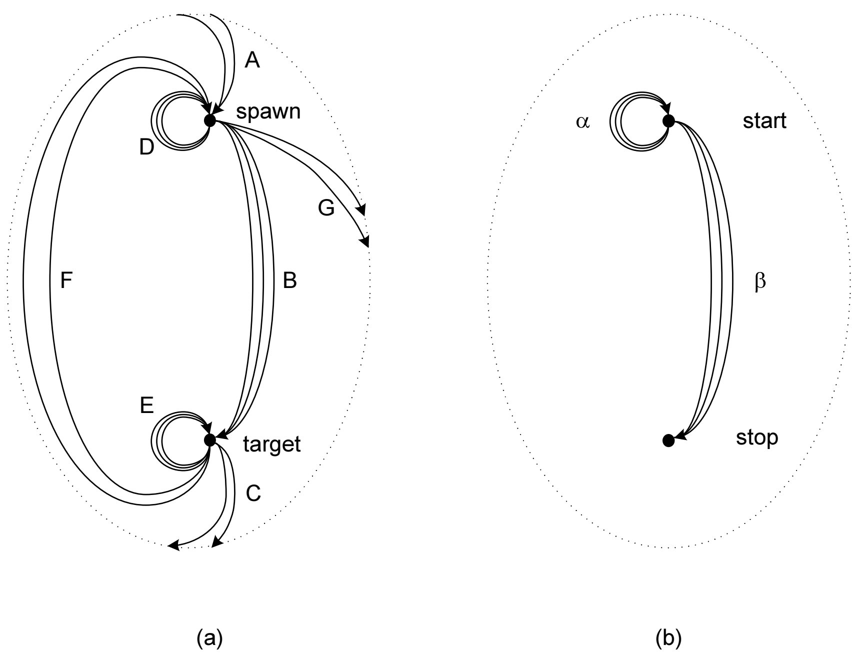
To model the effects of instruction cache accesses, we assume a two-level memory hierarchy composed of a finite-sized fully associative instruction cache with LRU replacement, and an infinite sized main memory so that all misses are either cold misses or capacity misses. Note that the control-flow path of program execution entirely determines the contents of this cache independent of the program's static code layout; hence, by considering the probability of taking each possible path through the program it is possible to determine the probability with which a given instruction resides in the cache at any given point in the program execution.

2.2.2 Computing Statistics via Path Expressions

In this section we describe the particular statistical quantities provided by the modeling framework and motivate their inclusion in the framework by referring the reader to the example and associated dimensions of optimization described in Section 2.1. The statistical quantities are the reaching probability, posteriori probability, expected path length and variance, and expected path footprint. Subsequently, a technique for transforming the evaluation of these quantities to the classic path expression problem (also known as an algebraic path problem) is introduced [AMC$^+$02, AMC$^+$03, AWSH], which can in turn be solved efficiently using Tarjan's path expression algorithm [Tar81a, Tar81b].

The statistical quantities the framework provides are computed over sets of execution paths that start at one instruction and end at another. Figure 2.4(a) motivates this approach by considering all possible execution paths between a particular spawn and target point selected within a particular program region (represented by the dotted oval). Comparing against Figure 2.2, the prefix region is contained in the set of paths A, D, and F; the infix region is entirely represented by the set of paths B, and D; and the postfix region is contained within the set of paths C, E, and F. Figure 2.4(b) shows just the subset of paths between the two instructions, which correspond to the infix region (i.e., the set of paths B, and D). This is the set of paths over which the statistical quantities considered below are defined.

Given a starting basic block x and a sequence of branch outcomes, the unique *path length* between the starting block x and end block y is the number of instructions executed along the associated path through the program. Given the underlying Markov model of control flow described above defined with appropriate probabilities obtained from profiling, the path length



Part (a) shows, abstractly, the paths between a spawn and associated target point in a program. Solid lines (ending in arrows) represent sequences of basic blocks visited from start point to end point. The dotted oval indicates a program 'region' such as a procedure. Part (b) shows the paths that begin at 'start' and end at 'stop' but include only one instance of 'stop'—statistical quantities are defined over the set of all such paths and evaluated using path expressions and the mappings in Table 2.1.

Figure 2.4: Abstraction of Program Execution Paths

is a random variable that takes on a distribution of values resembling that seen by the execution paths starting at x and ending at y seen by the actual program when it operates under similar conditions to that experienced during profiling. Determining the mean and variance of this path length distribution enables the estimation of the probability that the path length is within a given range. This distribution is important because it is closely related to the timeliness of the work performed by the speculative thread (i.e., instruction prefetches in the example). (An important factor for computing timeliness, taken into account later, is the average CPI of instructions executed along these paths.)

Refering again to the example: To avoid selecting spawn points that never perform useful instruction prefetching because the instructions they prefetch are either evicted before they are needed, or are likely to reside in the cache already, the concept of a path's instruction cache footprint is useful. The instruction cache footprint is defined as the capacity required to store all the instructions along a given path assuming full associativity. The *expected path footprint* between two points is determined by computing the average instruction cache footprint between two points in the program. This quantity, which is somewhat specific to instruction prefetching, also relates to the timeliness of the work performed by the speculative thread in the case of instruction prefetch helper threads.

The *reaching probability*, $RP(x, y)$, between basic blocks x and y is defined as the probability that y will be encountered at some time in the future given the processor is at x. In prior work [MG02], the point y is said to be *control quasi-independent* of the point x if the reaching probability from x to y is above a given threshold (for example 95%). The reaching probability helps quantify just how *speculative* the speculative thread is when it is spawned.

Similarly, the *posteriori probability* is defined as the probability of having previously visited state x since the last occurrence of y or the start of program execution, given the current state is y. This quantity helps quantify the amount of *coverage* the speculative thread provides.

Path Expressions

A path expression [Tar81a] is simply a regular expression summarizing all paths between two points in a graph. For example, in Figure 2.3 the set of all paths from block a to block x, start by following edge A, then going around the loop any number of iterations along either control path inside the loop, before finally taking edge B. This can be summarized by the path expression:

$$\mathsf{P(a,x)} = \mathsf{A} \cdot (((\mathsf{B} \cdot \mathsf{C}) \cup (\mathsf{D} \cdot \mathsf{E})) \cdot \mathsf{F})^* \cdot \mathsf{B}$$

Here the symbols $\cup$, $\cdot$, and * denote the regular expression operators *union*, *concatenation*, and *closure*, respectively, and parentheses are used to enforce order of evaluation. These operators have the following interpretation: The **union** operator is used to combine two distinct paths that start and end at the same point, the **concatenation** operator joins one path ending at a particular point with another one that starts there, and the **closure** operator represents the union of all paths that make any number of iterations around a loop. As the union operators defined later in this paper are not idempotent (where idempotent means $\forall \mathsf{A}, \mathsf{A} \cup \mathsf{A} \equiv \mathsf{A}$), it is vital that the path expressions we use are *unambiguous* in the sense that no path can be enumerated in two, or more ways (informally, there is no way to further simplify the path expression). For example, "$\mathsf{A} \cup \mathsf{A}$" enumerates A twice so this path expression is ambiguous (for a more formal definition, see Tarjan [Tar81b, Appendix B]).

We apply path expressions by interpreting each edge as having some type of value, and the regular expression operators (union, concatenation, and closure) as functions that combine and transform these values. We give an example of this process in the next section. Tarjan's fast path algorithm produces unambiguous path expressions very efficiently. In particular Tarjan's algorithm solves the single source path problem (one source many destinations) in $O(m\alpha(m, n))$ time on reducible flow graphs, where n is the number of nodes and m is the number of edges and α is a functional inverse of Ackermann's function [Tar81b]. This means we get a path expression for every pair of basic blocks in a procedure in $O(nm\alpha(m, n))$ time. To better utilize Tarjan's algorithm, a tree-like representation of path expressions is used to support on-demand updating when an edge weight is set to zero (a requirement described later).

Note that path expressions can be generated for arbitrary directed graphs. This means that path expressions can be computed for programs with unstructured or irreducible control flow (e.g., caused by use of ANSI C `goto` statements). Tarjan [Tar81b] provides two path expression algorithms—the one described above for reducible control flow, and another less efficient version that can compute path expressions for arbitrary directed graphs. As the benchmarks employed in this dissertation all have reducible control flow, this more general version of Tarjan's algorithm was used only in Chapter 5 (Section 5.3) of this dissertation, where a directed graph based upon a second-order model of control flow statistics is used that is not always reducible even when the underlying program control flow is reducible.

Table 2.1: Path Expression Mappings

	Reaching Probability	**Expected Path Length**	**Path Length Variance**
concatenation $[R_1 \cdot R_2]$	pq	$X + Y$	$v + w$
union $[R_1 \cup R_2]$	$p + q$	$\frac{pX + qY}{p + q}$	$\frac{p(v + X^2) + q(w + Y^2)}{p + q} - \left(\frac{pX + qY}{p + q}\right)^2$
closure $[R_1^*]$	$\frac{1}{1 - p}$	$\frac{pX}{1 - p}$	$\frac{p(v + X^2)}{1 - p} + \left(\frac{pX}{1 - p}\right)^2$

The mappings used for computing reaching probability, expected path length and path length variance are summarized in Table 2.1 and justified in the following sections. These mappings are not arbitrary, but rather arise from a rigorous analysis of the underlying probabilistic model. To the best of our knowledge, the mappings for expected path length, and path length variance are novel to this work. The mapping used for reaching probability also arises, for example, in the solution of systems of linear equations [Tar81a], and dataflow frequency analysis [Ram96]. However our method of zeroing edges to ensure the implied summation is over disjoint events appears to be novel. Note that the calculation of path length variance requires the calculation of the expected path length, which in turn requires the calculation of the reaching probability.

Reaching Probability

The reaching probability, $RP(x, y)$, from x to y, for $x \neq y$, may be computed as follows: Label all transitions in the Markov model with their respective transition probability. Set the probability of edges leaving y to zero, so that paths through y are ignored (because setting these edges to zero means these paths have zero probability). Then evaluate $RP(x, y)$ by recursively replacing the path expression in square braces in the first column in Table 2.1, with the value computed by the expression in the second column.

The validity of the mapping used for reaching probabilities arises from several facts: First, probability theory states that the probability of one event occurring out of a set of disjoint events is the sum of the individual probabilities of each event. Thus, the probability of taking any path encoded by the union of two path expressions is the sum because the path expressions formed by Tarjan's algorithm are unambiguous. Second, the assumption of independent branch outcomes implies that the probability of taking a particular path is the product of the probabilities of all branch outcomes along the path. This, combined with the fact that multiplication distributes over addition, allows us to evaluate the concatenation of two path expressions simply by multiplying their individual values because this is the same as adding the probabilities of each individual path enumerated by the combined path expression.

The closure mapping for a loop with probability p of returning from the loop header back to itself can be found by viewing the set of paths encoded by the closure operator as an infinite sum of products, and applying the well known formula for the geometric series:

$$\sum_{i=0}^{\infty} p^i = \frac{1}{1-p}\ , \ \text{if}\ |\,p\,| < 1$$

Note that the result of the closure mapping does not represent a probability, because the paths it combines are not disjoint events. The path expression analysis is valid if the profile data represents a program run to completion so that there are no loops with a loop probability of one.

Example: **The reaching probability from a to x.**

Each path from a to x in Figure 2.3 must start with edge A and end with edge B, however, in between there can be any number of iterations around the loop taking the path segment composed of edges DEF. The result of applying the procedure outlined above is:

$$\begin{aligned}
\mathsf{P(a,x)} &= \mathsf{A} \cdot (((\mathsf{B} \cdot \mathsf{C}) \cup (\mathsf{D} \cdot \mathsf{E})) \cdot \mathsf{F})^* \cdot \mathsf{B} \\
[\mathsf{P(a,x)}] &= 0.98 \cdot \left(\frac{1.0}{1.0 - \big(0.1(\underline{0.0}) + 0.9(1.0)\big) \cdot \big(0.999\big)} \right) \cdot 0.10 \\
&\approxeq 0.97
\end{aligned}$$

The value underlined in the denominator represents edge C and, as explained earlier must be set to zero to ensure paths going through x are ignored.

This example provides an important and perhaps counter-intuitive insight: The probability of reaching a block can only be accurately (or reliably) determined by examining global behavior. The probability of taking the path directly from a to x is only 0.098, yet the probability of reaching x at least once before control flow exits the region along the edges marked x or y in Figure 2.3, is 0.97, which is much higher. An intuitive explanation of this result is that the probability of exiting the loop each iteration, 0.001, is much lower than the probability of executing x each iteration, 0.1, so that it is very likely for the program to execute x at least once before the loop exits.

Posteriori Probability

To evaluate the posteriori probability that x precedes y, reverse the direction of control flow edges, and re-label them with the frequency a block precedes, rather than follows another block. Then, setting the probability of edges from x to successors of x, and from predecessors of y to y

to zero (referring to the new edge orientation), apply the mapping for reaching probability.

Expected Path Length and Variance

Using path expressions, the expected path length, and the path length variance from x to y (assuming the current state is x), can be computed as follows. With each edge we associate a 3-tuple. The first element represents the probability of branching from the predecessor to the successor (set to zero for edges emanating from y), the second element represents the length of the predecessor basic block, and the third element represents the path length variance of the edge, and is thus zero. Similarly, for path expressions R_1 and R_2 we associate 3-tuples $< p, X, v >$ and $< q, Y, w >$. The rules for computing the first, second, and third element are listed in the second, third, and fourth columns of Table 2.1, respectively. In a manner similar to the relationship between reaching probability and posteriori probability, we may define the posteriori expected path length and variance.

The mappings in the third and fourth column of Table 2.1 arise by analyzing the expected value (mean), and variance of the path length given the probability of following a particular path, as determined by the edge profile data. It can be verified that the mappings for concatenation and union in Table 2.1 satisfy the distributive law, as required. The mapping derivations for the expected path length are described first, followed by a sketch of the derivations for the path length variance (the full details of which are presented in Appendix A).

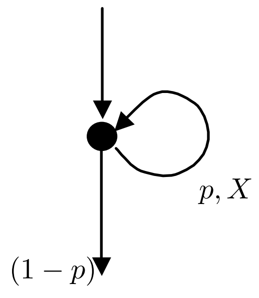


Figure 2.5: Analysis of the closure operator.

For a given path expression R, let $\sigma(R)$ represent the set of all paths enumerated by R. The correctness of the expected path length mapping for concatenation follows immediately from the fact that the expected value of the sum of a set of independent random variables equals the sum of the expected values of those random variables. The correctness of the formulation for the union operator follows immediately from the fact that R_1, R_2 and $R_1 \cup R_2$ are unambiguous and therefore:

$$\begin{aligned} [R_1 \cup R_2] &= E[R_1|\text{follow } p \in \sigma(R_1)] \cdot \Pr(\text{follow } p \in \sigma(R_1)) + \\ &\quad E[R_2|\text{follow } p \in \sigma(R_2)] \cdot \Pr(\text{follow } p \in \sigma(R_2)) \end{aligned}$$

To derive the formulation for closure we exploit the fact that the sum of a set of independent random variables is the sum of the expected values and focus on examining a loop with expected path length per iteration of X and backedge probability of p (see Figure 2.5). For the closure operator we are interested in the expected path length upon entering the loop up to, but not including the edge exiting the loop.

$$\begin{aligned} E[\,\text{closure length}\,] &= \sum_{p_i = \text{ path formed by iterating } i \text{ times, for all } i \text{ from } 0 \text{ to } \infty} E[\,\text{length}(p_i)] \cdot \Pr(p_i) \\ &= 0 \cdot (1-p) + X \cdot p(1-p) + 2X \cdot p^2(1-p) + ... \end{aligned}$$

The reason each term carries a factor of $(1-p)$ is that we are interested in the event that the loop iterates a specific number of times and then definitely exits. The above summation can be expressed as:

$$E[\,\text{closure length}\,] = p\,(1-p)\,X \sum_{i=0}^{\infty} i p^{i-1} = \frac{pX}{1-p}$$

The variance of the sum of two independent random variables is simply the sum of the variances, hence the concatenation operator for path length variance. As with the expected path length, the union and closure operators for path length variance are derived by evaluating the conditional expectation, marginalized over the actually taken path using the probabilities found with the reaching probability mapping. The detailed derivation is more elaborate than for the expected path length, and is therefore summarized in Appendix A.

Expected Path Footprint

Assuming x and y are in the same procedure, and ignoring storage requirements of subroutine calls, the expected path footprint between x and y, denoted $F(x,y)$, can be computed using the formula:

$$F(x,y) = \frac{1}{RP(x,y)} \sum_{v} \text{size}(v) \cdot RP_{\alpha}(x,v|\neg y) \cdot RP_{\beta}(v,y) \tag{2.1}$$

where the summation runs over all blocks v on any path from x to y for which y only appears as an endpoint[2], size(v) is the number of instructions in basic block v, and $RP_{\alpha}(x,v|\neg y)$ is the probability of traversing from x to v along any path except those through y,

$$RP_{\alpha}(x,v|\neg y) = \begin{cases} RP(x,v) \text{ s.t. no path thru } y & \text{if } x \neq v, \\ 1 & \text{if } x = v \end{cases} \tag{2.2}$$

This quantity is computed similar to $RP(x,v)$, except that edges leading from y are evaluated as having zero probability (in addition to those leading from v). $RP_{\beta}(x,y)$ is defined as,

$$RP_{\beta}(x,y) = \begin{cases} RP(x,y) & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases} \tag{2.3}$$

Equation 2.1 is significant because it allows us to evaluate the expected path footprint in terms of values we know how to compute efficiently. To take into account the code footprint used by subroutine calls we weigh the size of each callee basic block by the probability it is reached at least once.

The derivation of Equation 2.1, in particular the interpretation of $RP_{\beta}(x,y)$ and $RP_{\alpha}(x,v|\neg y)$, is as follows. The expected path footprint from x to y is the sum of the fraction of times we traverse from x to y following path p, times the footprint of path p, over all paths from x to y such that y only appears as the endpoint of any particular path:

$$F(x,y) = \sum_{p \in \sigma(P_o(x,y))} \text{freq}(p|x,y) \cdot f(p)$$

where $P_o(x,y)$ is the path expression enumerating all ways of going from x to y such that y is encountered only once (a by-product of the reaching probability computation), freq$(p|x,y)$ is the fraction of times starting from x and ending at y we followed path p, and $f(p)$ is the path

[2]This set can be found while evaluating the reaching probability.

footprint due to all blocks along p except y. By expressing the fraction of times we traverse from x to y following path p as the probability of taking path p starting from x, $\Pr(p)$, divided by the probability of reaching y from x, we can rewrite this as:

$$F(x,y) = \frac{1}{RP(x,y)} \cdot \sum_{p \in \sigma(P_o(x,y))} f(p) \cdot \Pr(p)$$

By expanding $f(p)$ in terms of the sizes of the unique basic blocks encountered along the path p we can express the latter equation as:

$$F(x,y) = \frac{1}{RP(x,y)} \cdot \sum_{p \in \sigma(P_o(x,y))} \left(\sum_{v \in p, v \neq y} \text{size}(v) \cdot \Pr(p) \right)$$

We further transform this equation by exchanging orders of summation to obtain:

$$F(x,y) = \frac{1}{RP(x,y)} \cdot \sum_{v \in B} \text{size}(v) \cdot \sum_{p \in P_o'} \Pr(p) \qquad (2.4)$$

where B is the set of all blocks except y, which are passed through one or more times by at least one path in $\sigma(P_o(x,y))$, and P_o' is the subset of paths in $\sigma(P_o(x,y))$ passing through v. Each path from x to y passing through v, such that y appears only as an endpoint, can be expressed as the concatenation of two disjoint paths: Assuming x, and y are distinct from v, the first path goes from x to v without passing through y such that v appears only as an endpoint, and the second path goes from v to y such that y appears only as an endpoint. If x equals v, the first path consists of the single vertex x. When v is y, the second path consists of the single vertex y. This decomposition defines two path sets. The sum of the probabilities over all paths in the first set is given by Equation 2.2, and the sum of the probabilities over the paths in the second set is, given by Equation 2.3. Thus we can express the sum of the probabilities of all paths from x to y passing through v, such that y only appears as the product of these factors:

$$\sum_{p \in P_o'} \Pr(p) = RP_\alpha(x, v | \neg y) \cdot RP_\beta(v, y)$$

Substituting this result into Equation 2.4 yields Equation 2.1.

Note that $RP_\alpha(x, v | \neg y) \leq RP(x, v)$ because paths from x to v that pass through y are are included when computing $RP(x, v)$ but not included when computing $RP_\alpha(x, v | \neg y)$. Thus, we may conservatively estimate path footprints by using $RP(x, v)$ rather than $RP_\alpha(x, v | \neg y)$. This approximation was used when generating the spawn-target pairs evaluated in Chapter 3 and 4 and yields an $O(n)$ reduction in the number of edge weight modifications (e.g., setting an edge weight to zero) that need to be evaluated.

Eliminating Spawn-Point Redundancy

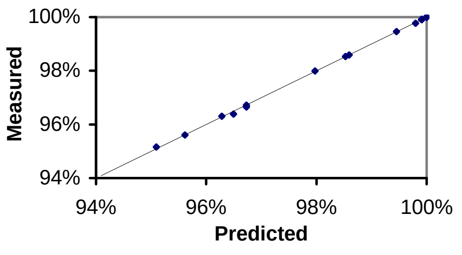
A spawn-point s_1 implies another spawn-point s_2 for a given target t if any path from s_1 to t passes through s_2. In other words, if s_2 is reached along the path from s_1 to t, spawning when the main thread reaches s_2 is redundant. Two spawn-points are said to be independent with respect to a common target if neither spawn-point implies the other. By selecting a set of mutually independent spawn-points we are assured that only one will execute for a given dynamic instance of t. Furthermore, the reduction in execution time due to independent spawn-points is additive. Path expressions provide a convenient way to determine spawn-point independence: given s_1, s_2 and t, we can determine whether s_1 implies s_2 by checking whether any edge in the path expression from s_1 to t starts or ends with s_2 after eliminating edges emanating from t. This can be performed efficiently while evaluating the reaching probability.

2.3 Framework Accuracy

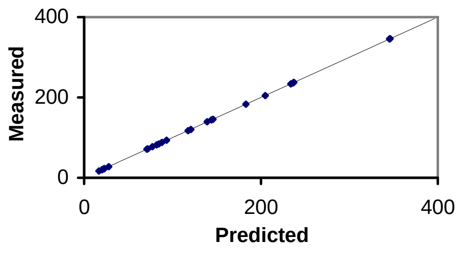
To assess the accuracy of the modeling framework, we measure the reaching probability, expected path length, path length variance, and expected path footprint for 25 spawn-target pairs chosen by the selection algorithm for subroutine `Evaluate()` in benchmark `crafty` by simulating a Markov model of the subroutine, and by collecting statistics from program traces. The `Evaluate()` subroutine accounts for roughly one quarter of all I-cache misses in `crafty`, and its control flow graph is non-trivial—containing 230 basic blocks, 345 edges, and several loops. Figure 2.6(a)-2.6(d) compare predicted statistics with a Monte Carlo simulation based upon an idealized Markov chain with the exact same edge probabilities used by the framework ("measured" in these figures represents averages from 1 million trials per spawn-target pair). The units in Figures 2.6(b), (c) and (d) are Itanium® bundles[3], while Figure 2.6(a) plots relative frequency versus probability both measured in percentage. Note that path variation is reported as the standard deviation—i.e., the square root of the variance. The expected footprint predictions are computed using Equation 2.1—i.e., without substituting $RP(x, v)$ for $RP_\alpha(x, v|\neg y)$. The strong correlation present in Figures 2.6(a)-2.6(d) underscores the robustness of the path expression based methodology described in Section 2.2, and Table 2.1.

Figures 2.7(a)-2.7(d) shows a similar comparison for the same spawn-target pairs, but in-

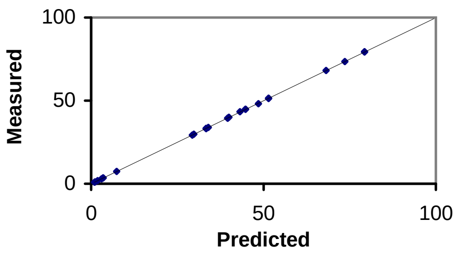
[3]Each Itanium bundle contains two, or (more usually) three instructions.



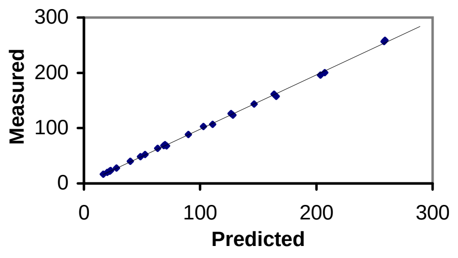
(a) reaching probability



(b) expected path length



(c) path length variation

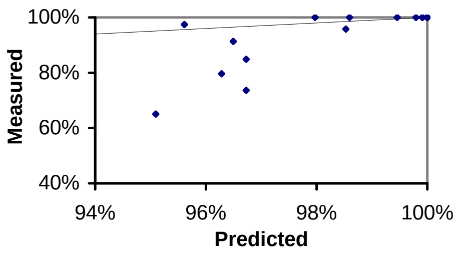


(d) expected footprint

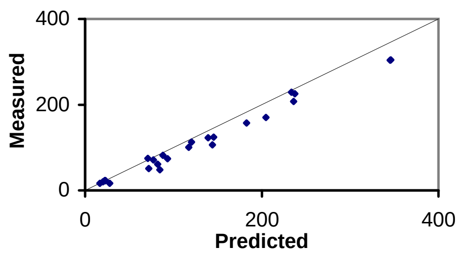
Figure 2.6: Monte Carlo Simulation vs. Framework.

stead plots statistics gathered during actual program execution of the 100 million instruction segment used in Chapter 3 for performance evaluation, rather than from the 1 million trial spawn-target sample paths generated using the markov chain model plotted in Figure 2.6. Both the reaching probability, and path length variation exhibit several outlying data points. However, note that 16 of the 25 data points are clustered within 1% of the upper right corner in Figure 2.7(a) so that reaching probability is more correlated than casual observation may suggest. The outlying data points appear to result from two approximations in the modeling framework: First, transition probabilities were determined by profiling over the *whole* program execution, which averages out behaviors unique to distinct phases of execution. Second, even within a single phase of program execution the correlation between branch outcomes is ignored in the model. For example consider the control flow graph in Figure 2.8 reproduced from Ofelt and Hennessy [OH00, Figure 4.1]. In this example, the observed frequency of actual execution paths are compared with those predicted using a first order Markov chain approximation. For instance, the path ABDFG is never executed by the program, however the model indicates it is expected to be traversed 9 times out of 100 passes through this region of the program. Similar observations have been made earlier [BL96].

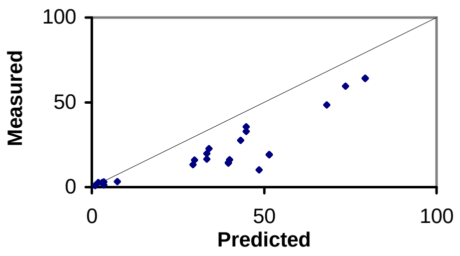
Indeed, both of the approximations described above can lead to the existence of paths never



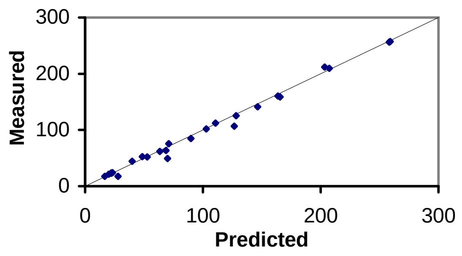
(a) reaching probability



(b) expected path length



(c) path length variation



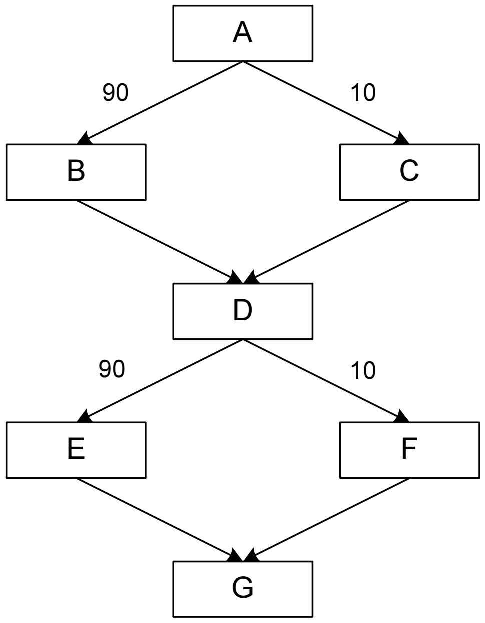
(d) expected footprint

Figure 2.7: Execution Trace Statistics vs. Framework.

executed by the program having a finite probability in the model. Similarly, some paths with a very small *predicted* probability of occurring in the model may actually have a high probability of occurring during execution due to strong branch correlation. Imprecise path probability estimates can lead to a weighting of possible outcomes during path expression evaluation that differs from real program execution. By separately modeling program behavior in distinct program phases [SPHC02a], and taking into account the impact of branch correlation (perhaps by extending the current approach to use higher order Markov chain models) these quantities may be predicted with greater accuracy, however such enhancements are beyond the scope of this dissertation.

2.4 Summary and Contributions

This chapter described and evaluated the modeling framework proposed in this dissertation. This framework is the core contribution of this dissertation. The accuracy of the framework was verified experimentally and shows good agreement with actual program behavior with a varying degree of sensitivity between the different statistical quantities of interest. In particular, the expected path length and expected path footprint show the closest agreement, and the reaching probability and path length variance show greater sensitivity. In subsequent chapters the framework is applied



Path	Real Freq.	BB Freq.
ABDEG	90	81
ABDFG	0	9
ACDEG	0	9
ACDFG	10	1

(This figure reproduces Figure 4.1 in Ofelt and Hennessy [OH00])

Figure 2.8: Example of inaccuracy due to first-order Markov model approximation

to the task of modeling and optimizing helper threads for instruction prefetch and data prefetch.

Chapter 3

Prescient Instruction Prefetch

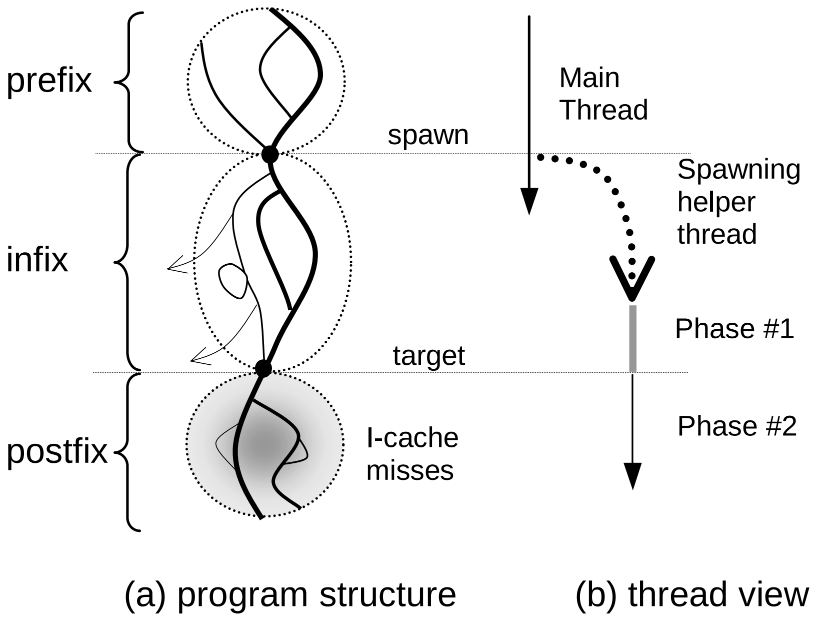
3.1 Introduction

Instruction supply may become a substantial bottleneck in future generation processors that have very long memory latencies and run application workloads with large instruction footprints such as database servers [RBG$^+$01]. Prefetching is a well-known technique for improving the effectiveness of the cache hierarchy. This chapter and the next investigates the use of spare simultaneous multithreading (SMT) [TEL95, Eme99, Int] thread resources for prefetching instructions and focuses on single-threaded applications that incur significant I-cache misses.

Even though SMT has been shown to be an effective way to boost throughput performance with limited impact on processor die area [HS01], the performance of single-threaded applications does not directly benefit from SMT and running such workloads may result in idle thread resources. Recently, a number of proposals have been put forth to use idle multithreading resources to improve single-threaded application performance by running small *helper threads* that reduce the latency impact of D-cache misses, and branch mispredictions that foil existing hardware mechanisms [DS98, CSK$^+$99, CTYP02, ZS00, ZS01, RS01, MPB01, APD01, Luk01, CWT$^+$01, CTWS01, LWW$^+$02, KY02].

However, prior to the publications associated with this dissertation [AMC$^+$03, ACH$^+$04], there has been little published work focused specifically on improving I-cache performance using such helper threads. A key challenge for instruction prefetch is to accurately predict control flow sufficiently in advance of the fetch unit to tolerate the latency of the memory hierarchy.

In this chapter we describe how to apply the modeling framework described in Chapter 2 to reduce processor stall cycles caused by instruction cache misses. A technique called *prescient*



(a) Control-flow graph with spawn and target points highlighted. From profiling, the region labeled postfix is known to encounter heavy instruction cache misses. (b) Two phases of helper thread execution: Phase #1 live-in precomputation; Phase #2 postfix precomputation and instruction prefetch. Note that distance along the vertical axis in (b) corresponds to the number of dynamic instructions executed by the main thread (i.e., it does not directly measure time).

Figure 3.1: Prescient Instruction Prefetch.

instruction prefetch is introduced that uses helper threads to prefetch instructions for single-threaded applications. The next chapter (Chapter 4) describes specific implementation strategies and hardware support mechanisms for prescient instruction prefetch.

The term prescient carries two connotations: One, that the helper threads are initiated in a timely and judicious manner based upon a profile-guided analysis of a program's global behavior, and two, that the instructions prefetched are useful as the helper thread follows the same control-flow path through the program that the main thread will follow when it reaches the code that the helper thread has prefetched.

Profile information is used to identify code regions that incur significant I-cache misses in the single-threaded application[1]; candidate target points are then identified as instructions closely

[1]While the statistics of the "path footprint"—introduced in Section 2.2.2 on page 27—could, in principle, be used to estimate the instruction cache misses incurred by each basic block, in this dissertation instruction cache misses are measured directly. Studying techniques for accurately predicting the I-cache miss rates using only control flow statistics is beyond the scope of this dissertation.

preceding the basic blocks that incur the I-cache misses. For each candidate target identified in the single-threaded application, a set of corresponding spawn points are found that can serve as trigger points for initiating the execution of a helper thread for prefetching instructions beginning at the target. Once a spawn-target pair is identified, a helper thread is generated and attached to the original program binary (i.e., the main thread). At run time, when a spawn point is encountered in the main thread, a helper thread is initialized and begins execution on an idle thread context. The execution of the helper thread effectively prefetches for anticipated I-cache misses along a control flow path subsequent to the target.

Figure 3.1 illustrates the operation of prescient instruction prefetch. This figure is very similar to Figure 2.2, except that the postfix region has been selected because it often incurs a large number of instruction cache misses.

In general, the helper thread may need to precompute some live-in values before starting to execute the program code in the postfix region. Since effective instruction prefetch is enabled by accurate resolution of branches in the postfix region, we study helper threads in which the precomputation consists of the backward slice of these branches. A register, or memory location is said to be ‘live-in’ to a region, if its value is read before being written in that region. As shown in Figure 3.1(b), helper thread execution consists of two phases. The first phase, live-in precomputation, reproduces the effect of the code skipped over in the infix that relates to the resolution of branches in the postfix. We refer to these precomputation instructions as the *infix slice*. Instructions in the infix region that are not related to any branch instruction in the postfix region are not executed. This is the key aspect that allows the helper thread to run faster and reach the target point earlier than the main thread does. In the second phase, the helper thread prefetches same code in the postfix region that the main thread will when the main thread reaches the target. The computation in the second phase resolves control-flow for the helper thread in the postfix region. Prefetching may be achieved either by using prefetch instructions in the helper thread, or by virtue of the shared address space of the helper thread and the main thread, by directly executing the postfix region of the application within the helper thread thread context (to effectively prefetching instructions for the main thread). The postfix region is a predefined region of the application and the helper thread is terminated after it finishes executing the postfix region or when the main thread catches up with it.

The prescient instruction prefetch mechanism differs from traditional branch predictor based instruction prefetch mechanisms, such as that proposed by Reinman et al. [RCA99], in that it

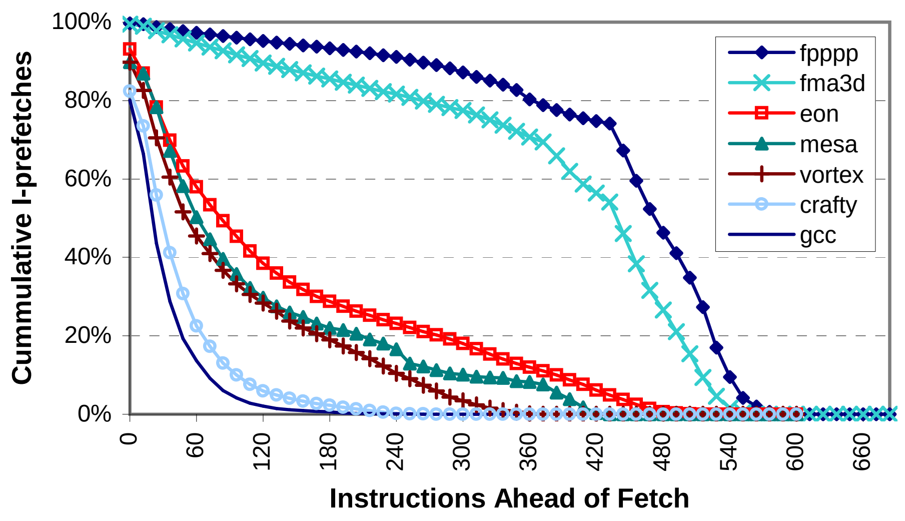


Figure 3.2: Prefetch distribution for a branch predictor guided I-prefetch mechanism

uses a global view of control-flow gained from program profile and is thus able to anticipate potential performance degrading regions of code from a long range (e.g., the distance between spawn and target could be thousands of dynamic instructions). The key challenges are two-fold: One, identification of an appropriate set of distant yet strongly control-flow correlated pairs of spawn-target points; and two, accurate resolution of branches in the postfix region. It is possible that inaccuracy in the resolution of some postfix branches inside the helper thread may not matter if the same cache blocks are prefetched independent of whether a branch is precomputed correctly. However, the Itanium instruction set employed in this study contains support for predication, so many branches associated with nested "if" statements undergo "if-convertion" [Tow76, AKPW83, PS91] resulting in a larger number of instructions between branch instructions. Quantitative analysis of how frequently a postfix branch is evaluated even when it does not impact the cache blocks prefetched is beyond the scope of this dissertation.

To illustrate how prescient instruction prefetch may potentially improve performance beyond existing techniques, Figure 3.2 shows a plot of the cumulative distribution of I-prefetch distances for prefetches successful in reducing or eliminating the latency impact of an I-cache miss generated by a branch predictor guided instruction prefetch technique called *fetch directed instruction prefetch* [RCA99] (abbreviated to FDIP herein, see Table 1 for implementation details). A point (x, y) in Figure 3.2 indicates that y percent of instruction prefetches required to avoid an I-cache miss were issued x or more dynamic instructions ahead of the fetch unit. The effectiveness of this prefetch mechanism is a consequence of the branch predictor's ability to anticipate the path of the program through code regions not in the I-cache. What this figure shows is that as memory latency grows and the minimum prefetch distance required to tolerate that latency increases, the fraction of timely prefetches will tend to decrease. The same trend occurs for a fixed memory latency as more parallelism is exploited by the processor core.

By exploiting the global control-flow correlation of spawn and target, prescient instruction prefetch threads can provide ample, scalable slack and achieve both timeliness and accuracy.

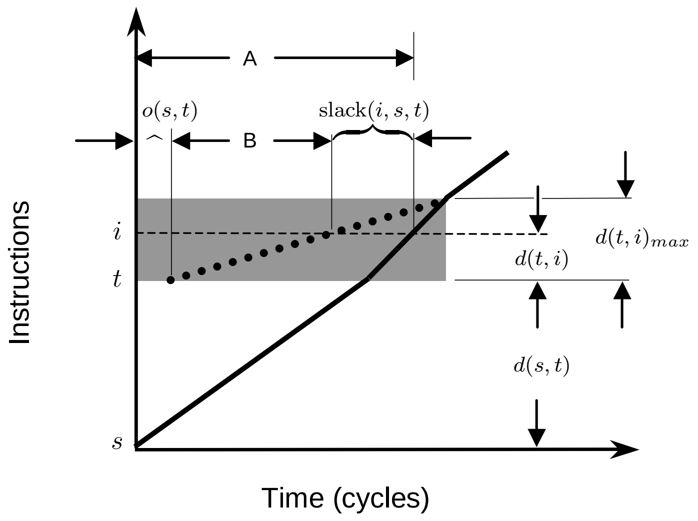


Figure 3.3: Illustration of Equations 3.1 and 3.2.

3.2 Optimization Algorithm

In this section we develop an optimization algorithm for selecting prescient instruction prefetch helper threads. The algorithm is developed by considering the behavior of a single helper thread instance (i.e., a sample path of the random process) given the statistical model described in Chapter 2, and then using the modeling framework to account for the probabilistic nature of the model. Later in the chapter, after describing the optimization algorithm, prescient instruction prefetch is evaluated using a simulation model in which the infix precomputation overhead is effectively ignored while other components are modeled in full detail.

3.2.1 Analysis: Single Helper Thread Instance

To characterize the distance that a helper thread runs ahead of the main thread, we model the prefetch slack of an instance of instruction i, targeted by a helper thread spawned at s with target t, using the following expression (illustrated in Figure 3.3):

$$\begin{aligned}\text{slack}(i,s,t) = &\overbrace{(\text{CPI}_m(s,t)\cdot d(s,t) + \text{CPI}_m(t,i)\cdot d(t,i))}^{\text{A}}\\ &\underbrace{-\text{CPI}_h(t,i)\cdot d(t,i)}_{\text{B}} - o(s,t)\end{aligned} \tag{3.1}$$

where $\text{CPI}_m(s,t)$ represents the average cycles per fetched instruction for the main thread when traversing the infix region between this particular instance of s and t; $\text{CPI}_m(t,i)$ represents the average cycles per fetched instruction for the main thread in the postfix region between t and i; $\text{CPI}_h(t,i)$ represents the average cycles per fetched instruction for the helper thread between t and i; $d(x,y)$ denotes the distance between instructions x and y, measured in number of instructions executed; and $o(s,t)$ represents the overhead, in cycles, incurred by the helper thread of spawning at s and performing precomputation for live-ins to the postfix region[2] A given instance of a helper thread can reduce the I-cache miss latency of an instruction access in the main thread if the prefetch slack for this instruction is positive.

The amount of prefetching any helper thread can perform, and therefore the maximum extent of the postfix region it can target, are limited by two factors. First, prefetching will improve the average IPC of the main thread, but not that of the helper thread, leading to an upper bound on how far ahead a helper thread can run before the main thread will catch up to it. In particular, the main thread will catch up with the helper thread when $\text{slack}(i,s,t) = 0$ in Equation 3.1 or,

$$d(t,i)_{max} = \frac{\text{CPI}_m(s,t)\cdot d(s,t) - o(s,t)}{\text{CPI}_h(t,i) - \text{CPI}_m(t,i)} \tag{3.2}$$

Figure 3.3 portrays a graphical representation of this concept by plotting instructions executed versus time for a portion of a program's execution trace. The solid line indicates the progress of the

[2]For simplicity, we assume this overhead is constant however a more detailed analysis would incorporate statistics of the execution time of the precomputation slice versus the main thread. Such analysis is deferred to Chapter 5 were the subject of data prefetch helper threads is investigated in detail.

main thread, and the dotted line indicates the progress of the helper thread after overhead $o(s,t)$ due to thread invocation and live-in precomputation. The slack of one particular instruction i is shown. The helper thread ceases to provide useful prefetch slack when the main thread catches up to it. This point is where the dotted line intersects the solid line. The distance computed in Equation 3.2 corresponds to the height of the shaded box in Figure 3.3.

The other factor limiting the postfix region is the size of the infix slice required for target live-in precomputation. The size of the infix slice depends upon the amount and type of code skipped between the spawn and target, and the number of branches in the postfix region. As the distance between spawn and target grows, the number of operations in the infix that may affect the outcome of a branch in the postfix increases. On the other hand, given a fixed spawn-target distance, increasing the size of the postfix may require additional branches to be precomputed; some of these may depend upon computation in the infix that is unrelated to earlier postfix branches.

It is interesting to note that Equation 3.2 implies a lower bound on the number of concurrent helper threads required to prefetch all instructions. Assuming negligible precomputation overhead (i.e., $o(s,t) = 0$),

$$\begin{array}{c}\text{\# helper threads for}\\ \text{full coverage}\end{array} \geq \left\lceil \frac{\mathrm{CPI}_h}{\mathrm{CPI}_m} \right\rceil$$

where CPI_m, and CPI_h are the values assuming full coverage is achieved. Lower CPI_h reduces the number of concurrent helper threads required. In Chapter 4 a technique called finite state machine recall is introduced that increases the efficiency of instruction prefetch helper threads by reducing CPI_h.

The necessity (related to *timeliness* in the sense of not being too early) of instruction prefetches can be assessed by analyzing the instruction footprint distribution along paths between the target and the spawn. The *timeliness* (in the sense of not being too late) of prefetches can be quantified by analyzing the distribution of slack values in relation to the latency of the memory hierarchy by viewing the distances in Equation 3.1 as statistical quantities. Finally, the *accuracy*, and *coverage* of prefetches can be gauged by analyzing the control flow correlation quantified in the next section by the reaching probability, and posteriori probability.

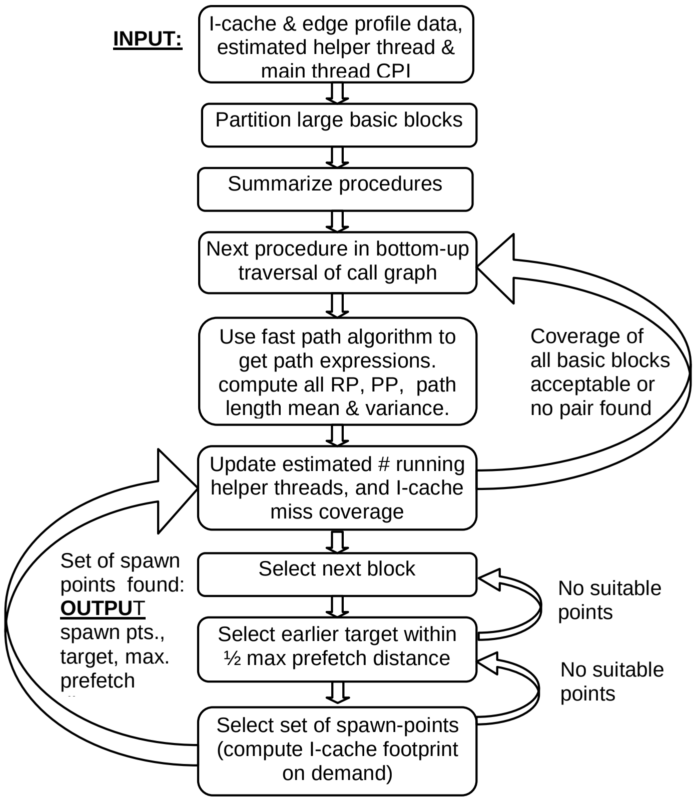

INPUT:
I-cache & edge profile data, estimated helper thread & main thread CPI
Partition large basic blocks
Summarize procedures
Next procedure in bottom-up traversal of call graph
Use fast path algorithm to get path expressions. compute all RP, PP, path length mean & variance.
Coverage of all basic blocks acceptable or no pair found
Update estimated # running helper threads, and I-cache miss coverage
Set of spawn points found: OUTPUT spawn pts., target, max. prefetch
Select next block
No suitable points
Select earlier target within ½ max prefetch distance
No suitable points
Select set of spawn-points (compute I-cache footprint on demand)

Figure 3.4: Spawn-Target Selection Algorithm

3.2.2 Optimization via Greedy Spawn-Target Pair Selection

This section describes one particular spawn-target selection algorithm based upon the framework presented in Section 2.2. As part of this dissertation, this algorithm was implemented for the Itanium® architecture and the effectiveness of this algorithm demonstrated (the experimental results are presented in Section 3.3). A high-level flow-chart of the algorithm is shown in Figure 3.4.

The inputs to the algorithm are: a branch frequency profile, instruction cache-miss profile, estimated main CPI, and estimated helper thread CPI. The output is a set of mutually independent spawn-target pairs, and associated postfix region sizes. Profile data is supplied via procedure control flow graphs annotated with basic-block and branch execution frequencies, and total instruction cache misses per basic block. The Itanium® simulator was modified to collect both the program's control flow structure and the branch outcome frequencies. The collected control flow graphs include information about procedure calls such as the frequency a given subroutine is called from a given site. A call site is considered to be a basic block boundary, the graph is augmented with an edge from the call instruction to the instruction following the call instruction in the caller (i.e., the return point of the call instruction). This edge has probability one minus the probability the call is predicated false, and path length and variance computed by summarizing subroutines.

The algorithm first splits control flow graph nodes that represent large basic blocks into linear sequences of nodes, each representing only a portion of the original basic block. Spawn and target locations are constrained to the beginning of basic blocks to reduce complexity—without performing this step it may happen that some instructions cannot be prefetched as no admissible target would be "close enough" to these instructions that a helper thread could reach them before being caught by the main thread. Next, the algorithm computes procedure summary information. In particular, the expected path length and variance from entry to exit, and the reaching probabilities from the entry to each block are computed. The latter is used for computing the expected path footprint taking into account procedure calls.

Once summary information is computed, we rank the basic blocks by the absolute number of I-cache misses they generate. We target basic blocks that account for the top 95% of all instruction cache misses. In each procedure visited, the reaching probability, expected path length and variance, posteriori probability, are computed between *every* pair of basic blocks according to the method described in Section 2.2.2. It may be possible to reduce the time it takes the algorithm to compute spawn-target pairs by pruning some of these evaluations.

We maintain an estimated number of instruction cache misses remaining in each block given the spawn-target pairs already selected. The initial value is the number of I-cache miss found during profiling. Then, we iteratively select the block in the current procedure with the highest estimated remaining I-cache misses, determine a target and corresponding set of independent spawn points for this block, and update the estimated remaining cache misses in all blocks that may be prefetched by helper threads with these spawn-target pairs. The latter update is based on estimating the amount of coverage each block receives using the posteriori probability, and the probability the slack will be sufficient to hide the memory latency. Similarly, we maintain an estimated number of running helper threads at each point in the program such that a spawn-point is not selected if this selection leads to a high probability of attempting to concurrently run more helper threads than available thread contexts.

For each selected block v, potential targets are earlier blocks with high posteriori probability of being visited before v. Thus, the targets do not necessarily suffer heavy I-cache misses themselves. It is often necessary to perform this step to find targets with high reaching probability from potential spawn points. The selection process for spawn and target are coupled in the following way: A set of potential targets with posteriori expected distance less than half the (preset) maximum postfix distance (where *postfix distance* refers to the number of instructions executed after the target) we wish to allow[3] is generated, and ranked in descending order by distance from the selected block. By selecting the target to be an earlier block in this way, we reduce the selection of spawn-target pairs with overlapping postfix regions. For each potential target in turn, a set of independent spawn points is found using the following process.

For a given target t, a set of mutually independent spawn points are selected among all blocks in the procedure with reaching probability to t greater than a threshold[4], by computing a heuristic value indicating their effectiveness as spawn points for the target. In particular, the merit of a given spawn-point s is computed as the product of the following factors: The first factor is the posteriori probability that the spawn precedes the target. This factor along with the expected path footprint quantify the fraction of I-cache misses at the target that are covered by instances of the helper thread. Furthermore, together with the restriction on reaching probability it ensures the spawn and target are control flow correlated. The second factor uses the expected path footprint to penalize those spawn points whose average target-to-spawn footprints are less than the cache size, because this condition implies a greater likelihood the prefetched instructions are still in the I-cache (for target-to-spawn footprints less than the cache size we reduce the value by a factor of the cache size divided by the expected footprint size). Similarly, spawn points with expected spawn-to-target path footprints larger than the instruction cache capacity are given a value of zero. The third factor is the size of the postfix region the helper thread can prefetch by finding the *expected maximum postfix distance* that always provides a minimum threshold probability[5] that the slack of the last prefetch issued by the helper thread is still positive (for

[3]The latter was experimentally set to 100 Itanium® bundles. Ignoring infix precomputation overhead, the size of the postfix region that can be targeted effectively depends upon the latency of the memory hierarchy and the average spawn-target distance. See, for example, row 4 in Table 4.2. Another consideration is that a larger postfix region may require a higher precentage of instructions from the infix region be included in the infix slice. An extensive evaluation of the sensitivity of resulting speedup to the maximum postfix distance is beyond the scope of this work.

[4]Experimentally set to 0.95. This value does not depend upon microarchitecture parameters such as memory latency, but rather the behavior of the programs under the influence of typical input data. Some limited exploration of the sensitivity of performance to this parameter is presented in Section 3.3.2.

[5]Set to 0.5 in this study with minimal experimental exploration. This value was primarily set with the inuition that a prefetch should be "fairly likely" to provide benefit.

simplicity we assume path lengths take on a Gaussian distribution). As spawn-target distance increases, this term increases until the number of instructions the helper thread can prefetch reaches the preset maximum postfix distance. The number of instructions the helper thread is likely to prefetch before getting caught by the main thread is also recorded (and used to terminate the helper thread during simulation if the main thread does not catch up with the helper thread first).

3.3 Simulation Results

This section presents a performance analysis of prescient instruction prefetch based upon cycle accurate simulation. We examine the performance impact of prescient instruction prefetch threads defined by spawn-target pairs selected using the algorithm described in Section 3.2.2.

3.3.1 Methodology

We select four benchmarks from Spec2000 and one from Spec95 that incur significant instruction cache misses on the baseline processor model. The application binaries were built with the Intel Electron compiler [BCC^{+}00], all, except for `fpppp`, with profile feedback enabled[6]. We profile the branch frequencies and I-cache behavior by running the programs to completion. The spawn-target generation algorithm selected between 34 and 1348 static spawn-target pairs per benchmark as shown in Table 3.1, which also quantifies the number of dynamic occurrences of these spawn target pairs, and the average distance between spawn and target (infix), and the average number of instructions prefetched per dynamic helper thread instance for a 4-way SMT (postfix). The small number of static pairs implies a small instruction storage requirement for prescient instruction prefetch helper threads. The large infix size means long memory latencies can be tolerated, while the small postfix size relative to infix size results from the helper threads running slower than the main thread.

The baseline architecture is a cycle-accurate research in-order SMT processor model for the Itanium® architecture [HMR^{+}00] based upon SMTSIM [TEL95] with microarchitectural parameters as summarized in Table 3.2. The baseline uses a hardware instruction prefetch mechanism

[6]These Itanium binaries were provided by members of the Intel Research Compiler Group. The only version of `fpppp` that was conveniently available when the study was conducted were generated without profile guided optimizations.

Table 3.1: Spawn-Target Characteristics

benchmark	# static	# dynamic	infix	postfix
145.fpppp	62	378528	622	162
177.mesa	34	210519	1186	255
186.crafty	166	560200	573	129
252.eon	152	407516	691	120
255.vortex	1348	438722	1032	142

that issues a prefetch for the next sequential line on a demand miss, and supports Itanium® instruction stream prefetch hints tagged onto branch instructions by the compiler. The rest of the memory hierarchy organization models latencies resembling an Itanium® 2 at 2GHz. Evaluation of the performance impact of prescient instruction prefetch on an out-of-order SMT processor model [WWK^{+}01] may show greater benefit due to the tendency for out-of-order processors to achieve higher average IPC thus placing increased pressure on fetch bandwidth and magnifying the impact of the instruction cache miss latency on performance, however this is beyond the scope of this dissertation.

We evaluate the potential of prescient instruction prefetch assuming values for all live-in registers, and memory operands to the postfix region are available at no cost as soon as the helper thread spawns. A helper thread is spawned when the main thread commits a spawn-point. If no free thread contexts are available the spawn-point is ignored. The helper thread may begin fetching from the target the following cycle and runs for its expected maximum postfix distance before exiting. If the main thread catches up with it before that point the helper thread is killed and stops prefetching instructions. Helper thread instructions contend for execution resources with the main thread, but have reduced fetch and issue priorities relative to the main thread. We assume store operations executed by helper threads do not commit to memory, but that they correctly forward their values to dependent loads within the same helper thread. Once a helper thread is stopped and its instructions drain from the pipeline, the thread context is available to run other helper threads. As this is a limit study we do not simulate the effect of a helper thread spawned when the target does not appear (only the last spawn-point seen before a given instance of a target-point initiates a helper thread) and we do not model off-path fetching by helper threads. Helper threads branches are resolved by consulting a perfect oracle branch predictor. Therefore they do not interfere (constructively or destructively) with branch predictions made for the main thread. Note that the positive effect on helper thread average IPC (and therefore prefetch slack) of consulting an oracle branch prediction is diluted both by the lower priority of helper threads access to fetch and execution resources and the larger number of instruction (and potentially data) cache misses they incur relative to the main thread in the postfix region[7].

Table 3.2: Processor Resources

Threading	SMT processor with 2, 4, or 8 hardware threads.
Pipelining	In-order: 12-stage pipeline.
Fetch	2 bundles from 1, or 1 bundle from 2 threads prioritize main thread, helpers ICOUNT [TEE+96].
I-Prefetch	next line prefetch (triggered on miss) stream prefetcher (triggered by compiler hints) max. 4 outstanding prefetches per context
Branch Pred.	2k-entry gshare. 256-entry 4-way assoc. BTB. helper threads: oracle branch prediction
Issue	2 bundles from 1, or 1 bundle from 2 threads prioritize main thread, helpers: round-robin.
Function units	4 int., 2 fp., 3 br., and 2 data mem. ports
Register files per thread	128 int, 128 fp, 64 predicate, 8 br. 128 control registers.
Caches	L1 (separate I&D): 16KB (each). 4-way. 1-cyc. L2 (shared cache): 256KB. 4-way. 14-cycles L3 (shared cache): 3072KB. 12-way. 30-cycles. Fill buffer: 16 entries. caches have 64B lines. helper threads: infinite store buffer 1-cyc.
Memory	230-cycles. TLB miss penalty: 30 cyc.

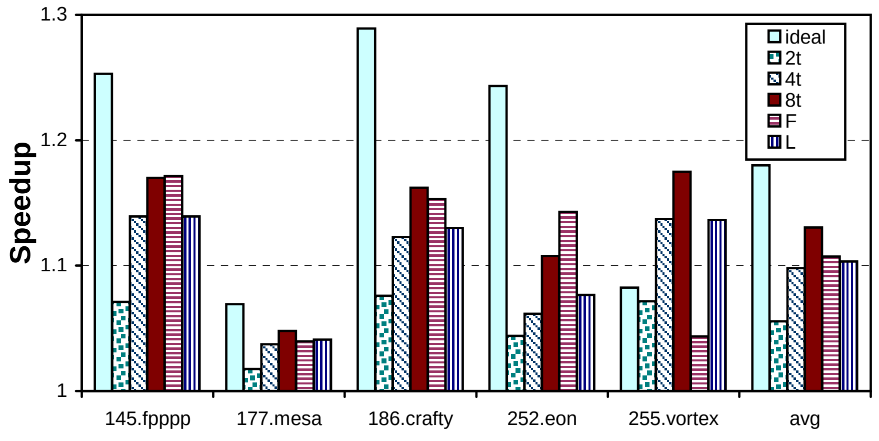
To evaluate the performance impact of prescient instruction prefetch we collect data for 100 million instructions after warming up the cache hierarchy while fast-forwarding past the first billion instructions.

3.3.2 Results

Figure 3.5 illustrates the speedup of prescient instruction prefetch compared to the baseline model. The first bar on the left for each benchmark is the speedup if *all* instruction accesses hit in the first level cache, and shows speedups ranging from 7% to 25% with a harmonic mean of 18%. Thus the benchmarks we study do indeed suffer significantly from instruction cache misses. The bars labeled 2t, 4t, and 8t represent the speedup of idealized prescient instruction prefetch on a machine with 2, 4 and 8 hardware thread contexts respectively (the bars labeled F and L are

[7]Exact quanitification of the components of helper thread IPC (such as that shown in Figure 4.11 on page 72 in Chapter 4 for each application's main thread) is beyond the scope of this dissertation.

described later). In each case spawn-target pairs are generated assuming only one main thread will share processor resources with the helper threads. Speedup improves with increasing number of hardware thread contexts with harmonic mean speedups of 5.5%, 9.8%, and 13% respectively, with speedups of up to 4.8% to 17% on the 8 thread context configuration. As might be expected, increasing the number of concurrent helper threads leads to a reduction in the IPC of individual helper threads due to resource contention. The last two bars in this figure will be described below.



"ideal" speedup if all I-accesses hit (e.g., infinite instruction cache with same hit latency as actual instruction cache). "nt" prescient instruction prefetch speedup for n thread contexts. "F" spawn-target pairs selected assuming four thread contexts and *fast* helpers (low CPI$_h$), performance modeled so that helpers do not block on an I-cache miss. "L" 4t and spawn-target pairs selected with *low* reaching probability threshold (0.75).

Figure 3.5: Limit study performance.

Most applications see a substantial reduction in the number of I-cache misses that improves as additional thread contexts are made available with a reduction in I-cache misses averaging 16% for 2t, 33% for 4t, and 42% for 8t. To better understand how speedup is obtained, as well as the quality of the spawn-target selection algorithm, Figure 3.6 shows the source of remaining I-cache misses normalized to the number of baseline instruction cache misses. Each miss is classified into one of five categories: "no target" means that the cache miss is not preceded by an instruction selected as a target, within that target's maximum postfix distance; "no spawn" means that although there is at least one target, none of them were preceded by a spawn since the last occurrence of the target; "no context" means there was a spawn-target pair that could have prefetched the I-cache miss, but no SMT thread context was available when the spawn

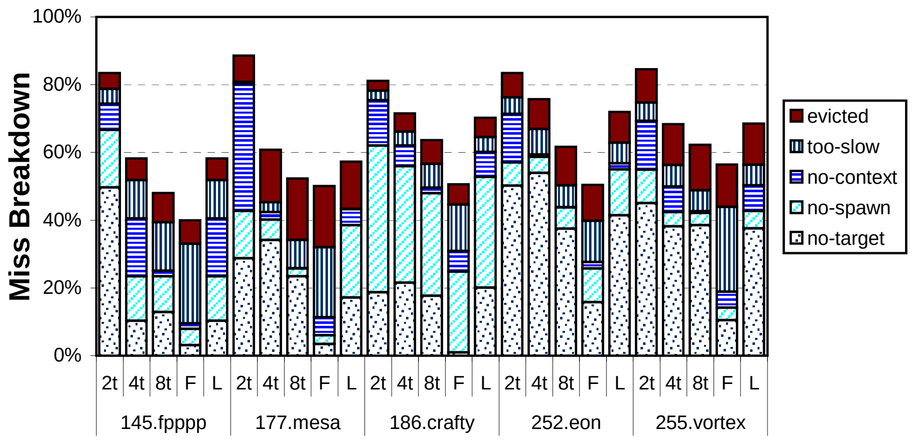


Figure 3.6: Breakdown of remaining I-cache misses relative to baseline.

committed; "too slow" means a helper thread was running that could have prefetched the I-cache miss, but it was caught by the main thread before it could do so (note: partial misses are included in this chart); finally, "evicted" means a helper thread did prefetch the accessed line, but the line was evicted before being reached by the main thread.

Ideally, all components should be small, but especially the "too slow" and "no contexts" should be low if our framework is able to predict run time behavior well and the algorithm is properly tuned. On average, for 4t, 32% of I-cache misses remain because there was no target, 12% because there was no spawn, 7% due to lack of SMT thread contexts, 6% because the associated helper threads ran too slowly, and 9% because the prefetches were too aggressive and prefetched instructions were evicted. The large "no target" component results when the selection algorithm could not find a spawn-target pair to target the miss, either due to resource constraints, or because it could not find an early enough location that was control flow correlated because we constrained spawn-target pairs to be within the same function body. Figure 3.6 indicates that increasing the number of thread resources (and static spawn-target pairs) does not seem to significantly impact "no target". So it is more likely the latter cause is the culprit. To further corroborate this view, we examine the bar labeled F in Figure 3.5 and Figure 3.6, which represents the impact when spawn-target pairs are selected assuming the helper thread progresses with the same CPI as the main thread (and the simulator is modified to simulate helper thread instruction fetches as non-blocking). This reduces the "no target" component to an average 7%, highlighting the importance of finding ways to improve helper thread efficiency (e.g., reduce resource requirements per instruction prefetch generated by reducing precomputation overhead). Note that `vortex` shows significant speedups due to data prefetching for all models except F (where the associated execution model eliminates the data prefetching side effect of load instructions in the postfix region).

For `crafty`, there was a large fraction of "no spawn". To reduce this, a seemingly straightforward approach is to lower the threshold for reaching probability in hope to include more spawn candidates. To quantify this intuition, the bars labeled L are for spawn-target pairs selected with a reaching probability threshold of only 75%. However, this lowered threshold does not seem to help significantly. Detailed inspection shows many targets associated with this type of miss have spawns with low posteriori probability. Often these were the only remaining locations within the current procedure that provided sufficient slack. Thus, lowering selection threshold alone is not sufficient, further improvements may come from a better (i.e., non-greedy) selection algorithm.

3.4 Summary

In this chapter the notion of prescient instruction prefetch was introduced. The framework introduced in Chapter 2 was applied to a simple model of prescient instruction prefetch and an optimization algorithm was implemented. The resulting helper threads were evaluated assuming an idealized execution model and shown to provide a harmonic mean speedup of 13% over a system without helper threads (but including conventional instruction prefetch hardware) on a subset of SPEC benchmarks suffering signficiant instruction cache misses. This improvement demonstrates the ability of the framework to effectively predict program behaviour from the perspective of higher-level control flow so that it can select good spawn-target pairs for prescient instructin prefetch helper threads.

The idealized execution model excludes the overhead of pre-executing the infix slice, and includes the overhead of pre-executing more instructions from the postfix region than are strictly required to accurately anticipate the postfix control flow. In the next chapter two realistic implementation techniques for prescient instruction prefetch are proposed and evaluated, including the overhead of infix pre-execution, along with several hardware support mechanisms including two aimed at reducing the overhead of postfix pre-execution. Furthermore, these realistic implementation techniques are compared against a very aggressive hardware-only research instruction prefetch mechanism.

Chapter 4

Hardware Support for Prescient Instruction Prefetch

This chapter describes two feasible implementations of prescient instruction prefetch and proposes associated hardware support mechanisms. When supplied with prescient instruction prefetch enabled application software, these hardware mechanisms yield significant performance improvement while remaining complexity-effective. While the potential benefit of prescient instruction prefetch was demonstrated in Chapter 3 via a limit study, the techniques presented in this chapter show how to implement prescient instruction prefetch under realistic constraints. In addition, the performance scalability of prescient instruction prefetch as memory latency scales up (relative to core processor cycle time) is evaluated in comparison to aggressive hardware-only I-prefetch mechanisms. It is demonstrated that prescient instruction prefetch may yield similar or better performance improvements than the most aggressive hardware-based instruction prefetch technique, by leveraging existing SMT resources without large amounts of additional specialized hardware.

The rest of this chapter is organized as follows: Section 4.1 presents a straightforward technique for implementing prescient instruction prefetch called *direct pre-execution,* along with hardware mechanisms supporting it; Section 4.2 introduces an enhanced technique for implementing prescient instruction prefetch called *finite state machine recall,* which also leverages hardware support for a counted instruction prefetch operation and precomputes only hard to predict branches; Section 4.3 presents a performance evaluation of both techniques; and Section 4.4 summarizes the contributions of this chapter.

4.1 Direct pre-execution

In this section a straightforward implementation of prescient instruction prefetch called *direct pre-execution* is described. During direct pre-execution, instructions from the main thread's postfix region are prefetched into the first-level I-cache by executing those same instructions on a spare SMT thread context. In this section requirements for effective direct pre-execution are examined and hardware mechanisms supporting these requirements are described.

4.1.1 Requirements for direct pre-execution

To begin, several significant challenges to obtaining effective direct pre-execution are highlighted.

Constructing precomputation slices.

For direct pre-execution to correctly resolve postfix branches, the outcome of the backward slice of each postfix branch must be accurately reproduced. This slice may contain computations from both the infix and postfix regions. Thus, as shown in Figure 3.1(b), direct pre-execution consists of two phases: The first phase, live-in precomputation, reproduces the effect of the code skipped over in the infix region that relates to the resolution of branches in the postfix region. We refer to these precomputation instructions as the infix slice. Similar to speculative precomputation [CWT$^+$01, LWW$^+$02], infix slices for direct pre-execution helper threads could be encoded as additional instructions embedded in a program's binary image. In the second phase, the helper thread executes the remaining slice operations while executing the postfix region. As instructions in the infix region not related to any branch instruction in the postfix region are not duplicated in the infix slice they are neither fetched, nor executed by the helper thread which enables the helper thread to start executing through the postfix region before the main thread.

To construct infix slices, we consider dynamic slices similar to those studied in previous work [AH90, ZS00]. A dynamic slice can be decomposed into value, address, control and existence sub-slices [ZS00]. In the present context, the *value slice* refers to those operations that compute values directly used to determine the outcomes of branches in the postfix region. The *address slice* includes those operations that compute load and store addresses in the value slice to resolve store-to-load dependencies between memory operations in the value slice. Likewise, the *control slice* resolves the control dependencies of the value and address slices. Lastly, the *existence slice* is the portion of the control slice that determines whether the target is reached. Instructions may be

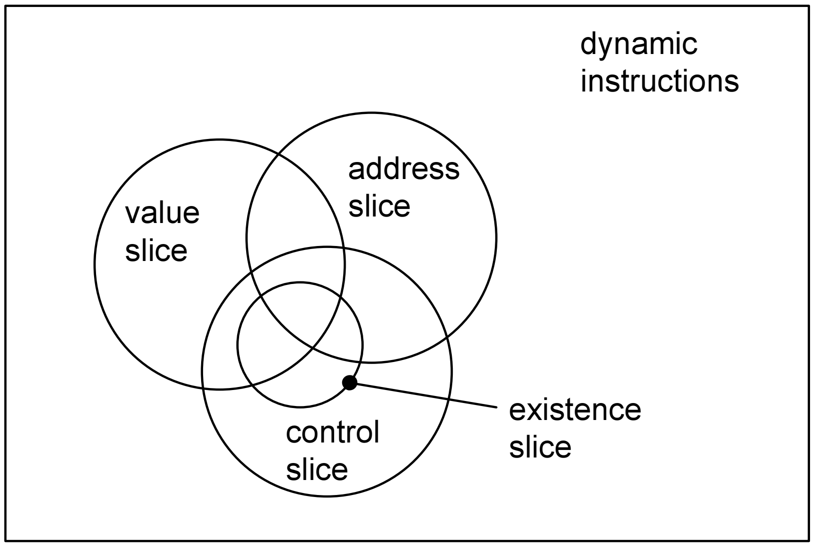


Figure 4.1: Venn diagram showing relationships of various dynamic slice classifications.

in more than one type of slice as illustrated in Figure 4.1.

Similar to Zilles and Sohi [ZS00], in this study we examine variable infix-slices, extracted per dynamic spawn-target instance from the control-flow through the infix and postfix region, assuming perfect memory disambiguation, and containing the value, and address sub-slice, as well as the subset of the control sub-slices that generate instruction predicates for computation in the value and address slice[1].

Handling stores in precomputation threads.

Helper threads run ahead of the main thread. In the postfix region helper threads execute the same code as the main thread and will inevitably encounter store instructions. These stores should not be executed the same way as in the main thread, but they cannot simply be ignored: On the one hand, allowing stores from a helper thread to commit architecturally could violate program correctness (e.g. a load from the main thread may end up erroneously depending on a later store committed by a helper thread). On the other hand, as shown in Section 4.3, store-

[1]It was observed that typically, a large fraction of dynamic infix slices tend to contain the same sequence of static instructions. A more detailed quantitative evaluation of the potential overhead of control slice instructions from the infix region for those control slice branches in the infix region that both (a) remain after if-convertion, and (b) are otherwise hard to predict, is beyond the scrope of this dissertation.

```
<spawn>
...
<target>
S1       cmp p3 = r1,r2 // r2 arbitrary
S2  (p3) add r1 = ...
S3*      cmp p5 = r3,r1
S4* (p5) br X
```

The code after <target> is the postfix region for a helper thread started when the main thread reaches <spawn>. p3 and p5 are predicate registers controlling whether S2 and S4 are executed. Mnemonics are “cmp” for compare, “add” for addition, “br” for branch (if predicate is true).

Figure 4.2: Postfix pre-execution and filtering

to-load communication can indeed be part of the slice required to resolve control flow through the postfix region. To satisfy these conflicting requirements a mechanism called the *safe-store* is described in Section 4.1.2.

Handling ineffectual instructions.

To ensure the effectiveness of prescient instruction prefetch, helper threads should avoid executing postfix instructions that are not related to the resolution of postfix branches. These instructions are called *ineffectual instructions*[2]. Although computing these instructions may potentially bring beneficial side effects such as data prefetching (if data addresses are computed correctly), they could waste processor resources. Worse yet, these ineffectual instructions may even corrupt the computation responsible for resolving postfix branches if they are not filtered out.

For example, Figure 4.2 shows four instructions from a postfix region where asterisks are used to indicate operations in the slice leading to the resolution of a branch. Statement S2 is found to be predicated “false” most of the time (so register `r1` is usually not updated at S2). Hence, the dynamic slice algorithm considered statements S1 and S2 ineffectual and excluded them from the slice, however, executing S1 and S2 may cause the helper thread to diverge from

[2]Sudaramoorthy et al. [SPR00] use this term to describe those dynamic instructions that do not impact the results of program execution.

the main thread's future path: As `r2` contains an arbitrary value at S1, S2 may (incorrectly) be predicated "on" by the helper thread, resulting in an update to `r1`. This update to `r1` may result in S3 computing the wrong value for `p5` leading to control flow divergence after S4[3].

To filter out ineffectual computation a novel mechanism called the *YAT-bit* is introduced in Section 4.1.2 that would allow only S3 and S4 to execute in the preceding example.

4.1.2 Hardware support

In this section, we describe hardware mechanisms to support direct pre-execution.

Helper thread spawning.

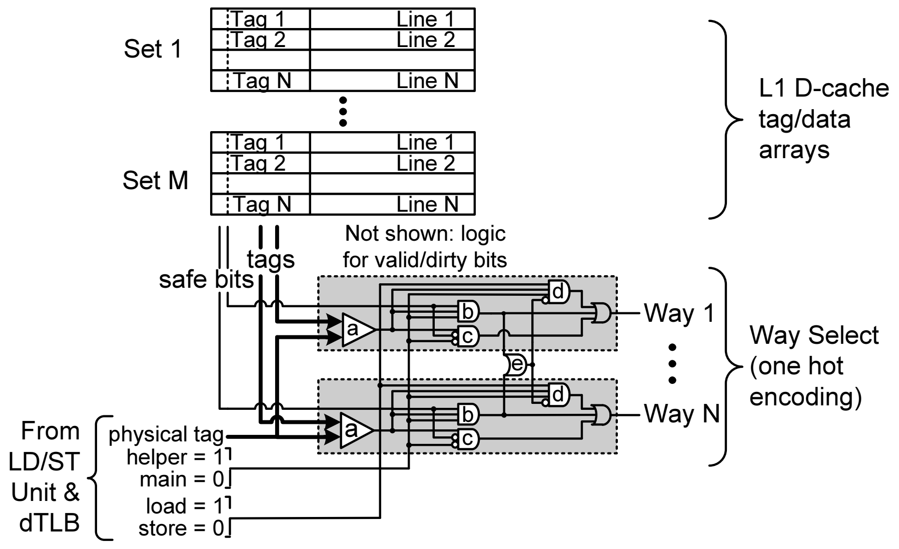
When the main thread commits a spawn-point, a helper thread is spawned if there is an idle thread context. The helper thread first reads the set of live-in register and memory values required for infix-slice precomputation from the main thread. We consider a register or memory value to be *live-in* to a region if the value is used before being defined in that region. The infix slice is responsible for computing the set of live-in register and memory values relevant to branch resolution in the postfix region. Once infix slice precomputation completes, the helper thread jumps to the target and starts executing the main thread's code in the postfix region.

Safe-stores.

To honor store-to-load dependencies during helper-thread execution without affecting program correctness we propose *safe-stores*—a simple microarchitectural mechanism for supporting a type of speculative store. Store instructions encountered during helper thread execution are executed as safe-stores. A safe-store is written into the first-level D-cache, but cannot be read by any other non-speculative thread. This selectivity is achieved by extending the cache tags with a safe-bit to indicate whether or not the line was modified by a helper thread.

To implement safe-stores the tag match logic is extended to include comparison of the safe-bit as shown in Figure 4.3. The safe-bit is initially cleared for cache lines brought in by the main thread but is set when a line is modified by a helper thread. The augmented tag matching logic guarantees that a load from the main thread will never consume data produced by a store in

[3]The problem illustrated is not unique to predication: Executing ineffectual stores can also lead to control flow divergence.



Way selection enabled if: (a) conventional tag match found; and: (b) line has safe-bit set and access is from helper thread; or (c) line does not have safe-bit set and access is from main thread; or (d) single matching line does not have its safe bit set, and access is from helper thread.

Figure 4.3: Safe-store way-selection logic

a helper thread, thus ensuring program correctness. However, upon a D-cache hit, a load from the helper thread is allowed to speculatively consume data stored by the main thread or another helper thread.

For example, a safe-store modified line may be evicted from the D-cache by a store from the main thread with the same tag, which is then followed by a dependent load from the helper thread that hits in the cache and receives the data written by the main thread (we call this condition a *safe-store eviction*). Hence it cannot be guaranteed that the data consumed was produced by the safe-store upon which it logically depends. Similarly, when multiple helper threads operate concurrently, they may (erroneously) read values stored by each other (we call this *safe-store aliasing*). If multiple helper threads often write to the same lines concurrently, performance may improve if safe-store modified lines are further distinguished by adding a couple of helper thread instance specific tag bits in addition to the safe-bit.

After a helper thread exits, a safe-store modified line may remain in the cache long enough

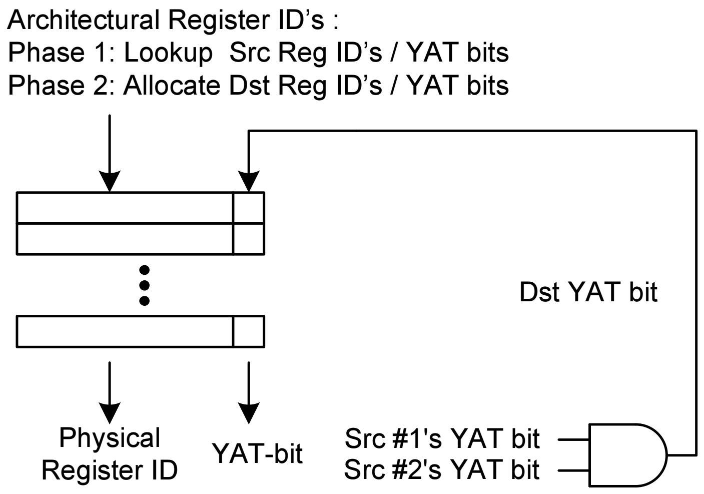

Architectural Register ID's :
Phase 1: Lookup Src Reg ID's / YAT bits
Phase 2: Allocate Dst Reg ID's / YAT bits
Dst YAT bit
Physical Register ID
YAT-bit
Src #1's YAT bit
Src #2's YAT bit

Figure 4.4: YAT-bit rename logic

that a subsequent helper thread reads a "stale" value from it[4]. One way of reducing the chance of this happening is to mark loads that read helper thread live-in values so they invalidate matching safe-store modified lines. As shown in Section 4.3, we found the safe-store mechanism can frequently enforce the correct store-to-load dependency within a helper thread.

If the D-cache uses a write-through policy, the line modified by a safe-store is prevented from being written into the next level cache hierarchy. If the D-cache instead uses a write-back policy, a line with safe-bit set behaves as an invalid line when it participates in a coherence transaction. Furthermore, when using a write-back policy, a write-back of non-speculative state must be generated when a helper thread writes to a line modified by the main thread before that line may be modified by the helper thread and have its safe-bit set. When a safe-store modified line is evicted it is not written back to the next level of the memory hierarchy. Safe-store modified lines may reduce performance by increasing the number of D-cache misses incurred by the main thread, and by triggering additional write-backs. Note that the relaxed correctness requirements imposed by helper threads relative to speculative multithreading enable the safe-store mechanism to be much simpler than the *speculative versioning cache* proposed for use in implemeting speculative multithreading [GVSS98], and the extensions of invalidation based cache coherence protocols proposed for *thread level speculation* [SM98, SCZM00].

[4]In this context the term "stale" means the line is from a different helper thread instance executing on the same processor core (in this research we focus the use of helper threads on an SMT processor), rather than a line invalidated due to a modification being made in the local L1 cache of a distant processor core.

YAT-bits.

To avoid executing ineffectual instructions, we introduce the *YAT-bit*[5], a simple extension to existing register renaming hardware. Register renaming is used in out-of-order processors to eliminate false dependencies resulting from reuse of architectural registers. Similarly, a form of register renaming is used to reduce function call overhead, via register windowing, and software pipelining overhead, via register rotation, in the Itanium® architecture [HMR+00]. In the proposed extension each architectural register is associated with an extra bit, called the YAT-bit that indicates whether the corresponding physical register likely contains the same value as the main thread will when it reaches the point where the helper thread is currently executing. Upon spawning a helper thread all YAT-bits for the helper thread are cleared. After copying live-in registers from the main thread, the YAT-bit for each live-in register is set to indicate the content is meaningful, then precomputation of the infix slice begins. YAT-bits propagate as follows: When an instruction enters the rename stage, the YAT-bits of each source operand are looked up. If all source YAT-bits are valid, the destination register YAT-bit(s) will also be marked valid, and the instruction will execute normally. However, if any source register YAT-bits are invalid, the instruction's destination register YAT-bits will be invalidated, and the instruction will be treated as a "no-op" consuming no execution resources down the rest of the pipeline (see Figure 4.4)[6].

The hardware support used to implement the YAT-bit mechanism is reminiscent of that used to support runahead execution [DM97, Dun98, MSWP03]. In runahead execution first proposed by Dundas [DM97, Dun98] the processor continues to prefetch data after a cache miss would otherwise stall the processor. The form of runahead execution proposed by Mutlu and Patt [MSWP03] assumes an out-of-order superscalar processor microarchitecture similar to the Intel Pentium 4, and they propose that the register-renamer be resposible for tracking "invalid registers" (those registers dependent upon a value produced by a load that misses in the L2-cache), whereas Dundas proposed [DM97] that a backup copy of the register file be keep when the processor enters runahead mode. YAT-bit operation fundamentally differs in that an invalid YAT-bit indicates a register is dependent upon a live-in that was not copied from the main thread. An invalid YAT-bit therefore indicates that the value is both inaccurate and likely irrelevant when

[5]YAT-bit stands for "yes-a-thing bit" in analogy to the Itanium® NAT-bit ("not-a-thing bit") used for exception deferral.

[6]The infix slice may leave stale temporary register values—therefore, clearing their YAT-bits before postfix precomputation improves prefetch accuracy.

resolving postfix branches.

It is interesting to note a few concerns specific to architectures supporting predication and/or register windows. If an instruction is predicated “false”, the destination registers are not updated and hence the associated YAT-bits should remain unchanged. If the predicate register YAT-bit is invalid the correct value of the predicate is unknown and hence it may be unclear whether the destination YAT-bits should change. In practice we found it was effective to simply let the destination register YAT-bits remain unchanged.

On architectures using register windowing, a register stack engine typically spills and fills the values in registers to/from memory automatically. If this mechanism is triggered during helper thread execution, saving and restoring YAT-bits can improve prefetch accuracy and therefore may improve performance.

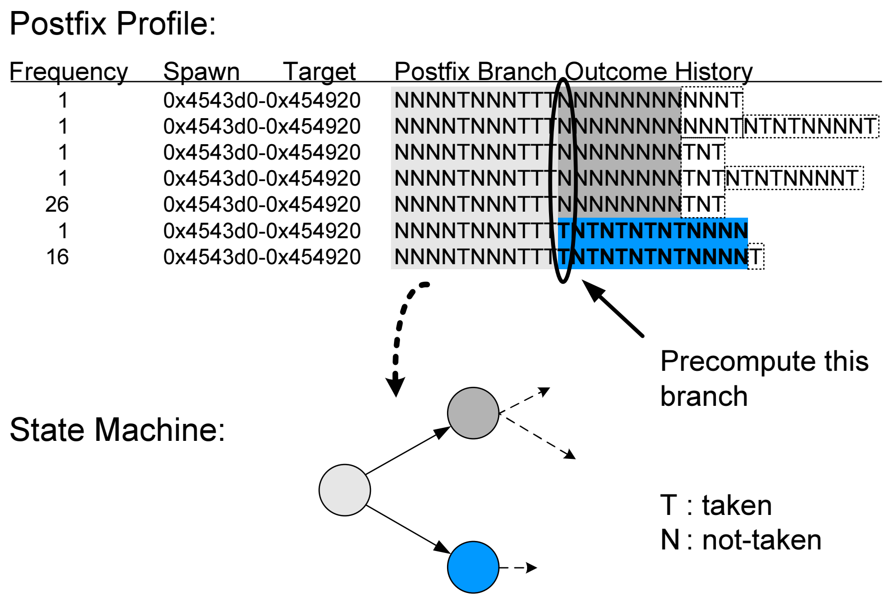
Helper thread kill.

To ensure helper threads do not run behind the main thread or run astray, a helper thread is terminated when the main thread catches it (in practice–when both threads having the same next fetch address), or when a maximum number of postfix operations (predetermined during spawn-target pair selection) have executed.

4.2 Exploiting predictability

In this section we examine a technique for reducing the amount of precomputation required for achieving prescient instruction prefetch. Typically, many postfix branches are either strongly biased, thus predictable statically, or dynamically predictable with high confidence using a modern hardware branch predictor. Hence, using branch prediction can reduce the amount of precomputation required to ensure accurate instruction prefetch.

An interesting alternative to the direct pre-execution technique uses a counted I-prefetch operation encoded in the processor’s instruction set to decouple I-prefetch from precomputation. The precomputation slices used by this technique, called *transition slices* (or *t-slices*), merely resolve weakly-biased branches. A counted I-prefetch operation provides the processor with a starting address, and number of lines to prefetch as a hint (i.e., the action may be ignored without affecting program correctness). A hardware prefetch engine queues these instruction prefetch requests, and issues them to the memory system at a later time. The counted I-prefetch



Frequency	Spawn Target	Postfix Branch Outcome History
1	0x4543d0-0x454920	NNNNTNNNTTTNNNNNNNNNNNNT
1	0x4543d0-0x454920	NNNNTNNNTTTNNNNNNNNNNNNTNTNTNNNNT
1	0x4543d0-0x454920	NNNNTNNNTTTNNNNNNNNNTNT
1	0x4543d0-0x454920	NNNNTNNNTTTNNNNNNNNNTNTNTNTNNNNT
26	0x4543d0-0x454920	NNNNTNNNTTTNNNNNNNNNTNT
1	0x4543d0-0x454920	NNNNTNNNTTT**TNTNTNTNTNNNN**
16	0x4543d0-0x454920	NNNNTNNNTTT**TNTNTNTNTNNNN**T

Figure 4.5: Postfix profile & FSM construction

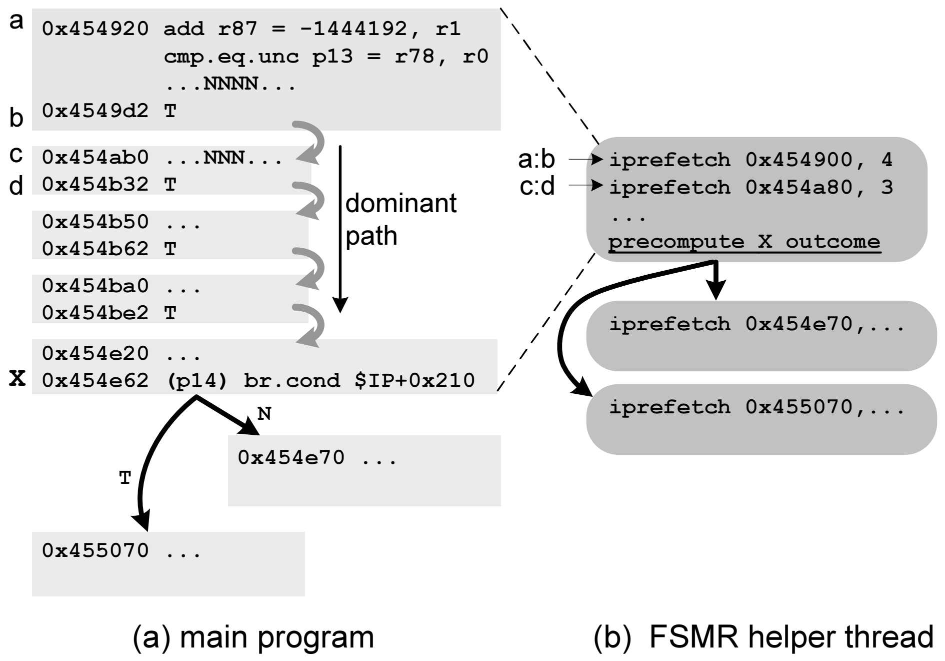
operations we envision resemble the I-prefetch hints encoded in the `br.many` and the `br.few` instructions of the Itanium® architecture, except the hint would also express the number of lines to be prefetched.

The following section describes how to implement prescient instruction prefetch helper threads that use counted instruction prefetches.

4.2.1 Finite state machine recall

The control flow paths through a postfix region alternate between path segments containing strongly-biased branches (which may be summarized with a sequence of counted prefetches) separated by weakly-biased branches. For example, the top half of Figure 4.5 shows profile information (from 186.crafty) for a single spawn-target pair's postfix region illustrating that the target is followed by 10 strongly-biased branches ending at a weakly-biased branch. These dominant path segments can be summarized as states in a finite state machine in which transitions represent the resolution of weakly-biased branches (bottom half of Figure 4.5).

In *finite state machine recall* (FSMR) a t-slice is constructed for each of these weakly-biased branches. If some of these weakly-biased branch could be predicted accurately via some hardware mechanism then an instruction that queries the predictor could be used instead of a t-slice.



Counted prefetch "iprefetch x,n" triggers a hardware counted prefetch engine to prefetch n cache lines starting from address x (note: the lower 6-bits of x are zero as they represent the offset within a 64-byte cache block). Underlined statement in part (b) is a t-slice.

Figure 4.6: Finite state machine recall example

Figure 4.6 shows an FSMR helper thread example. In Figure 4.6(a) the block starting at 'a' and ending at 'b' is merged together with the block starting at 'c' and ending at 'd' into the first state in the FSMR helper thread because the branch at 'b' is strongly biased to be taken. Figure 4.6(b) shows a fragment of the finite state machine encoding the region shown in part (a). Note that only non-contiguous blocks of code require separate counted prefetch operations.

In contrast to the infix slice used in direct pre-execution, the code in a t-slice may include copies of computation from both the infix and the postfix region. If the outcome generated by a t-slice does not match one of the transitions in the state machine, the helper thread exits. Note FSMR helper threads may benefit from using safe-stores to enable store-to-load communication.

Since the instructions prefetched by any state are independent of the t-slice outcome, we scheduled counted prefetches first to initiate prefetches for code in the main thread before precomputing a t-slice to determine the next state transition. Counted prefetch memory accesses continue in parallel with the execution of the subsequent t-slices. A more aggressive implementa-

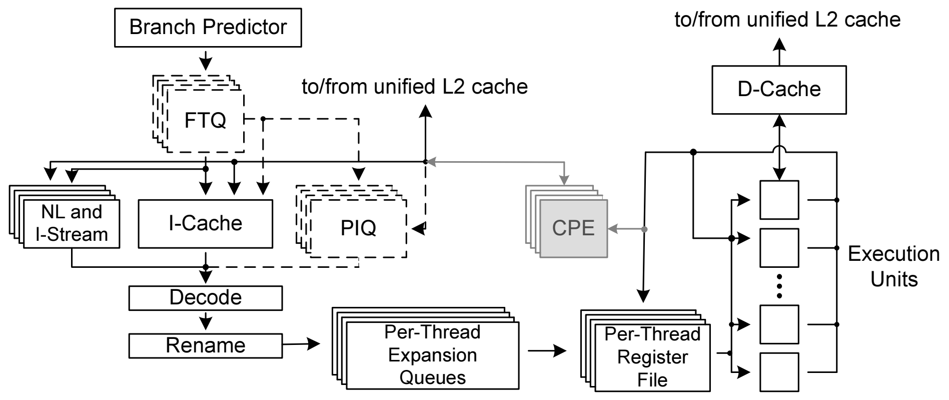


Figure 4.7: Processor model (FTQ and PIQ for FDIP only, CPE for FSMR only)

tion might spawn t-slices speculatively based upon less reliable control flow prediction techniques (cf. chaining speculative precomputation [CWT$^+$01] where essentially the prediction is that a backward branch is taken), or reuse common sub-computation from one t-slice to the next. In the extreme case, provided there are a sufficient number of idle thread contexts and fetch bandwidth, from one single state multiple t-slices can be eagerly spawned for multiple successor states.

Our simulator models a modest per-thread context decoupled counted prefetch engine. When a counted prefetch reaches the execute stage of the pipeline the counted prefetch is inserted to the back of the queue for the associated thread context. Prefetch requests are issued from this queue upon available bandwidth to the L2 cache.

4.2.2 FSM and t-slice construction

To construct the finite state machine and t-slices, the profile guided spawn-target pair selection algorithm [AMC$^+$03] is used to supply spawn-target pairs. Spawn-target pairs for FSMR are selected assuming the number of instructions prefetched per cycle by a helper thread allows it to keep the same pace as the main thread. For each pair, postfix path profiling is applied to the postfix region and the results of this postfix path-profile are then used to construct a separate FSM for each static spawn-target pair as described in Section 4.2.1. The t-slices are extracted similarly to dynamic slices. The data in Table 4.2 shows that these finite state machines are typically small. For example, the number of t-slice transitions per dynamic helper thread instance ranges from 0.39 to 6.8 (see Row 15 in Table 4.2), even though the number of branches encountered varies from 1.53 to 47 (Row 8).

4.3 Performance evaluation

In this section we evaluate the performance of the proposed techniques.

4.3.1 Processor model

We model a research Itanium® SMT processor with four hardware thread contexts based upon SMTSIM [TEL95] (see Figure 4.7). The processor configurations listed in Table 4.1 are modeled. To gauge performance scalability with respect to future memory latencies, we evaluate two memory hierarchies, labeled 2x and 4x, representative of 2GHz and 4GHz Itanium® processors, respectively.

The baseline configuration includes the following I-prefetch support: Upon a demand miss, a request is generated for the next sequential cache line. In addition, Itanium® instruction prefetch hints (e.g., `br.many`) are used to trigger a streaming prefetch address generator that can send additional requests up to four cache lines ahead of the fetch address. If an instruction prefetch request encounters a second-level iTLB miss that requires accessing the page table in main memory, the request is ignored without generating an iTLB update. Instruction cache miss requests are prioritized in the memory hierarchy. Demand fetches have highest priority, followed by next line and streaming prefetches, followed by counted prefetch requests. For FDIP configurations the Itanium® stream I-prefetch mechanism is not used.

In our processor model, helper threads contend for fetch and issue bandwidth with the main thread, but have lower priority. For I-cache access, each cycle the fetch unit can access two cache lines. If the main thread is currently waiting for an outstanding I-cache miss, up to two active helper threads are selected (in round-robin order) to access the I-cache. Otherwise, one access from the main thread, and one from an active helper thread are allowed.

During instruction issue, ready instructions are selected from the main thread provided there is availability of function units. Any left over issue bandwidth is available for helper thread instructions.

4.3.2 Prescient instruction prefetch models

We model both direct pre-execution and FSMR mechanisms for prescient instruction prefetch.

Direct pre-execution.

For direct pre-execution we model both an idealized version that assumes *perfect live-in prediction* (PLP) and a more realistic version that uses dynamic infix slices to perform precomputation, called *variable slice* (VS). PLP models the ideal case where helper threads begin executing at the target as soon as the spawn is encountered by the main thread and initially see the same architectural state that the main thread will be in when it also reaches the target. As in the earlier limit study [AMC$^+$03], for PLP an infinite store buffer model is assumed (rather than the safe-store mechanism), and a helper thread is triggered when the main thread commits a spawn-point (if no contexts are available the spawn-point is ignored). The helper thread begins fetching from the target the following cycle and runs the maximum number of instructions, or until it is caught by the main thread, at which point it exits. For the VS model, after a helper thread is spawned it first pre-executes the infix slice and only afterwards jumps to the target-point to start executing (and thereby effectively prefetching) instructions in the postfix region. For VS, safe-stores are modeled with a 2-bit tag that reduces interference between concurrent helper threads (increasing D-cache misses incurred by the main thread). If a helper thread load, known to read a live-in value, hits a safe-store modified line, it invalidates the line and triggers a D-cache fill of non-speculative state, or reads the non-speculative value if it is already in the D-cache. When the main thread catches up with a helper thread (has matching next instruction fetch address), that helper thread stops fetching instructions. Once the helper thread's instructions drain from the pipeline, the thread context is available to run other helper threads.

The PLP model serves to gauge the upper bound for the direct pre-execution approach. In contrast, the VS model serves to evaluate the impact of live-in precomputation overhead; the performance penalty of safe-stores due to increased misses in the D-cache; the ability of YAT-bit filtering to reduce resource contention, and prevent ineffectual instructions from corrupting postfix precomputation; and the relatively less D-cache data prefetching side effects compared to PLP when fewer loads are executed by helper threads.

FSMR.

For FSMR, counted prefetch requests are queued into a FIFO buffer, and eventually issued to the memory system (if no buffers are available, new requests are discarded). Similar to direct pre-execution, counted prefetches issued via FSMR are allocated right into the I-cache rather than a prefetch buffer. Safe-stores are modeled as with VS, but without additional tag bits.

4.3.3 Simulation methodology and workloads

We selected six benchmarks from SPEC2000 and one from SPEC95 that incur significant instruction cache misses on the baseline processor model. For spawn-target pair selection we profiled branch frequencies and I-cache miss behavior by running the programs to completion. To evaluate performance we collect data for five million instructions starting after warming up the cache hierarchy while fast-forwarding past the first billion instructions assuming no prescient instruction prefetch. For FSMR, postfix path profiling was also performed over this simulation window. It was found that longer simulations employing the VS and FSMR models (up to 100 million instructions in duration) did not exhibit significant IPC variation for the workloads employed in this dissertation. Furthermore, the simulation models employed for this part of the dissertation were far too slow to make significant experimentation with longer simulations practical in the time available.

Our spawn-target generation algorithm selected between 34 and 3545 spawn-target pairs per benchmark as shown in Row 1 of Table 4.2, which also quantifies several characteristics of the slices executed for the VS and FSMR configurations. The small number of static spawn-target pairs implies a small instruction storage requirement for prescient instruction prefetch helper threads. Hence, we model single-cycle access to infix slice and t-slice instructions, and assume they can be stored in a small on-chip buffer.

Comparing the average spawn-target distances (i.e., the average number of instructions executed by the main thread between spawn, and target) in Row 3 of Table 4.2 with the data in Figure 3.2 for FDIP highlights the fact that prescient instruction prefetch has the potential to provide significantly larger slack. For the benchmarks we study, most I-cache misses are satisfied from the low latency L2 cache so they do not show substantial benefit from this larger slack (i.e., applications with larger instruction footprints than those we study may obtain larger benefits from prescient instruction prefetch).

The number of infix slice instructions averaged between 4.9 and 35 (Row 6), which is comparable in size to those required for speculative precomputation of load addresses for data prefetching [CWT$^+$01, MPB01], and implies prescient instruction prefetch can start executing through the postfix region long before the main thread will reach the target. On average the number of store-to-load dependencies within an infix slice was less than three (Row 7). Furthermore,

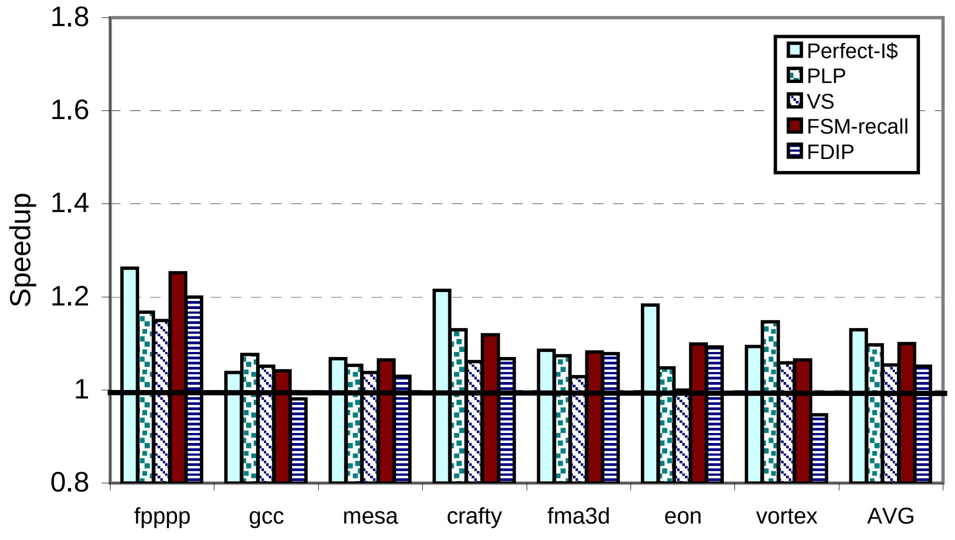


Figure 4.8: Performance on 2x configuration

very few safe-store evictions, or safe-store alias errors occur for the 4-way set associative D-cache we modeled. For FSMR there were almost no safe-store evictions, and less than 1.54% of helper threads encountered a safe-store alias error. For VS less than 5.9% of all helper threads encounter a safe-store eviction, and fewer than 1.22% of helper thread instances encounter a safe-store alias error when using a 2-bit tag in addition to the safe-bit mechanism (Row 12). (Row 11 shows the increase in evictions and aliasing that occur for VS using safe-bits without additional tag bits.)

The average postfix region sizes were fairly large, ranging from 85 to 239 instructions (Row 4). The number of branches in a postfix region can be quantified both in terms of the number of static branches it contains, and by the number of dynamic branch instances executed each time a helper thread executes through it. Averages for both are given in Row 8, and indicate most benchmarks are control flow intensive in addition to having poor I-cache locality.

For FSMR the total number of slice instructions per helper thread instance tends to be larger than for VS, however, FSMR t-slices may also include operations from the postfix region. Furthermore, slice operations from one branch precomputation may intersect the set of slice operations from a subsequent branch. For FSMR the average number of branch outcomes precomputed per helper thread instance is between 0.5 and 6.8 (Row 15), which is significantly lower than the number of branches encountered in the postfix region per helper thread instance.

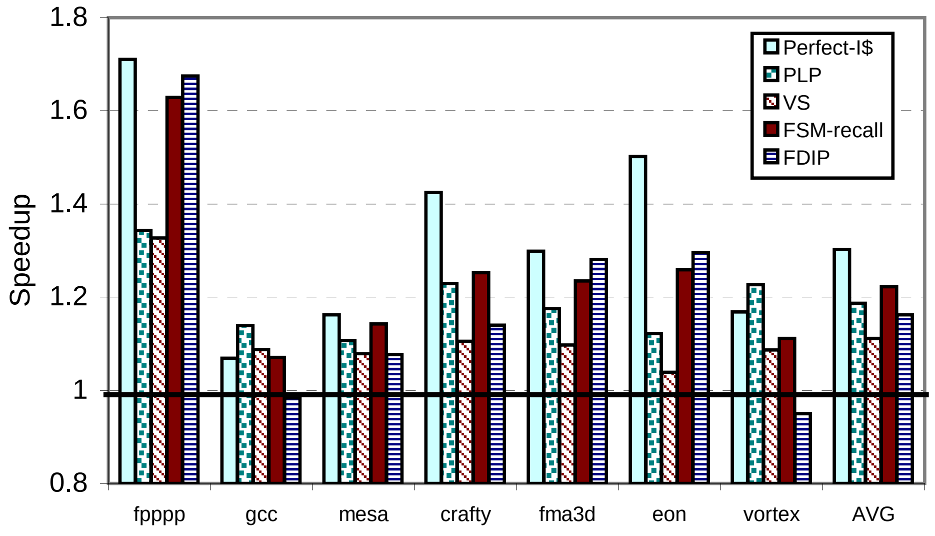


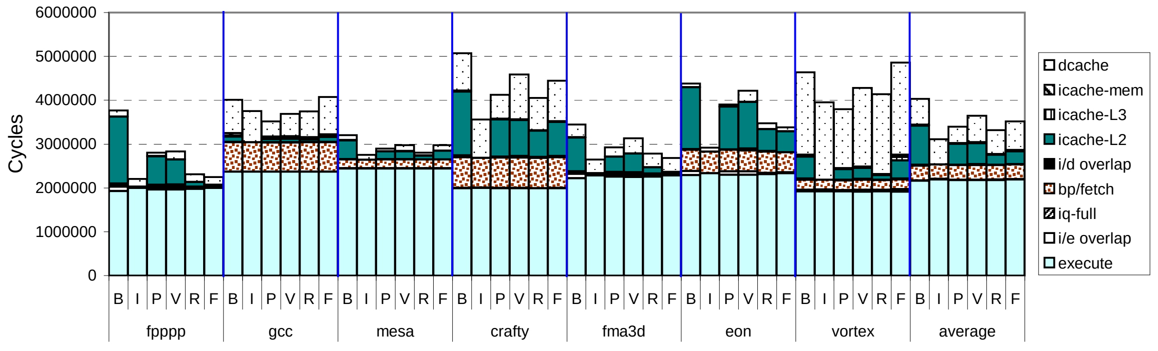
Figure 4.9: Performance on 4x configuration

4.3.4 Performance evaluation

Figures 4.8 and 4.9 display the performance for the 2x and 4x memory hierarchy configurations described more fully in Table 4.1. Each figure shows five bars per benchmark. The first bar from the left, "Perfect I$", represents the speedup obtained if all instruction accesses hit in the I-cache. The next few bars, labeled "PLP", "VS", and "FSMR" represent the speedup of the various models of prescient instruction prefetch described earlier. The bar labeled "FDIP" shows the performance of our implementation of Reinman *et al.*'s fetch directed instruction prefetch mechanism using the configuration described in Table 4.1.

On the 2x memory configuration, PLP obtains a harmonic mean speedup of 9.7%, VS obtains a 5.3% speedup, FSMR obtains a 10.0% speedup, and FDIP obtains a 5.1% speedup. Similarly, on the 4x memory configuration PLP obtains a harmonic mean speedup of 18.7%, VS obtains a 11.1% speedup, FSMR obtains a 22% speedup, and FDIP obtains a 16.2% speedup. The data shows that prescient instruction prefetch can obtain significant speedups over our baseline model and in particular, FSMR is competitive with, and often outperforms fetch directed instruction prefetch.

To provide insight into this result Figure 4.10 provides a breakdown of how execution cycles are spent, and Figure 4.11 analyzes the cause of remaining I-cache misses. As shown in Figure 4.10, prescient instruction prefetch outperforms "Perfect-I$" on gcc and vortex (for PLP) due to data



B=baseline, I=Perfect-I$, P=PLP, V=VS, R=FSMR, and F=FDIP.

Figure 4.10: Classification of execution cycles (4x memory configuration).

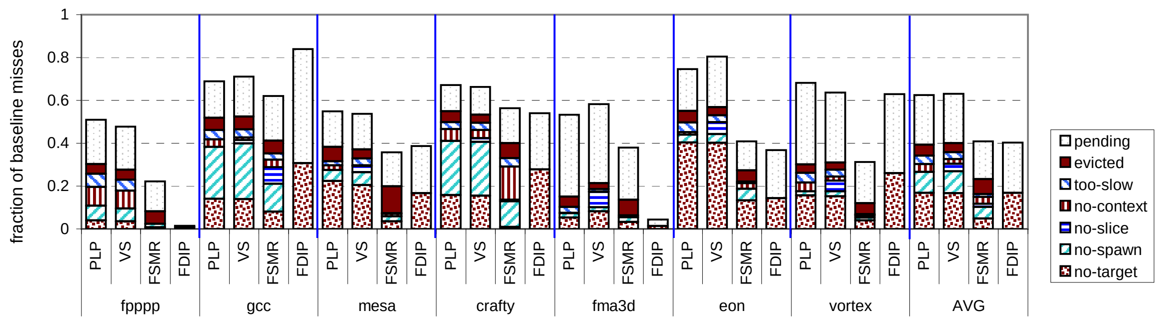


Figure 4.11: Classification of remaining I-cache misses (4x memory configuration)

prefetching side effects.

Figure 4.10 shows the total execution time broken down into nine categories: *d-cache* represents cycles lost to D-cache misses; *icache-L2, icache-L3* and *icache-mem* represent cycles spent waiting for instructions to return from the associated location in the cache hierarchy; *i/d overlapped* and *i/e overlap* represents I-cache stall cycles overlapped with D-cache and execution cycles; *bp/fetch* represents cycles lost due to branch mispredictions, misfetched branch targets, taken branches, and crossing cache line boundaries; *iq-full* represents cycles lost due to the need to refill the front-end of the pipeline after the expansion queue overflows; finally, *execute* represents the number of cycles required given available function units and the schedule produced by the compiler.

On the baseline configuration each benchmark spends an average of about 22% of total execution time waiting for instruction fetch requests to return from L2, which drops to 14% for

PLP, 13% for VS, 7% for FSMR, and 8% for FDIP. Hence, all of the mechanisms yield performance gains by reducing stall cycles due to I-cache misses. However, the prescient instruction prefetch techniques also reduce execution time by prefetching some data. For example FSMR reduced D-cache stall cycles on gcc, mesa, crafty and vortex. Most notably, 177.mesa has a 42% reduction in D-cache stall cycles for FSMR, translating into a 1.6% speedup, and 176.gcc has a 33% reduction in D-cache stall cycles for VS, translating into a 6.3% speedup. Even larger gains are seen for PLP because all loads in the postfix region are executed and helper thread stores do not write to the D-cache. On the other hand, some benchmarks see an increase in data cache stall cycles for the VS and FSMR configurations. This increase is due to safe-stores increasing the number of D-cache misses seen by the main thread as described in Section 4.1.2. For FDIP, the D-cache stall cycle component increases by an average of 10.9% and 11.8% on the 2x and 4x memory configurations due to increased contention with I-prefetches for bandwidth in the cache hierarchy.

The cause of the remaining I-cache misses are analyzed in Figure 4.11. For FDIP, the bottom portion of each bar represents complete I-cache misses, and the top portion represents partially prefetched misses. For prescient instruction prefetch, I-cache misses are classified in more detail by examining the history of events leading up to them. If no target was encountered recently enough that the corresponding postfix region would have included the instruction that missed, the classification is *no target*. If a target was found, but no preceding spawn could have triggered a helper thread to start at this target, the classification is *no spawn*. For a given implementation technique (e.g., FSMR), “no-spawn” and “no-target” I-cache misses can only be reduced by selecting spawn-target pairs that yield higher coverage. On average about 43% and 25% of remaining I-cache misses fall in these categories for VS, and FSMR respectively.

The FSMR mechanism has significantly fewer “no target” and “no spawn” misses than VS because the spawn-target selection for FSMR was performed with the expectation that FSMR helper threads can make faster progress through the postfix region than VS because fewer branches need to be precomputed. The shorter spawn-target distance provides more flexibility for spawn-target selection and hence higher coverage. Thus, faster helper threads may lead to better performance because they enable more flexible spawn-target selection.

If both a spawn and target are found, but an idle thread context did not exist at the time the spawn-point was committed by the main thread, the classification is *no context* (4% and 8% of I-cache misses on average for VS and FSMR, respectively). If a helper thread was successfully spawned there are three remaining possibilities: First, that the helper thread ran *too slow* so the main thread caught it before it could prefetch the instruction (5% and 3%); second, that the helper thread ran so far ahead that the instruction was brought into the cache but then *evicted* (7% and 17%). The number of "evicted" misses increases for FSMR relative to VS on 177.mesa. So while faster helper threads may help spawn-target selection, it may be helpful to eliminate such I-cache misses by regulating the subsequent issuing of counted prefetches. Finally, the access may already be *pending* (37% and 43%)[7].

FSMR improves performance more than FDIP for all benchmarks on the 2x configuration but only for gcc, mesa, crafty, and vortex on the 4x memory configuration. Comparing this data with that in Figure 3.2 we see that the benchmarks on which FDIP does best were the same three which obtained the largest prefetch distances. FSMR improved performance more on those benchmarks that showed smaller prefetch distances with FDIP in Figure 3.2.

4.4 Summary

This chapter presents two realistic implementation techniques for prescient instruction prefetch and introduces several associated hardware support mechanisms. The *Direct Pre-execution* implementation technique employs the *YAT-bit* mechanism to filter out 24% to 92% of postfix instructions so they consume no execution resources, and achieves a harmonic mean speedup of 11.1%. The *Finite State Machine Recall* technique achieves greater speedup of 22% by exploiting biased control flow and leveraging special prefetch instructions. The Finite State Machine Recall technique achieves even better speedup than a very aggressive hardware-only research instruction prefetch mechanism, Fetch Directed Instruction Prefetch, which yields an average speedup of 16.2%. This is largely a result of the fact that the prescient instruction prefetch technique can achieve larger prefetch slack than branch predictor guided techniques. Both direct pre-execution and finite state machine recall make use of the safe-store mechanism to support helper threads with dynamic store-to-load memory dependencies. When employing a two-bit tag, fewer than 2% of helper thread store-to-load dependencies are inaccurate with the safe-store mechanism, yet program correctness of the application main thread is maintained.

In the next chapter the modeling and optimization framework introduced in Chapter 2 is applied to the optimization of data prefetch helper threads.

[7]Slicing was limited to dynamic spawn-target pairs under 4000 instructions apart leaving a small number of I-cache misses due to *no slice*.

Threading	SMT processor with 4 thread contexts
Pipelining	In-order: 8-stage (2x), or 14-stage (4x) pipeline
Fetch	1 line per thread, 2 bundles from 1 thread, or 1 bundle from 2 threads priority main thread, helper threads ICOUNT [TEE^{+}96]
Instruction Prefetch	maximum 60 outstanding I-cache requests of any type (a) next line prefetch on demand miss (b) I-stream prefetch, max. 4 lines ahead of fetch (c) counted i-prefetch engine (CPE) - allocates into L1 I-cache 64-entry FIFO per thread, max. 2 issued per cycle (d) fetch-directed instruction prefetch [RCA99]: enqueue cache probe filter (enqueue-CPF) dedicated port for tag lookup 16- entry fetch target queue (FTQ), FIFO prefetch instruction queue (PIQ)
Branch Predictor	2k-entry GSHARE, 64-entry RAS with mispec. repair [SAMC98] 256-entry 4-way associative fetch target buffer (FTB)

Issue		priority main thread, helper threads round-robin
Function Units		4 int., 2 FP, 3 br., 2 mem. units
Register Files (per thread)		128 general purpose, 128 FP, 64 predicate, 8 branch, 128 control regs.
Cache	2x	L1 (separate I&D): 16 KB, 4-way (each) L2 256KB, 4-way 14-cycle L3 3MB, 12-way 30-cycle 64-byte lines, D-cache uses a write-through, no write allocate policy
	4x	latencies changed to: L2 28-cycle L3 60-cycle
Memory		2x: 230-cyc. TLB miss 30 cyc. 4x: 450-cyc. TLB miss 60 cyc.

Table 4.1: Processor Configurations

		benchmark	145.fpppp	176.gcc	177.mesa	186.crafty	191.fma3d	252.eon	255.vortex
row	description	abbrev.	fpppp	gcc	mesa	crafty	fma3d	eon	vortex
1	Static spawn-target pairs		62	3545	34	166	34	152	1348
2	Dynamic spawn-target pairs		18483	16105	10515	32092	16973	20786	22035
3	Avg. spawn-target distance (insts.)		638	645	1161	553	835	690	1028
4	Avg. PLP postfix region size (insts.) (2x/4x)		150 / 146	85 / 92	239 / 181	98 / 104	103 / 64	136 / 130	124 / 122
	Direct pre-execution (4x)								
5	Avg. # live-ins (register / memory)		0.88 / 1.07	4.5 / 13.7	8.9 / 14	8.0 / 12.9	3.4 / 5.3	3.9 / 7.4	5.1 / 12.1
6	Avg. # infix slice instructions		4.9	9.1	26	35	20	27	26.1
7	Avg. # infix store-to-load pairs		0.158	0.94	1.19	1.52	0.74	2.2	1.67
8	Avg. # postfix branches (static / dynamic)		2.6 / 1.53	25 / 29	22 / 18.8	31 / 13	1.09 / 2.4	17.0 / 8.4	26 / 47
9	Fraction of YAT-ed postfix inst.		8%	68%	54%	74%	24%	55%	76%
10	Avg. # YAT-ed loads / stores		3.1 / 0.61	16.1 / 2.1	19 / 2.2	15.0 / 3.0	7.1 / 0.73	11.2 / 2.4	18.1 / 4.3
11	Avg. # threads with a safe-store	safe-bit	14 / 9	19.7 / 113	100 / 88	37 / 96	0 / 0	13.4 / 11.7	78 / 173
12	eviction / alias (per 1000)	+ 2-bit tag	0.18 / 0.31	2.9 / 2.1	0.54 / 0.33	18.3 / 12.2	0 / 0	0 / 0.09	59 / 10
	FSMR (4x)								
13	Avg. # total t-slice instructions		7.4	94	25	140	38	79	14
14	Avg. # sliced postfix branches (static)		0.72	13.3	12.1	30	0.35	9.9	1.20
15	Avg. # precomputed postfix branches (dyn.)		0.50	6.8	1.91	6.4	0.39	2.1	0.53

Table 4.2: Helper Thread Statistics

Chapter 5

Optimization of Data Prefetch Helper Threads

This chapter investigates helper threads that improve performance by prefetching data on behalf of an application's main thread. An extension of the modeling framework presented in Chapter 2 is developed and applied to the task of optimizing data prefetch helper threads that lack branch instructions and generate prefetches for one dynamic instance of a delinquent load instruction per spawned helper thread. This form of helper thread, sometimes called a *simple p-thread*, has been studied previously by Roth and Sohi who proposed a framework for optimizing the performance impact of p-threads by judiciously selecting the length of the p-thread [RS02]. In their framework, the input to the optimization process is a program trace containing information about the dynamic dependencies between instructions. While they suggest it should be possible to achieve similar optimization results using less detailed program information, they did not propose any specific techniques for achieving this goal in [RS02]. The main contribution made in this chapter of the dissertation is the proposal and evaluation of such a technique–one that achieves comparable results to Roth and Sohi's optimization approach while requiring less detailed information about dynamic program behavior. The information used by the proposed technique is similar to that provided within current feedback directed optimizing compilers. A secondary contribution of this chapter is an assessement of the major sources of modelling error relevant to predicting the performance impact of data prefetch helper threads.

From a compiler developer's perspective, the benefit of the technique proposed here is that it fits well within traditional optimizing compiler implementation frameworks. Profile information is represented by annotating a graph data structure representing a static view of a program. Such graph data structures are widely used within typical compilers. On the other hand, execution trace data represents each dynamic instance of an instruction individually. While it is often used in computer architecture research, to the best of our knowledge such trace data is not processed by any commercially developed or open source compilers.

The technique proposed here may also generate helper thread enabled code faster than using trace data, because less profile data needs to be collected and stored for later retrieval, and correspondingly less profile data needs to be analyzed after profiling has been completed. The actual code generation speed tradeoff was not quantified in this dissertation because the implementation used for this study was designed to support the flexibility required for effective research rather than code generation speed typically required in a production compiler. Given profile information, the time taken to extract a slices was on the order of tens of seconds (thus not unreasonable) using our approach, however using more aggressive implementation techniques we feel it should be easy to reduce the overhead of this optimization. We feel the compiler implementation advantages would likely make the technique we propose here more favorable than a trace based approach even if it turned out to be somewhat slower.

The rest of this chapter is organized as follows. In Section 5.1, Roth and Sohi's optimization technique is summarized. In Section 5.2, a straightforward application of the techniques in Chapter 2 is used to transform their approach into an algorithm using path expressions instead of trace data. This approach is illustrated for a particular problem load in an important benchmark from the SPEC CPU 2000 integer benchmark suite and a problem with branch outcome correlation is identified. Section 5.3 proposes an extension of the algorithm described in Section 5.2 using higher order control flow profile information to obtain more accurate predictions of program behavior resulting in better helper thread optimization. Section 5.4 presents our experimental evaluation of the proposed techniques and Section 5.6 concludes.

5.1 Optimization Using Trace Data

In Roth and Sohi's framework, a program's execution trace is converted into a data structure, called the *slice tree* that implicitly represents all the helper threads for a specific delinquent load instruction (see Figure 5.1). The slice tree is a directed graph in which vertices represent instructions and a directed edge joins an instruction to a successor instruction in a single data prefetch helper thread's instruction sequence. Starting at any vertex in the slice tree and following outflowing edges at each vertex eventually leads to the root vertex of the slice tree. The root vertex represents the problem load instruction for which we wish to prefetch data. Each vertex in the slice tree implicitly represents a complete simple prefetch thread (p-thread) composed of the instructions encountered during the traversal from this vertex to the root. The instruction corresponding to the start vertex is called the *lead* instruction and the instruction in the main thread corresponding to the lead instruction is used to trigger a dynamic instance of the helper thread.

5.1.1 Motivation

During program execution, a helper thread instance implied by the slice tree may exist concurrently with many other implied helper thread instances either from the same slice tree or from another slice tree. As the structure of the slice tree is derived from the application program and its inputs, *ignoring the details of the processor microarchitecture*, running every helper thread represented in a slice tree may require executing more dynamic helper thread instances at a given moment in time than the number hardware can support. When all helper threads implied by a slice tree cannot be run concurrently, the use of hardware resources can be optimized by judiciously selecting the set of helper threads actually used from among all helper threads implied by the slice tree. Note that this situation is similar to the need to apply graph coloring to allocate registers so they are used efficiently when the number of variables in a program exceeds the number of registers in the processor architecture.

5.1.2 Aggregate Advantage

To optimize the selection of (static) helper threads Roth and Sohi introduce the notion of *aggregate advantage*, which is a numerical score given to each instruction in the slice tree to indicate how much execution-time benefit is obtained from using a helper thread that is launched at each occurrence of the lead instruction in the main thread. Aggregate advantage approximates the number of execution cycles saved by using the helper thread. Its predictions may differ from the number of cycles saved during p-thread assisted execution, an effect quantified using the *absolute accuracy* of the aggregate advantage compared to the actual speedup using p-thread assisted execution. In practice the importance of an optimization metric such as aggregate advantage is not its absolute accuracy, but rather how accurately it predicts the *relative* merit of different optimization choices (i.e., different helper threads). Thus, as long as aggregate advantage captures the important underlying trends that influence the difference in the performance impact of helper threads it remains useful for optimization even if large absolute errors exist. In some cases it may not be possible to effectively prefetch a particular problem load with any helper thread implied in the slice tree (for instance, if the backward slice contains all instructions in the program). In such cases the optimization metric should also accurately indicate whether the selected helper thread will *likely* improve performance. Thus, if the aggregate advantage indicates the best helper thread would likely *not* improve performance, the aggregate advantage framework does not select that helper thread.

Figure 5.1 illustrates a slice tree for a delinquent load in the benchmark 181.mcf from the SPEC 2000 integer benchmark suite. The code is from the function `refresh_potential()` and several lines directly contributing to the address computation are highlighted on the left. The slice tree is shown on the lower right side of the figure in the dashed box. Each node in the tree represents an instruction in the program binary, and is labeled with a letter corresponding to the line from the source code. The values next to each circle represent the aggregate advantage measured in millions of cycles. The best slice to use as a p-thread—determined experimentally by measuring the actual speedup obtained—is emphasized in bold in the slice tree starting from the node with a double edge labeled B.

The aggregate advantage is computed as follows [RS02]:

$$\mathrm{ADV}_{agg}(p) = \mathrm{LT}_{agg}(p) - \mathrm{OH}_{agg}(p) \tag{5.1}$$

where, $\mathrm{ADV}_{agg}(p)$ is the aggregate advantage of the helper thread p, $\mathrm{LT}_{agg}(p)$ is the aggregate latency tolerance of the helper thread, and $\mathrm{OH}_{agg}(p)$ is the aggregate overhead. The *aggregate latency tolerance* is the number of cycles of stalled execution that are removed by the helper thread. The *aggregate overhead* is the number of additional cycles incurred if the data prefetched by the helper thread were "ignored" by the main thread so that the main thread had to access these values from wherever they were in the memory hierarchy before the helper thread attempted to prefetch them. As latency tolerance is only provided by those helper thread instances that prefetch data not already in the cache that the main thread subsequently uses, the aggregate

```
    long refresh_potential( network_t *net )
    {
        node_t *stop = net->stop_nodes;
        node_t *node, *tmp;
        node_t *root = net->nodes;
        long checksum = 0;

        for( node = root, stop = net->stop_nodes; node < (node_t*)stop; node++ )
            node->mark = 0;

        root->potential = (cost_t) -MAX_ART_COST;
Ⓧ.......tmp = node = root->child;
        while( node != root ) {
            while( node ) {
Ⓐ...............if( node->orientation == UP )
                    node->potential = node->basic_arc->cost + node->pred->potential;
                else /* == DOWN */ {
                    node->potential = node->pred->potential - node->basic_arc->cost;
                    checksum++;
                }

Ⓖ...............tmp = node;
Ⓑ...............node = node->child;
            }

Ⓕ...........node = tmp;

            while( node->pred ) {
Ⓓ...............tmp = node->sibling;
                if( tmp ) {
Ⓒ...................node = tmp;
                    break;
                }
                else
Ⓔ...................node = node->pred;
            }
        }

        return checksum;
    }
```

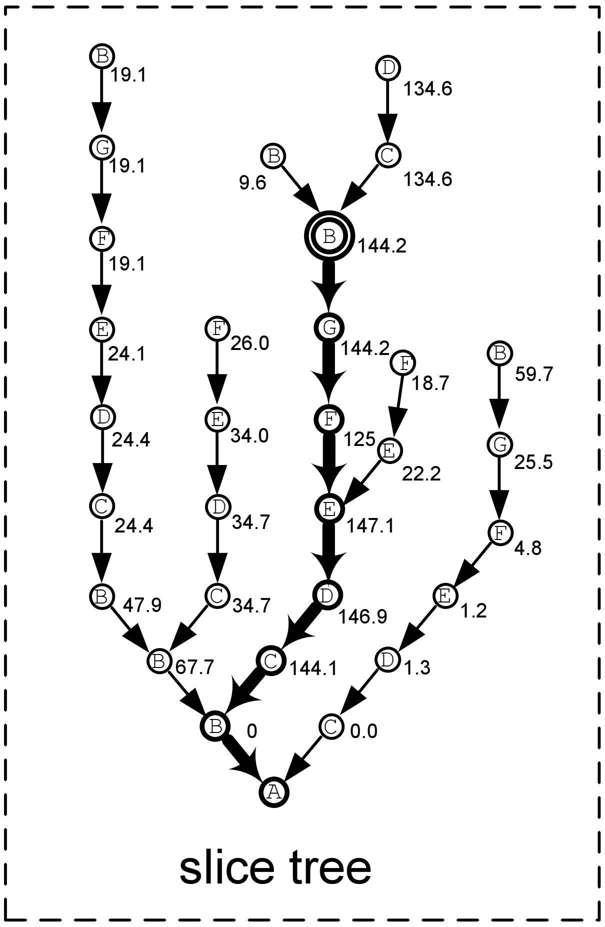


Figure 5.1: Slice tree example for a delinquent load in 181.mcf. Values beside nodes in slice tree are aggregate advantage in millions of cycles. The best helper thread, as determined by the actual speedup obtained, begins at the node B with the double edge and follows the highlighted path.

latency tolerance can be expressed as [RS02]:

$$\mathrm{LT}_{agg}(p) = \mathrm{DC}_{pt-cm}(p) \cdot \mathrm{LT}(p) \tag{5.2}$$

$$\mathrm{LT}(p) = min(\mathrm{SCDH}_{mt}(p) - \mathrm{SCDH}_{pt}(p), \mathrm{L}_{cm}) \tag{5.3}$$

where $\mathrm{DC}_{pt-cm}(p)$ is the dynamic count of helper threads that pre-execute *actual misses*, and $\mathrm{LT}(p)$ is the average per-helper-thread-instance latency tolerance of helper thread p. The latter can be expressed as the minimum of the average difference between the schedule constrained data flow heights of the execution sequence of the main thread and the helper thread ($\mathrm{SCDH}_{mt}(p)$ and $\mathrm{SCDH}_{pt}(p)$ respectively) and the latency of a cache miss (L_{cm}). For the example shown in Figure 5.1, the values measured for the p-thread starting at the node with the double edge are:

$$\begin{aligned}
\mathrm{DC}_{pt-cm}(p) &= 480753 \\
\mathrm{SCDH}_{mt}(p) &= 670 \\
\mathrm{SCDH}_{pt}(p) &= 111 \\
\mathrm{L}_{cm}(p) &= 300
\end{aligned}$$

In our implementation the value of $\mathrm{SCDH}_{mt}(p)$ was determined on a per slice instance basis by considering the latency impact of data cache misses and employing list scheduling on the main thread instructions encountered from the lead instruction to the delinquent load assuming the modeled issue bandwidth and ignoring instruction fetch sequencing constraints (the implementation used in prior studies is a simpler though potentially less accurate technique for evaluating this quantity that only considers the number of dynamic instructions between the lead and delinquent load encountered by the main thread and the main thread's fetch sequencing bandwidth [RS02]). Similarly, $\mathrm{SCDH}_{pt}(p)$ is computed using list scheduling and data cache latencies.

Substituting these value back into the equations for aggregate advantage yields:

$$\begin{aligned}
\mathrm{ADV}_{agg}(p) &= 480753 * min(670 - 111, 300) \\
&= 144.2 \times 10^6 \ \ \text{[cycles]}
\end{aligned}$$

The actual execution time saved when this helper thread was used was 96.3M cycles, which

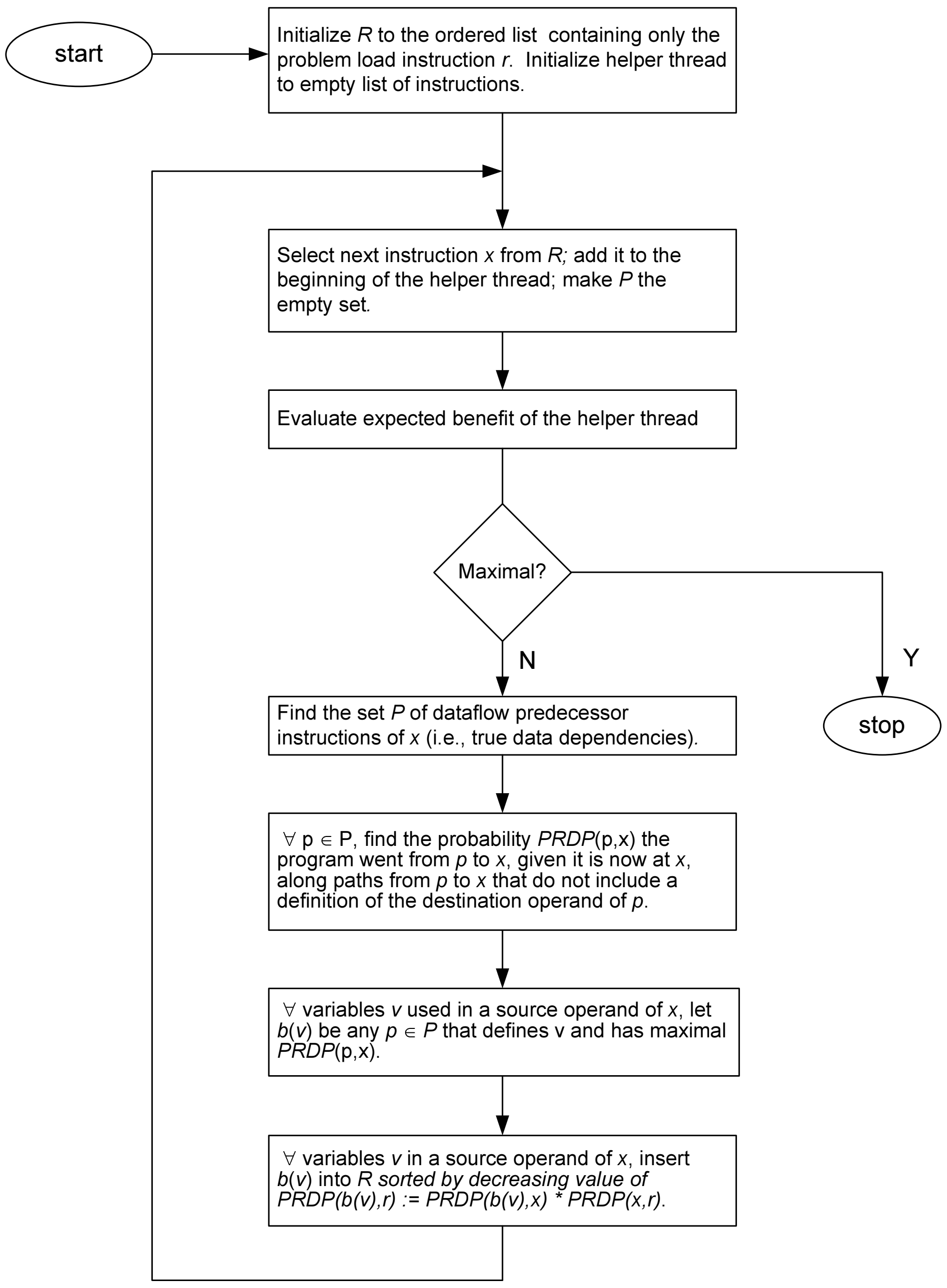

start
Initialize R to the ordered list containing only the problem load instruction r. Initialize helper thread to empty list of instructions.
Select next instruction x from R; add it to the beginning of the helper thread; make P the empty set.
Evaluate expected benefit of the helper thread
Maximal?
N
Y
stop
Find the set P of dataflow predecessor instructions of x (i.e., true data dependencies).
∀ p ∈ P, find the probability PRDP(p,x) the program went from p to x, given it is now at x, along paths from p to x that do not include a definition of the destination operand of p.
∀ variables v used in a source operand of x, let b(v) be any p ∈ P that defines v and has maximal PRDP(p,x).
∀ variables v in a source operand of x, insert b(v) into R sorted by decreasing value of PRDP(b(v),r) := PRDP(b(v),x) * PRDP(x,r).

Figure 5.2: Path expression p-thread optimization algorithm (PRDP definition on page 87)

translates into a speedup of 12.6%. We note that the node with the largest aggregate advantage was E, but the p-thread starting at this node had an actual speedup of only 1.4%. The causes of discrepancies between actual and predicted speedups are studied in Section 5.5. In the following section we show how to compute a quantity similar to the aggregate advantage using the path expression framework introduced in Chapter 2.

5.2 Optimization using Static Slicing and Path Expressions

The values used in the aggregate advantage optimization framework [RS02] are statistical quantities. In this section we introduce an algorithm for optimizing the spawn location of data prefetch helper threads without using trace data, by approximating these quantities using a path expression based statistical analysis of control flow through the program. The algorithm relies upon static program slicing to obtain information about an instruction's source operand data dependencies, and uses the path expression framework of chapter 2 to determine the expected benefit (similar to aggregate advantage) of using a given helper thread to prefetch data.

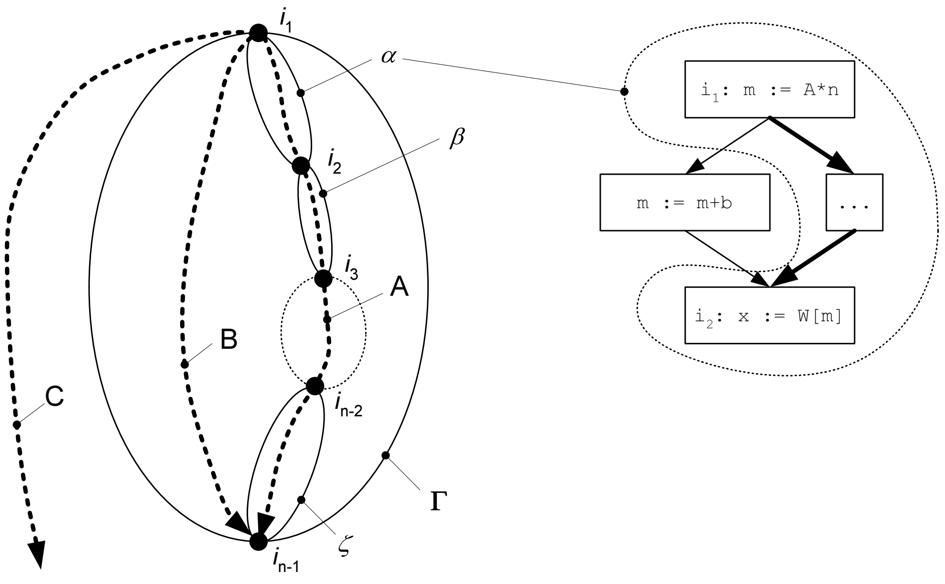
The algorithm we describe proceeds in an iterative fashion starting from the delinquent load and incrementally building up a p-thread for that load. The algorithm is outlined in Figure 5.2. An instruction is included in the p-thread if it is a true data flow predecessor of an instruction already in the p-thread and it is the "most likely" instruction to generate a live-in value to the existing p-thread. To quantify "most likely" we perform a form of *data flow frequency analysis* [Ram96] by leveraging the path expression based method of calculating the reaching probability introduced in Chapter 2.

5.2.1 Expected Benefit

The high-level statistical quantity used in our optimization framework to determine which slice is the best is the *expected benefit* of using a helper thread. This is computed by assuming, for simplicity, that OH_{agg} is negligible, and by expressing Equation 5.1 using the following formula. Given a sequence of instructions $i_1, i_2, ...i_n$ making up a p-thread p, the expected benefit of the p-thread is computed as follows:

$$\mathrm{E}[\mathrm{benefit}(p)] = \big(\prod_{k=1}^{n-1} RDP(k|p)\big) \cdot min\big((\mathrm{E}[\mathrm{lat}_{mt}] - \mathrm{E}[\mathrm{lat}_{ht}]), \mathrm{lat}_{mem}\big) \cdot \mathrm{f}_{sp} \cdot \mathrm{P}[\mathrm{miss}] \qquad (5.4)$$

Before explaining how this equation is related to Equation 5.1 we describe its components. The quantity $RDP(k|p)$ is the *reaching definition probability* from instruction i_k to i_{k+1} for the helper thread p. This is the probability that program execution goes from instruction i_k to instruction i_{k+1} without defining any variables that are live between instruction i_k and i_{k+1} in the helper thread. This set of paths forms the implied control flow of the p-thread. We define the *implied control flow* of a single-prefetch helper thread as the set of paths from the spawn point to the target delinquent load that pass through each instruction in the p-thread in the same order as



The implied control flow from instruction i_1 to i_{n-1} is the subset of all paths Γ from i_1 to i_{n-1} that pass through the instructions i_k in the slice defining the p-thread in the same order as the p-thread executes them, such that no live variable between two instructions in the p-thread is redefined along the paths between these instructions followed by the main thread. This set of paths is composed of the concatenation of the path expression α summarizing admissable control flow between instruction i_1 and i_2 (i.e., control flow paths avoiding definitions of variables live at that point in the helper thread), with the path expression β summarizing admissable control flow between instruction i_2 and i_3, etc... up to the path expression ζ summarizinng admissable control flow between instruction i_{n-2} and i_{n-1}. After spawning, the main thread either follows the implied control flow of the p-thread (e.g., path 'A'), a path that likely generates a different address (e.g., path 'B') or a path that avoids the target load altogether (e.g., path 'C').

Figure 5.3: Implied Control Flow

the p-thread, such that no live variable between two instructions in the p-thread is redefined along the paths between these instructions followed by the main thread (see Figure 5.3). If the main thread follows the implied control flow of the p-thread after spawning that p-thread, it is guaranteed to compute the same address for the delinquent load as the helper thread.

The quantity E[latency$_{mt}$] is the expected latency of the main thread from the spawn to the delinquent load and is evaluated using the path expression method for computing expected path length, modified to weight basic blocks by the number of cycles they take to execute on average (this value can be collected during profiling using performance monitoring hardware such as that in current generation microprocessors). We are only interested in the latency of the main thread when its control is compatible with the implied control flow of the helper thread. Hence, the value of E[latency$_{mt}$] we are interested in can be decomposed as follows:

$$\mathrm{E}[\mathrm{lat}_{mt}] = \sum_{k=1}^{n-1} latency(k|p) \tag{5.5}$$

where the quantity $latency(k|p)$ is the expected number of cycles from instruction i_k to i_{k+1} avoiding paths through other instructions in helper thread p, and also avoiding paths through instructions that kill i_k's definition.

To evaluate E[lat$_{ht}$] the execution latencies of each instruction are simply added. This ignores the potential for ILP in the helper thread, but requires less analysis. The assumption of little ILP for the helper threads turns out to be a valid one for the simple helper threads we examined in this study as the dependency graphs are often a single chain of dependent instructions.

The quantity f$_{sp}$ is the frequency the spawn instruction is executed (determined via basic block profiling) and finally, the quantity P[miss] is the relative frequency that the delinquent load misses in the cache.

The connection between Equation 5.4 and Equation 5.1 is as follows. DC$_{pt-cm}(p)$ is the number of instances of the slice p that lead to an occurrence of the delinquent load that suffers a cache miss. This is approximately equal to the total number of slices leading to the delinquent load multiplied by the cache miss rate of the delinquent load:

$$\mathrm{DC}_{pt-cm}(p) \approx \mathrm{DC}_{pt} \cdot \mathrm{P}[\mathrm{miss}] \tag{5.6}$$

The approximation in Equation 5.6 assumes that the probability of incurring a data cache miss is independent of the control flow path up to the target delinquent load instruction, and that all helper thread instances that prefetch data for an instance of the target load cache miss must follow the intermediate control flow assumed when slicing was performed. As described in Section 5.5, in some cases it was observed that significant prefetching of later instances of the target load may occur, even when the control flow of the main thread after the spawn deviates from the implied control flow of the helper thread. Furthermore, as noted in Section 5.5, the assumption that the cache miss rate is independent of the execution path leading up to the load also introduces some inaccuracy.

The quantity DC$_{pt}$ on the right hand side in Equation 5.6 is equal to the frequency the spawn instruction is executed by the main thread (f$_{sp}$), multiplied by the relative frequency that execution subsequently follows the implied control flow of the p-thread. The relative frequency that execution follows the implied path is the ratio of the frequency execution follows the implied

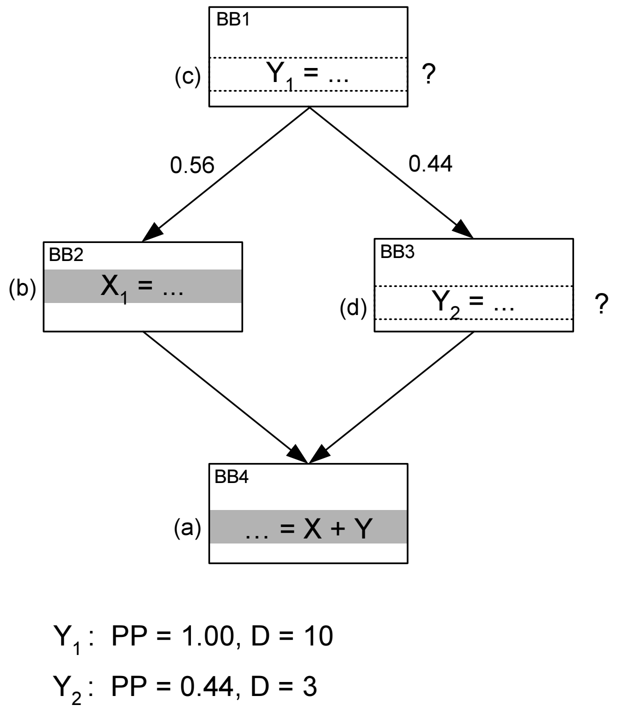


Figure 5.4: Example illustrating need for predecessor pruning during p-thread slicing.

path ($\mathrm{f}_{implied\ cf}$) over the frequency with which the spawn point is reached (f_{sp}). The ratio of $\mathrm{f}_{implied\ cf}$ over f_{sp} can be estimated using the path expression framework so that DC_{pt} can be expressed as:

$$\mathrm{DC}_{pt} = \mathrm{f}_{sp} \cdot \frac{\mathrm{f}_{implied\ cf}}{\mathrm{f}_{sp}} \tag{5.7}$$

$$\mathrm{DC}_{pt} = \mathrm{f}_{sp} \cdot \big(\prod_{k=1}^{n-1} RDP(k|p)\big) \tag{5.8}$$

To estimate the quantity LT_{agg} we use the value $\mathrm{E}[\mathrm{lat}_{mt}] - \mathrm{E}[\mathrm{lat}_{pt}]$ described above.

5.2.2 Predecessor Selection and Pruning

The form of static program slicing used in this study takes as input an instruction and returns a set of dataflow predecessors. (Other forms of static slicing take a set of variables and a location in the program such as an instruction, or an instruction and the calling context.) Rather than build an entire slice tree, the algorithm we propose builds a single slice equivalent to one of the paths in the slice tree. The goal of the algorithm is to find the slice in the slice tree with maximum expected benefit.

As shown in Figure 5.1, the aggregate advantage may have local optima with increasing distance from the delinquent load. For example, the aggregate advantage drops when going from node E to node F in the selected p-thread, whereas it increases going from node F to node G. Thus it is necessary to search past the first point that aggregate advantage decreases to ensure a better spawn location does not exist.

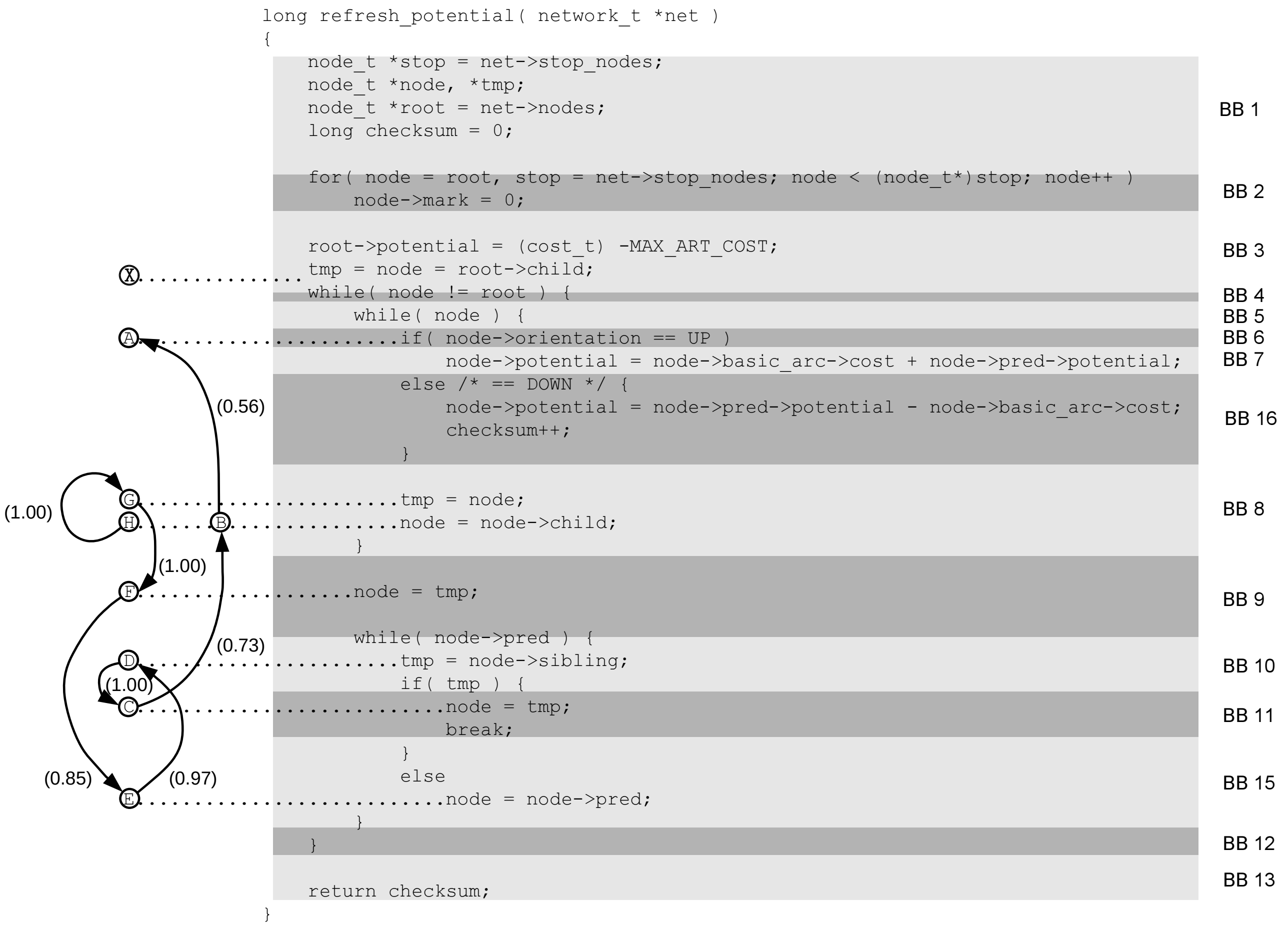
There are two interrelated challenges to implementing the desired algorithm: The first, is ensuring the p-thread being created incrementally is actually in the slice tree, and the second is selecting the best instruction to add to the p-thread among all those meeting the first criterion. By assumption, our algorithm does not have access to the slice tree, so when selecting a new instruction to add to the helper thread from the set of dataflow predecessors found via static slicing it will not immediately be clear whether or not adding that predecessor to the p-thread will result in a p-thread that exists in the slice tree. For example, in Figure 5.4, two instructions define register Y, however assuming that X_1 was already selected to be in the slice, adding Y_2 would lead to a p-thread that is not in the slice tree. The underlying reason is that Y_2 does not lie on a path leading to X_1 without intersecting another instruction already selected to be in the p-thread (in this case instruction "a").

To ensure the p-thread being created is in the slice tree, we filter the set of dataflow predecessors as new instructions are added to the slice to prune away predecessors inconsistent with the implied control flow of the resulting helper thread.

From among the remaining data flow predecessors we use the following heuristic to select the next best instruction to include in the slice: For each of the data flow predecessors, we compute the *posteriori reaching definition probability* (PRDP) excluding paths including other predecessors under consideration. The PRDP of a dataflow predecessor to a particular instruction is the reaching probability in the reversed graph excluding edges leaving basic blocks that contain a kill site for the definition generated by the predecessor. For each live-in variable to the p-thread, we select the predecessor with the largest PRDP.

5.2.3 Example

Continuing the example from the previous section, Figure 5.5 shows the same source code highlighting basic blocks in the optimized SimpleScalar [BA97] binary. On the left of Figure 5.5, the slice used to construct a simple p-thread is illustrated. Here the circles indicate the corresponding



```
long refresh_potential( network_t *net )
{
    node_t *stop = net->stop_nodes;
    node_t *node, *tmp;
    node_t *root = net->nodes;
    long checksum = 0;

    for( node = root, stop = net->stop_nodes; node < (node_t*)stop; node++ )
        node->mark = 0;

    root->potential = (cost_t) -MAX_ART_COST;
    tmp = node = root->child;
    while( node != root ) {
        while( node ) {
            if( node->orientation == UP )
                node->potential = node->basic_arc->cost + node->pred->potential;
            else /* == DOWN */ {
                node->potential = node->pred->potential - node->basic_arc->cost;
                checksum++;
            }

            tmp = node;
            node = node->child;
        }

        node = tmp;

        while( node->pred ) {
            tmp = node->sibling;
            if( tmp ) {
                node = tmp;
                break;
            }
            else
                node = node->pred;
        }
    }

    return checksum;
}
```

Figure 5.5: Code excerpt from Figure 5.1 with basic blocks in the `gcc-2.6.3` optimized, SimpleScalar program binary.

lines of code are replicated in the p-thread, and the arrows indicate data dependencies between these lines and the implied sequencing of p-thread instructions. Using the aggregate advantage methodology, the delinquent load on the line marked "A" would be prefetched by creating a separate thread that launches each time the main thread reaches the line marked "H".

The edges in the graph are labeled with the relative frequency with which the corresponding dependency occurs during program execution *given that the successor is executed.* After spawning helper thread execution starts with the instruction corresponding to the line of code marked "H" in Figure 5.5, then, capturing one iteration of the first inner loop, proceeds to "G", exits the first inner loop, and continues as traced out by the labels in descending alphabetical order until reaching the line marked "A". The delinquent load targeted by this helper thread corresponds to the dereference "`node->orientation`" on line "A" in the source code.

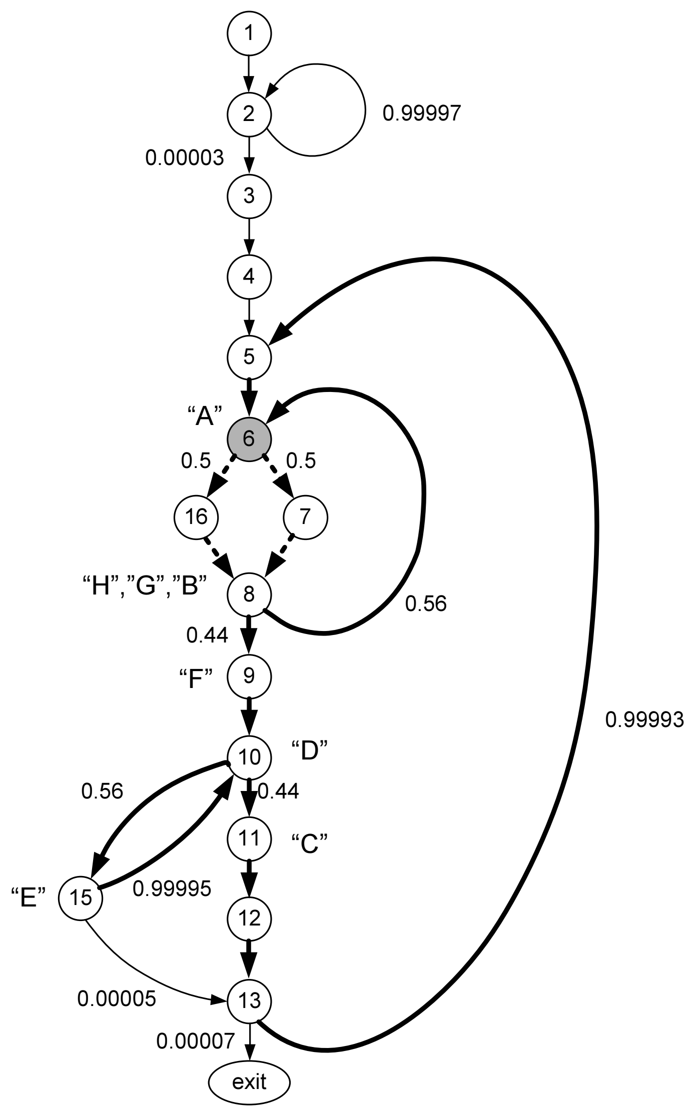


Figure 5.6: Control flow graph for code in Figures 5.1 and 5.5 showing branch transition probabilities.

In Figure 5.6 the control flow graph for the function `refresh_potential()` is plotted and the implied control flow of the helper thread highlighted in Figure 5.5 is traced out with bold arrows. This is the optimization result we wish to obtain. For this example we note that the implied control flow of the desired p-thread starts in basic block 8, follows the backward branch to basic block 6, goes along either path back to basic block 8, then falls out of the loop to basic block 9, continues to basic block 10, then to basic block 15, and back to block 10, then goes around the outer loop via blocks 11, 12, 13 and 5 to finally reach the delinquent load at 6.

The proposed p-thread optimization algorithm starts by including the delinquent load in

Sp. Inst.	(a) $\prod$ RDP	(b) f_{sp}	(c) P[miss]	(d) E[slack]	(e) E[benefit] (cycles)	(f) DC$_{pt-cm}$	(g) ΔSCDH	(h) ADV$_{agg}$ (cycles)	(i) Actual (cycles)
B	0.56	2.55M	0.43	0	0	842682	0	0	0.5M
C	0.56	1.12M	0.43	254	68M	567304	254	144.1M	5.4M
D	0.313	2.55M	0.43	260	**159M**	567304	260	146.9M	17.6M
E	0.138	1.43M	0.43	260	89M	565767	260	**147.1M**	12.0M
F	0.077	1.12M	0.43	260	39M	480753	260	125.0M	10.4M
G	0.077	2.55M	0.43	363	45M	480753	363	144.2M	87.5M
H	0.077	2.55M	0.43	559	25M	480753	559	144.2M	**96.3M**

Table 5.1: Comparison of static slicing with statistical control flow analysis using our path expression optimization metric in column (e) versus Roth and Sohi's aggregate advantage metric in column (h) and actual speedup in column (i).

the p-thread and performing static slicing to find that instructions B, C, E, F and X are dataflow predecessors of A. The posteriori reaching definition probabilities for these instructions are then evaluated using control flow profile information resulting in values of 0.560, 0.440, 0.000, 0.000, and 0.000 respectively (calculations not shown—these values are derived similar to the posteriori probability but excluding paths through instructions that define "node" in Figure 5.5). Therefore, the specific definition of the program variable "node" (in Figure 5.5) that is most likely to reach A given the program is executing instruction A is the definition of "node" at instruction B. Continuing in this way, the algorithm will follow the same slice selected by the aggregate advantage optimization framework.

To determine when to stop adding instructions to the p-thread, we examine the expected benefit of the p-thread as each additional instruction is added. Table 5.1 shows the value of the reaching definition probability (RDP) for the implied control flow of the entire p-thread as nodes are added (column a), the spawn frequency (column b), the cache miss frequency (column c), the expected slack (column d), and the expected benefit (column e). Also included are values from the aggregate advantage calculation, namely DC$_{pt-cm}$ (column f), the difference ΔSCDH = SCDH$_{mt-pt}$ in the schedule constrained data flow heights of the main thread and the helper thread (column g), and the aggregate advantage computed using the aggregate advantage technique (column h). Finally, the cycles actually saved if the p-thread is used are in column (i). To simplify the analysis, the estimate prefetch slack in column (d) was simply measured using SCDH$_{mt-pt}$ (accurately accounting for the impact on execution time of concurrent execution of instructions from different basic blocks along a given control flow path is challenging though not impossible [OH00]).

We see that the path expression model predicts the peak benefit to occur for the p-thread starting at node D, whereas the peak aggregate advantage value is for node E. Furthermore, the best speedup is actually obtained for the p-thread starting at node H. Examining columns (a) through (h) it can be seen that the discrepancy in predictions between the path expression technique and the aggregate advantage technique is largely due to the value RDP computed using the path expressions. For example, for the p-thread starting at node H, the low value of the predicted benefit is most heavily influenced by the low value of RDP. It turns out that the value of RDP significantly under estimates the actual value. For example, the probability of following the implied control flow from H to A is actually roughly 0.5 rather than 0.077.

As the main contributor to the suboptimal optimization decision made with the current version of our path expression technique is the computation of RDP, a more accurate approach to predicting this component seems desirable. Section 5.3 proposes an extension of the path expression technique that dramatically improves the accuracy of RDP prediction.

5.3 Improving Prediction Accuracy with Higher-Order Control Flow Statistics

In this section we show that by computing control flow statistics over a second order control flow graph, the accuracy in the prediction of the performance impact of a given p-thread using path expression based analysis can be improved significantly. Intuitively a more accurate statistical model would be expected to lead to better optimization decisions. The contrast between the results for the "first-order" technique presented in the last section and the "second-order" technique presented in this section also highlights the importance of the analysis of control flow on simple p-threads—an area that has not received significant attention in any prior work we are aware of.

In the last section we concluded that some way of capturing the impact of branch outcome correlation in our statistical analysis could improve the accuracy of the predicted expected benefit of a particular p-thread. To do this, a higher-order Markov chain can be employed in the analysis. We construct a higher-order Markov chain by considering a state to consist of the combination of both a basic block and the recent control flow history leading up to that basic block. A first

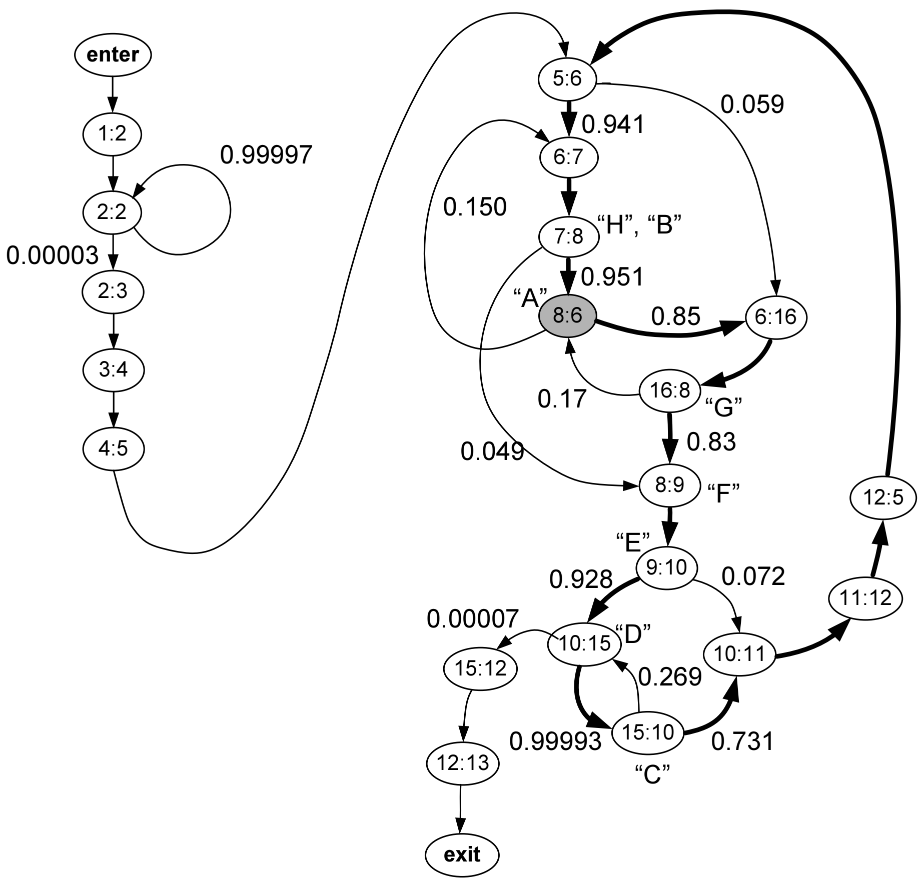


Figure 5.7: Second-order control flow graph for code in Figures 5.1 and 5.5.

order Markov chain thus consists of states encoding no branch outcome history leading up to the current basic block, while a second order Markov chain consists of states in which the last basic block and the current basic block are together encoded in each state. Even higher-order models could be constructed by including even more branch history in each state. We call the graph representing the nodes and state transitions of a second order Markov chain a *second order control flow graph.*

Figure 5.7 illustrates the second-order control flow graph for `refresh_potential()`. Each node in this graph represents a branch outcome in the function `refresh_potential()` and is labeled using the convention n:m, where n and m are basic block numbers in Figures 5.5 and 5.6. n represents the previous basic block, while m represents the basic block where the program is currently executing. The bold lines indicate the implied control flow of the p-thread selected by the aggregate advantage framework. Note that, even though the original source code's control flow is reducible, this graph is non-reducible. To apply the techniques of Chapter 2 to evaluate

Sp. Inst.	(a) $\prod$ RDP	(b) f_{sp}	(c) P[miss]	(d) E[slack]	(e) E[benefit] (cycles)	(f) DC$_{pt-cm}$	(g) ΔSCDH	(h) ADV$_{agg}$ (cycles)	(i) Actual (cycles)
B	0.951	2.55M	0.43	0	0	842682	0	0	0.5M
C	0.897	1.12M	0.43	254	72.5M	567304	254	144.1M	5.4M
D	0.897	2.55M	0.43	260	**169M**	567304	260	146.9M	17.6M
E	0.897	1.43M	0.43	260	94.3M	565767	260	**147.1M**	12.0M
F	0.833	1.12M	0.43	260	68.9M	480753	260	125.0M	10.4M
F	0.691	2.55M	0.43	363	150M	480753	363	144.2M	87.5M
H	0.651	2.55M	0.43	559	141M	480753	559	144.2M	**96.3M**

Table 5.2: Comparison of static slicing with second-order statistical control flow analysis using path expressions versus aggregate advantage.

statistical properties of program execution it is therefore necessary to use the version of Tarjan's path expression algorithm that generates path expressions correctly for non-reducible flow graphs. This version of the algorithm has slightly larger computational complexity compared to the simpler version that works correctly only for reducible flow graphs, however it enables us to reap the benefits of more accurate analysis during helper thread optimization.

Table 5.2 provides a comparison similar to that in Table 5.1 between the enhanced version of our optimization algorithm and the aggregate advantage framework. Examining the data in Table 5.2 and comparing it to the data in Table 5.1 we see that the value of RDP computed using the second-order control flow graph decreases much more slowly as the slice length increases, and that the estimated benefit is now closer to the aggregate advantage value for longer slices. For example, node H is predicted to have a benefit of 141M cycles using the second-order control-flow graph, versus 25M for the first-order control flow graph (in Table 5.1) and 144.2M for the aggregate advantage.

In Section 5.4 we extend the evaluation of the three approaches to p-thread optimization described in this chapter to a larger selection of benchmarks and again contrast them to the actual benefit obtained when employing the resulting p-threads.

5.4 Experimental Evaluation

In this section we evaluate the proposed data prefetch helper thread optimization algorithm. After describing our experimental methodology, we show that the proposed optimization algorithm compares favorably to an enhanced version of aggregate advantage methodology at predicting the speedup obtained when employing a given p-thread to prefetch for given problem load instruction. Then we delve deeper into the sources of modeling error that exist and measure sensitivities to various aspects of the modeling framework and the simulated microarchitecture.

5.4.1 Methodology

To evaluate the various approaches to simple p-thread optimization we selected five benchmarks from the SPEC 2000 integer benchmark suite [Sta] and two benchmarks from the Olden benchmark suite [Car96] that exhibited large performance degradation due to data cache misses. Simulation data presented in this chapter was collected using the benchmark reference inputs over a 100 million instruction segment after fast-forwarding to a representative location of program execution found using the SimPoint toolkit [SPHC02b]. Data cache, traditional edge profiling, and second-order control flow profile information was obtained by running the same input and program segment as used to evaluate performance. We extended the SimpleScalar microprocessor simulator to model an SMT processor with the microarchitecture parameters given in Table 5.3. Two mechanisms for creating helper thread enabled applications where developed that leveraged the path expression based modelling framework introduced earlier in this dissertation.

Table 5.3: Processor Configurations

Threading	SMT with 4 thread contexts	Func. Units	4 iALU, 2 LD/ST, 4 fpALU 1 iMUL/DIV, 1 fpMUL/DIV
Superscalar Pipeline	4-way issue/decode/commit 256 entry RUU, 64 entry LSQ	Reg. Files (per thread)	32 GPR, 32 FPR
Fetch	8 inst./thread (max 2 threads/cyc) priority main thread	Caches	L1I 32KB, 2-way, 32B blks, 1-cyc L1D 16KB, 4-way, 32B blks, 2-cyc L2 1 MB, 4-way, 64B blk, 14-cyc
Branch Prediction	max 2 taken branch/cyc/thread hybrid predictor with (a) 2048-entry bimodal (b) 256-entry global predictor (c) 1024-entry selector 64-entry return address stack	Memory	300-cycle

Initially, the infrastructure illustrated in Figure 5.8 was developed. Referring to this figure, the optimizer (a) implements the algorithm shown in Figure 5.2. Starting from a delinquent load, the optimizer requests the set of dataflow predecessors from the slicer (c). The slicer we used in these experiments is the CodeSurfer slicer available from GrammaTech Inc [Inc04]. We interface the optimizer to CodeSurfer by adding it as a shared library and using CodeSurfer's STK programming environment to provide high-level control and additional interface functionality. The optimizer also interfaces with an implementation of the path expression framework (b), described in Chapter 2 designed to process profile information collected from the SimpleScalar simulator [BA97]. The output of the optimization process is a *program dependence graph* (PDG) [FOW87] representation of the helper thread, containing a list of the lines of code in the p-thread and the live-in variables read by that p-thread. These are passed to a special compiler pass (d) developed on top of the SUIF [HAA+96] compiler infrastructure. This compiler pass automatically generates a source code file "`helper_threads.c`" containing one function per p-thread, as well as a text file "`spawn_info.txt`" describing the spawn locations. Although the current study does not employ helper threads containing control flow instructions, the compiler pass (d) was coded using an algorithm to generate sequential code from the PDG representation of a helper thread [FMS88] (interested readers may want to explore employing a more general algorithm by Steengaard [Ste93]). The helper thread code is compiled separately from the main application using the SimpleScalar version of gcc (2.6.3), labeled (e) in Figure 5.8, and is then linked into the main program by the program loader in our modified version of SimpleScalar's `sim-outorder` simulator (f), which also uses information about the spawn point and live-in registers contained in "`spawn_info.txt`".

Several complications eventually led to the development of a modified infrastructure that does not rely upon a true compiler infrastructure, but rather a "postpass binary optimization technique" that leverages information from the slice tree data structure rather than true program slicing. The main difficulty with the compiler based infrastructure illustrated in Figure 5.8 was that of associating the information from the program slicer and profiling with the correct portion of the SUIF compiler intermediate representation. This lead to the generation of incorrect helper threads and limited the usefulness of the infrastructure. These limitations are not fundamental, but they were very time consuming to resolve. The main source of inaccuracy in the modified infrastructure is the use of perfect memory dissambiguation—the dynamic data addresses accessed by load and store instructions are used during the creation of the slice tree, which means the slice tree contains more accurate dependence information than is available in the CodeSurfer based program slicer. For example, CodeSurfer may not be able to tell whether a load instruction depends upon a store instruction and will be conservative in reporting such memory dependencies. An additional simplification made was to use the main thread and expected helper thread latency information captured during slice tree generation. The expected helper thead latency in the slice tree was determined using the actual instruction execution times from the main thread ordering instructions in schedule sequence using a simple list scheduling algorithm.

The modified infrastructure is illustrated in Figure 5.9. The program slicer and compiler components from the original infrastructure have been elimintated and replaced by the use of the slice tree to represent memory-based data dependencies.

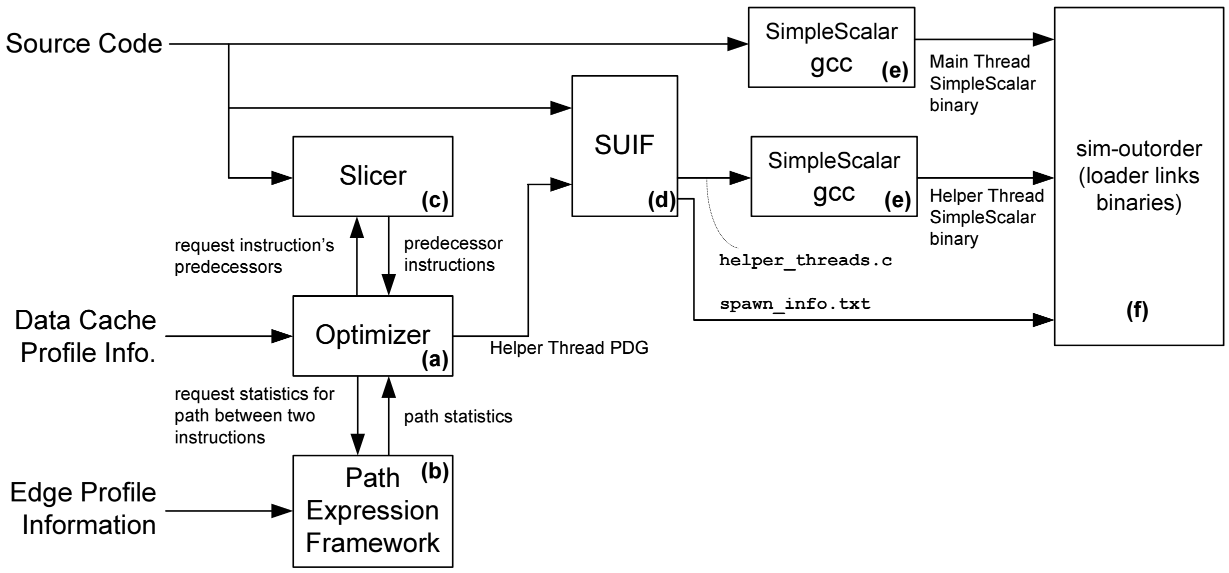


Figure 5.8: Inital version of the data prefetch slice enabling compiler infrastructure implementing the techniques described in this chapter.

5.4.2 Target Load Selection

With a framework that produces accurate predictions of performance improvement for helper threads it is possible to make good tradeoffs in the selection of which loads can most benefit from helper thread based prefetching. Since the aim of this investigation is to develop such a framework and in the process investigate the sources of inaccuracy in its predictions, the target loads of the helper threads are selected using a simple heuristic that estimates the improvement if a helper thread was always able to prefetch the load without consuming any resources. In practice, potential target loads would be selected based upon whether the selection algorithm found a helper thread that was able to obtain improvement in performance using available resources.

Helper threads consume execution resources and thus if not generated carefully may actu-

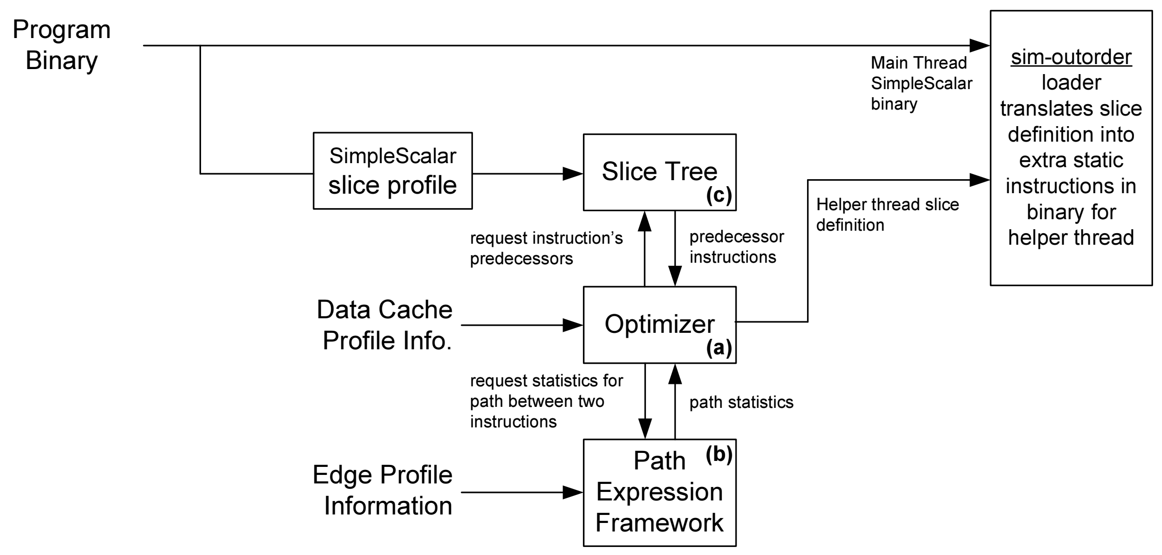

Program Binary
Main Thread SimpleScalar binary
sim-outorder loader translates slice definition into extra static instructions in binary for helper thread
SimpleScalar slice profile
Slice Tree (c)
Helper thread slice definition
request instruction's predecessors
predecessor instructions
Data Cache Profile Info.
Optimizer (a)
request statistics for path between two instructions
path statistics
Edge Profile Information
Path Expression Framework (b)

Figure 5.9: Modified helper thread infrastructure used to evaluate the techniques described in this chapter.

ally slow down the application. Careful generation of helper threads begins by selecting good candidate target loads. As in previous studies [CWT+01], we focus the search for good candidate target loads by examining so-called "delinquent loads"—loads that generate a large proportion of L2 cache misses (in this study we first narrow the search by examining the smallest subset of loads that together account for 90% of all L2 cache misses). A further property of importance is the sensitivity of performance to the individual cache misses [SL98a]. To establish the estimated impact on performance of prefetching for a given static load instruction the baseline simulator is augmented to reduce the latency of cache misses generated by the target load instructions to zero cycles. The resulting execution time is compared to the baseline performance and potential target loads are ranked by their potential speedup (see Table 5.4). This selection technique is merely a heuristic that assumes that the overhead of helper thread precomputation is roughly the same for each load. In practice, some loads have backward slices containing fewer instructions and/or can be executed faster in a helper thread than those of other loads and the helper thread optimization framework would have to take this into account when narrowing down the selection of target loads.

Table 5.4 lists up to the top five load instructions for each benchmark ranked in order of their estimated impact on performance. The potential speedup varies from 7% for vortex to 148% for treeadd indicating that these applications may benefit significantly from the use of data prefetch helper threads. In the following sections we assess how much of this potential speedup is realistically obtainable and how accurately the speedup from a given helper thread is predicted by the proposed analysis framework.

#	Bench.	Target Load	Frequency	Speedup	L1 miss rate	L2 miss rate*	Sensitivity
1	art	0x404c80	1067500	1.396	50%	50%	1.01
2	art	0x404c98	1067500	1.357	100%	100%	0.25
3	art	0x4045e8	567943	1.030	98%	100%	0.07
4	art	0x404358	284532	1.002	62%	98%	0.02
5	art	0x404c88	1067500	1.000	100%	100%	0.00
6	bh	0x403000	322628	2.161	100%	88%	1.01
7	bzip2	0x40bad0	410549	1.285	67%	65%	0.85
8	bzip2	0x40aca0	174995	1.064	50%	45%	1.07
9	bzip2	0x40abc0	174995	1.060	48%	43%	1.07
10	bzip2	0x40ab50	174995	1.056	47%	42%	1.07
11	bzip2	0x40ac30	174995	1.056	46%	42%	1.07
12	mcf	0x4009f8	2546268	1.294	57%	75%	0.66
13	mcf	0x400a18	1269615	1.000	100%	96%	0.00
14	mcf	0x400a50	1276653	1.000	99%	99%	0.00
15	treeadd	0x400508	1048575	2.483	50%	50%	1.10
16	vortex	0x4880e8	28965	1.072	121%	82%	1.00
17	vortex	0x4673d8	114655	1.061	23%	97%	0.94
18	vortex	0x466668	263188	1.025	10%	45%	0.82
19	vortex	0x45dc00	15027	1.012	63%	51%	1.01
20	vortex	0x461958	424163	1.011	6%	33%	0.55
21	vpr	0x41e5e8	394677	1.092	21%	81%	1.04
22	vpr	0x41dc38	152291	1.054	65%	77%	0.59
23	vpr	0x41e6b0	384331	1.035	63%	78%	0.18
24	vpr	0x41f400	377369	1.028	27%	72%	0.34
25	vpr	0x41f570	1905810	1.020	14%	71%	0.08

Table 5.4: Target Load Selection (“Sensitivity” listed in this table is defined in Section 5.5.1)

5.4.3 Framework Accuracy

To assess the accuracy of the data prefetch helper thread optimization framework, helper threads were generated for the selected target loads using the modified infrastructure illustrated in Figure 5.9 using second-order control flow profile information. The resulting “actual” speedup obtained on the sim-outorder based SMT simulation is plotted compared with the speedup “predicted” by the path expression framework in Figure 5.10. This plot shows far less correlation between prediction and outcome than observed for metrics such as reaching probability, expected path length, etc... in Figure 2.7 in Chapter 2. The data points for a given benchmark are plotted with the same symbol to illustrate how much correlation exists for different helper threads for the same target load.

For example, the symbol used for bzip2 is a small triangle and the data points for bzip2 form

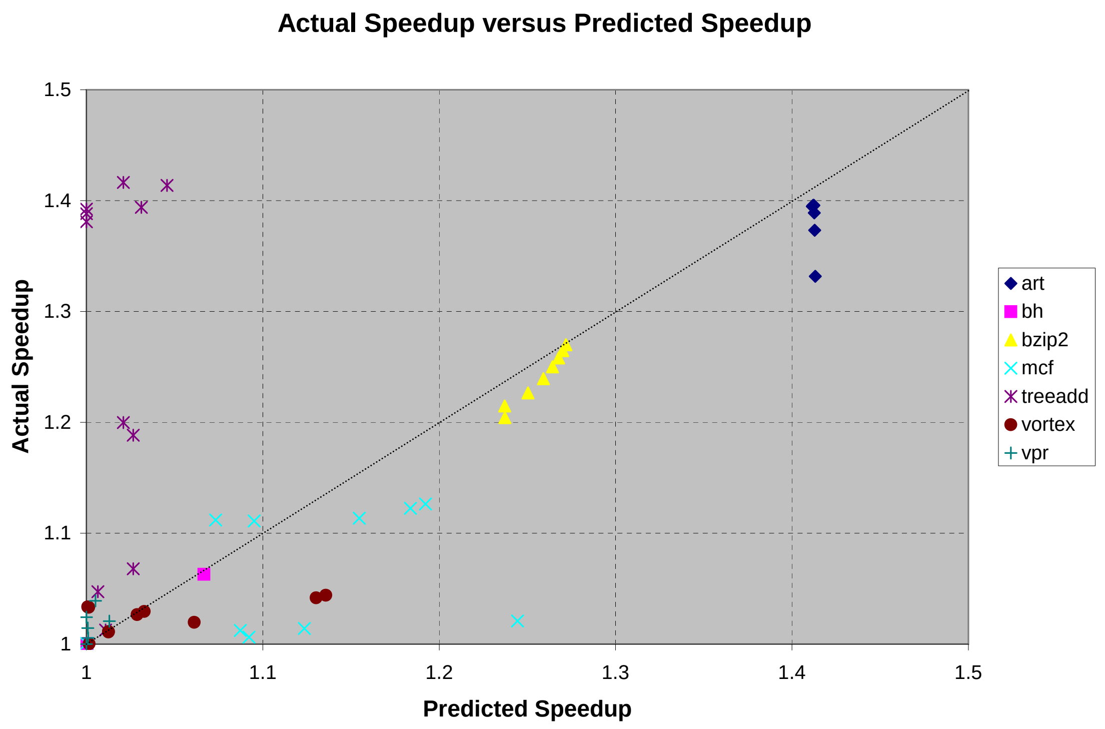


Figure 5.10: Framework Accuracy for Predicting Benefit of Data Prefetch Helper Threads

a line with positive slope close to the line $y = x$. On the other hand, for treeadd the data points do not fall on a line but rather have greater actual speedup than predicted (points mostly above the line $y = x$). Figure 5.12 shows another view of the data in Figure 5.10. In this figure the actual and predicted speedup is plotted versus the length of the helper thread used to prefetch the target load for each of the benchmarks. This figure shows very good agreement of predicted and actual speedup on 179.art, bh, and 256.bzip2, and illustrates the differing ways in which predictive accuracy breaks down as slice length increases for the other benchmarks.

To compare prediction accuracy with aggregate advantage, we compute the correlation coefficient of the data points for each benchmark using an enhanced version of the aggregate advantage technique and the second order path expression based prediction technique. The results are plotted in Figure 5.11, and show that on average the path expression based technique has correlation coefficient of 0.65, which is fairly close to the value 0.74 obtained using aggregate advantage.

The rest of this chapter explores reasons for poor correlation between actual and predicted speedups with data prefetch helper threads. This analysis points the way toward future research in improved feedback compiler based optimization frameworks for helper threads.

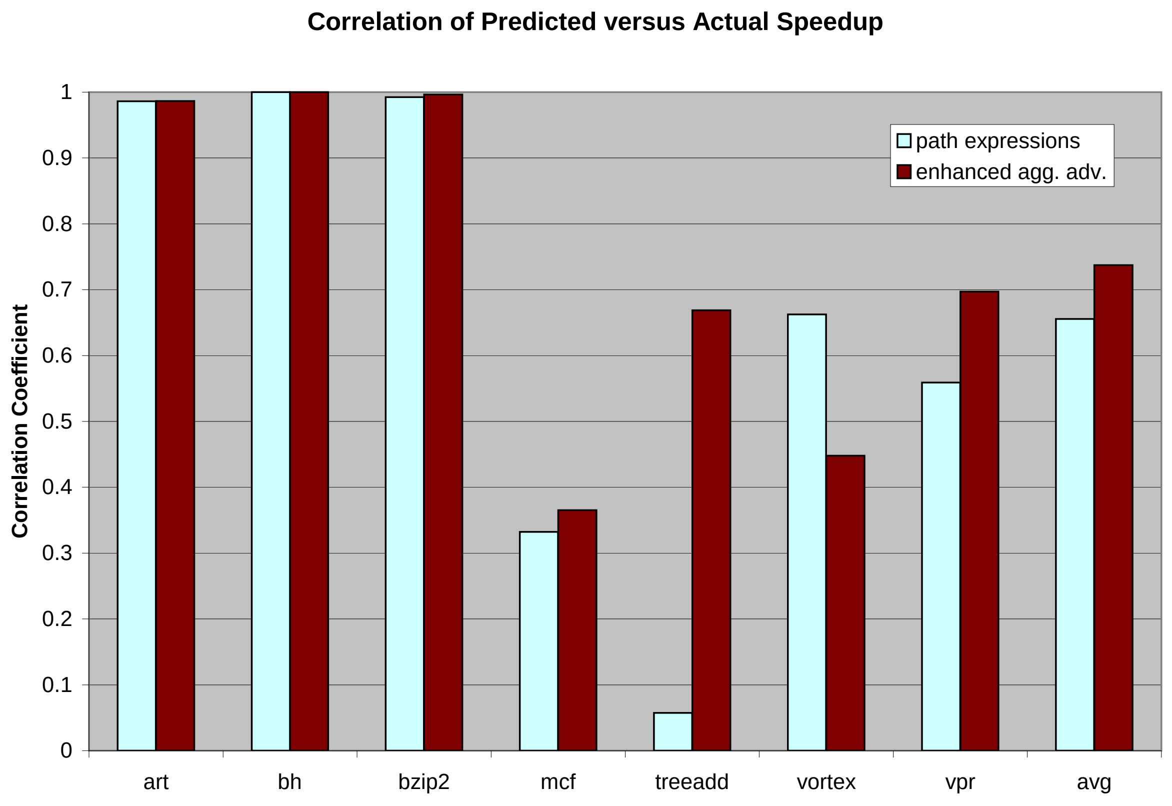


Figure 5.11: Comparison of path expression based framework to aggregate advantage framework

5.5 Sources of Modeling Error

In this section several sources of modeling error are analyzed.

Starting in the top row of Figure 5.12 we see that the predictions for 179.art and bh are quite accurate as are the predictions for 256.bzip2 in the next row. On the other hand, for 181.mcf the accuracy starts out poorer and then improves, while the predictions for treeadd are initially much too low and while increasing somewhat as slice length increases, they remain smaller than the actual speedup obtained by the helper threads. For vortex, the predictions closely match for shorter helper threads then begin to diverge with the predicted improvement being greater than the actual improvment. Finally, on the last row the predicted improvement for 175.vpr falls below the actual improvement.

A detailed investigation into the sources of modeling discrepancies was conducted as the proposed modeling framework was developed. The main sources of error can be summarized as follows:

prefetch slack sensitivity Superscalar processors employing out-of-order execution can significantly hide the latency of operations by scheduling around execution stalls due to events such as L1 data cache misses, that happen off the critical path through the computation [SL98a]. A result of this effect is that each cycle earlier a target delinquent load is prefetched does not necessarily save a cycle of execution time. As shown later, the impact is roughly equivalent to a constant multiplicative factor applied to the minimum of the prefetch slack and the memory latency.

prefetch slack distribution The prefetch slack achieved by a given p-thread tends to vary from one instance of the helper thread to the next. Due to the nonlinearity in the transfer function between prefetch slack and execution time savings, measuring only the average prefetch slack can lead to significant inaccuracies in characterizing the impact on execution time.

branch outcome correlation The path expression framework described in Chapter 2 assumes that branch outcomes are uncorrelated. This approximation leads the modeling framework to underestimate the frequency of execution paths from the spawn point to the target delinquent load causing the benefit of the associated helper threads to be underestimated.

spatial locality / loss of temporal locality Typical memory systems use caches that store several words of memory in a single block (or cache line). This organization helps take advantage of the observation that many memory accesses are to a location that is near to one that was recently accessed. The slice of a delinquent load may contain other loads. The impact of spatial locality is that during application profiling these other loads may appear to have a high hit rate, but when these loads are executed in a helper thread they may trigger cache misses. The impact of this is that the helper thread takes longer than expected to execute and does not initiate the prefetch of the critical target load as early as predicted by the modeling framework. This effect was observed for 181.mcf.

resource constraints The proposed model does not estimate the number of helper threads running concurrently. This was found to have significant impact on the accuracy of the predictions for 256.bzip2 and 175.vpr.

equivalent linked data traversals In at least one case (181.mcf) it was observed that significant prefetch benefit was obtained even when the main thread did not follow the "implied control flow" assumed by a helper thread after the spawn point. The reason for this *appears* to be that although the application would begin by traversing a different path through the linked data structure associated with the target load of the helper thread, it would later return back to the same node visited when the helper thread was spawned, and then followed the control flow assumed by the helper thread.

correlation of cache misses to control flow paths The assumption in Equation 5.6, that the probability of a cache miss is independent of the control flow leading up to the load instruction does not always hold in practice. For example, the probability of a cache miss occuring when the main thread follows the implied control flow of the p-thread starting at B is 0.766 versus a average L2 cache miss rate 0.432 for the target load. Similar observations have been made by others [ML97, Luk00].

In the following subsections we present data to quantify the magnitude of many of these effects.

5.5.1 Prefetch Slack Sensitivity

Figure 5.13 plots the relationship between the prefetch slack for target loads that miss in the L1 cache, and the resulting reduction in execution time. The reduction is typically less than 1 cycle of execution time saved per cycle of prefetch slack. Furthermore, as expected, there is no benefit to prefetching with slack larger than the latency to access memory (300 cycles). This effect can be captured by the various prediction models by ensuring that the x cycles of predicted prefetch slack result in $y = mx$ cycles of execution time savings if a load misses in the cache for x up to, but not more than the miss penalty—the L2 access time (14 cycles) for L1 cache misses that hit in the L2, or the main memory latency (300 cycles) for loads that miss in the L2 as well as the L1 cache. The *prefetch slack sensitivity* is defined as the slope of a best fit line to the speedup/prefetch-slack function in Figure 5.13 from zero prefetch slack to the main memory latency of 300 cycles.

The magnitude of the prefetch slack sensitivity was measured and this senstivity was incorporated in the predictions made by both the enhanced aggregate advantage prediction technique and the path expression based prediction technique summarized in Section 5.4.3.

5.5.2 Prefetch Slack Distribution

Figure 5.14 plots a histogram of the prefetch latencies of the best p-thread found for the selected target load in 175.vpr. This distribution is multimodal and shows that a significant fraction (18.3%) of the prefetches have slack less than the latency of memory (300 cycles). The isolated and well defined peaks in this distribution may indicate that the main thread iterates around a loop a variable number of times before reaching the target load. In principle it should be possible to capture this effect with a more sophisticated version of the "path length variance" mapping that takes into account distributions rather than just a few statistical parameters.

Using a simple gaussian approximation to this prefetch distribution yields a significant prediction error of approximately 13% due to the nonlinearity of the the sensitivity of execution time to prefetch slack.

The enhanced aggregate advantage technique measures the slack of each instance of a slice and compares it against the nonlinearity of the prefetch slack sensitivity function rather than first averaging the values. The path expression based technique uses the average and standard deviation of the estimated helper thread latency measured during slice tree extraction and applies a gaussian approximation to obtain the resulting predicted speedup.

Note that the significantly poorer prediction accuracy of the path expression framework for the benchmark treeadd seen in Figure 5.11 can be attributed to a prefetch slack distribution that has an average near zero and variance that does not enable a simple gaussian model to capture the beneficial impact of several of the helper threads under consideration.

5.5.3 Branch Outcome Correlation

Figure 5.15 plots the predicted speedup for several helper thread modeling techniques including the use of second-order control flow information (described in Section 5.3) to account for the impact of correlation among branch outcomes.

The prediction techniques evaluated in Figure 5.15 are as follows: *1st order 1:1* plots the path expression modeling prediction with prefetch sensitivity assumed to be one cycle improvement for each cycle of prefetch slack (for cache misses), up to the latency of main memory; *1st order* includes the impact of using the prefetch sensitivity measured earlier; *1st order-g* also includes the impact of prefetch slack distribution assuming a gaussian latency distribution; *2nd order-g* uses the second order control flow path expression model, prefetch slack sensitivity and the gaussian latency distribution model; *slice tree avg* represents an enhanced version of aggregate advantage formulation that includes the effects of prefetch slack sensitivity; and *slice tree dyn* accounts for the effects of prefetch slack distribution (this is the same data presented earlier as the enhanced aggregate advantage technique). Finally, these results are compared against the actual speedups obtained using the helper threads (labeled *actual* in Figure 5.15).

Focusing on the use of second order control flow information in the data labeled *2nd order-g*, both 181.mcf and 255.vortex were found to benefit from the application of this enhancement for modeling the impact of correlated branch outcomes.

5.5.4 Spatial Locality / Loss of Temporal Locality

Figure 5.16 plots the predicted and actual speedups assuming a memory hierarchy that stores data in 4-byte blocks, i.e., without grouping together contiguous locations into cache blocks, also using the same modeling techniques as in Figure 5.15. The use of 4-byte blocks eliminates the impact of spatial locality which is not modeled by the proposed techniques. The benchmarks that show the most significant change are treeadd and 181.mcf. The behavior for 181.mcf was investigated in detail and it was found that the profile data for 32B cache lines did not accurately predict the prefetch slack of the helper threads. The reason was that a load in the helper thread that almost always hit when executed as part of the main thread, was found to miss when executed by the helper thread in advance of another memory access that was not in the slice. When the cache line size is instead set to 4B (the size of a pointer), this effect is eliminated and the predictions for helper thread latency are closer to the actual observed latency.

5.5.5 Resource Constraints

Figure 5.17 and 5.18 illustrate the impact of making additional thread contexts available to run the selected helper threads for 256.bzip2 and 175.vpr, respectively. In both cases as additional thread contexts are made available the resulting speedup is closer to that predicted using the proposed modeling technique.

To provide a better sense of the relative importance of the various factors highlighted above, we again turn to the correlation coefficient. Figures 5.19 and 5.20 plot the correlation coefficient of the predicted and actual speedups for the helper threads computed per benchmark. The average of these values is also plotted to highlight the aggregate effect (grouping data for all benchmarks and then taking the correlation coefficient produces slightly different values but the trends remain the same).

Of the compiler implementable prediction techniques, the use of second-order control flow profile information, prefetch slack sensitivity and latency distribution information provided the most accurate predictions. All first order control flow techniques had roughly the same prediction accuracy for 32B cache lines, however for 4B cache lines, the impact of latency distribution and prefetch slack sensitivity was more pronounced.

5.6 Summary

This chapter proposes and evaluates a technique for optimizing simple p-threads using control flow profile information and the path expression modeling framework described in Chapter 2. This technique combines the effectiveness of the aggregate advantage optimization framework [RS02] with the efficiency and compiler friendly implementation advantages of our path expression framework. The control flow modeling approach introduced in Chapter2 is extended to incorporate a limited amount of branch outcome correlation by profiling the outcome of pairs of consecutive branches. This improves the accuracy of the predicted speedup, as does more accurately modeling of the distribution of prefetch latencies. Further research incorporating more realistic assumptions on memory dissambiguation (perhaps including probabilistic pointer analysis techniques [JCO98, CHH$^+$03b, CHJL04]) is needed to evaluate the practicality of the proposed approach in the setting of an optimizing compiler. Also, exploration of methods of accurately predicting dataflow events such as equivalent linked data structure traverals may yield improvements in prediction accuracy.

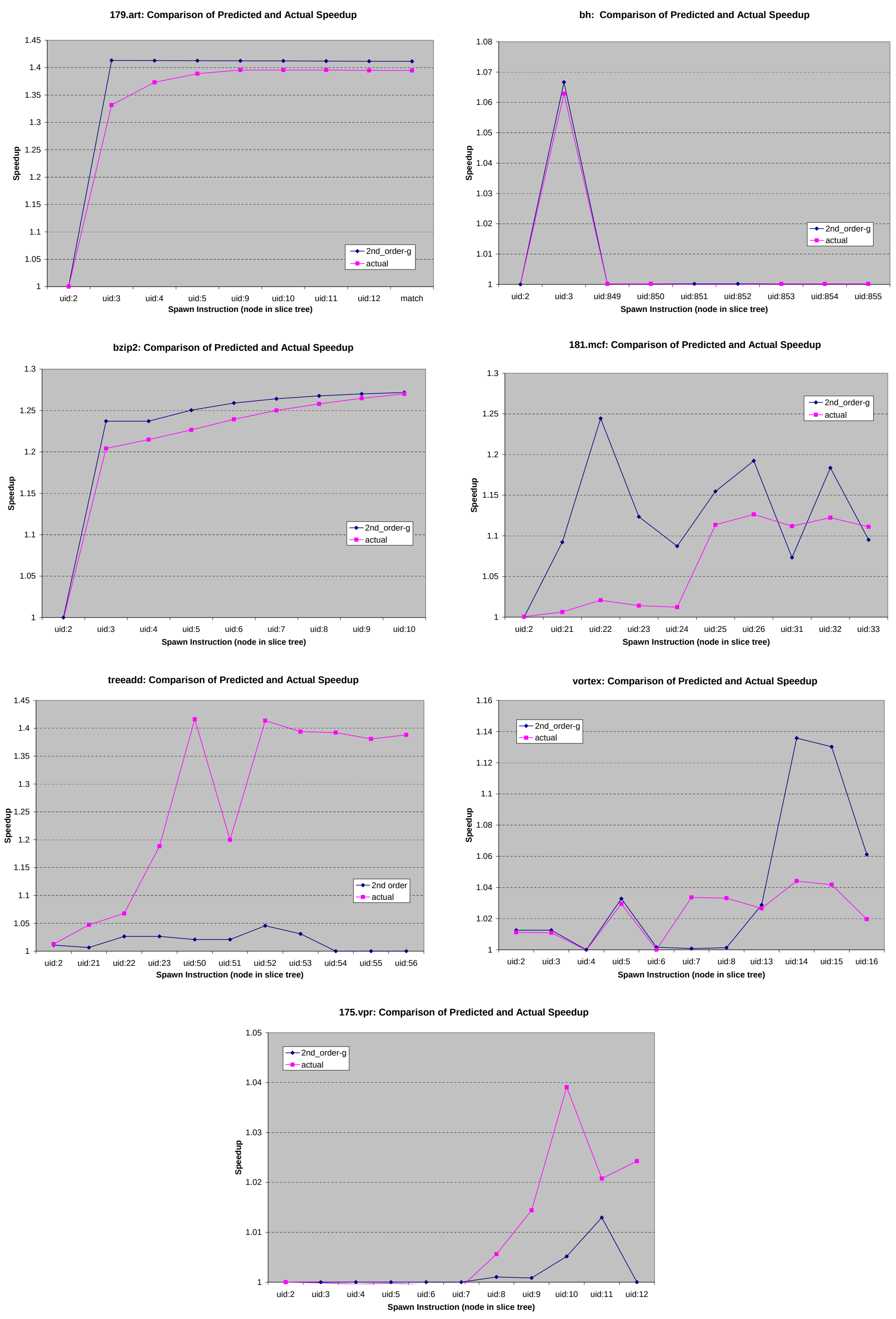


Figure 5.12: Predicted and Actual Speedups with 32 Byte Data Cache Lines

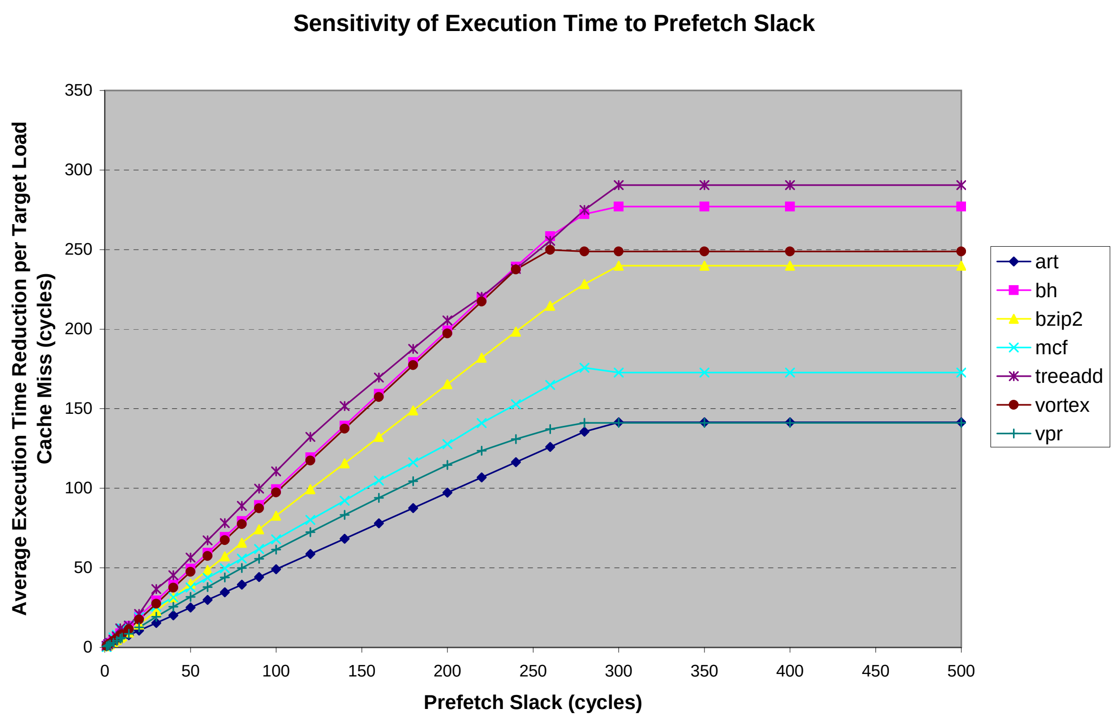


Figure 5.13: Prefetch Slack Sensitivity

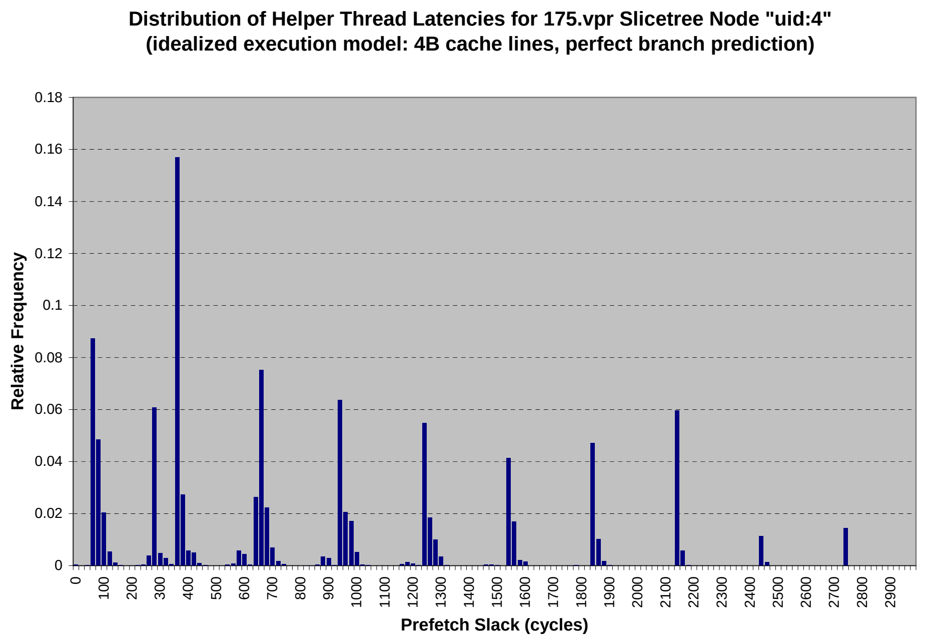
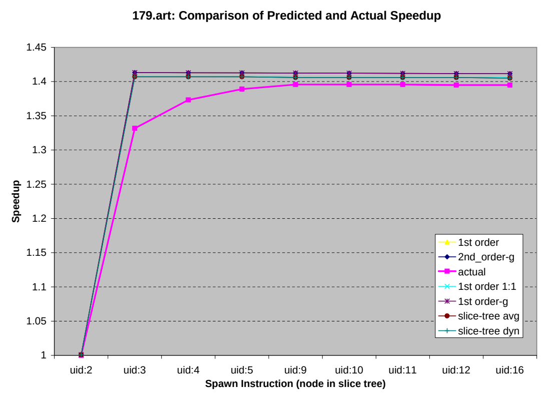
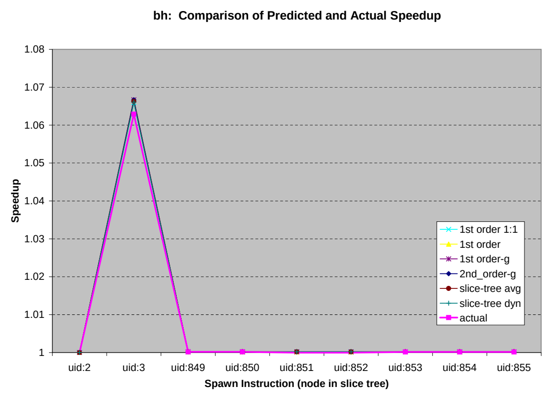
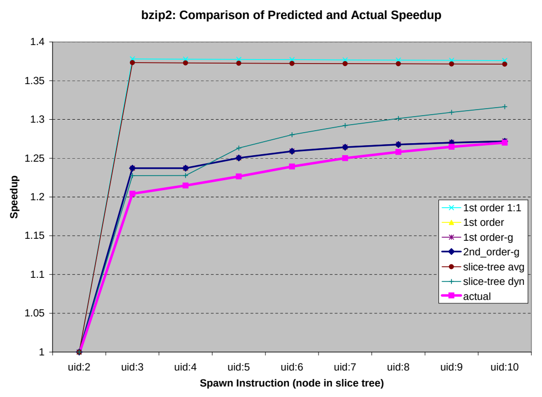
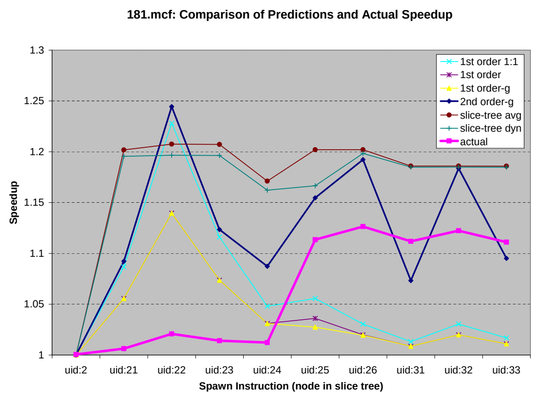


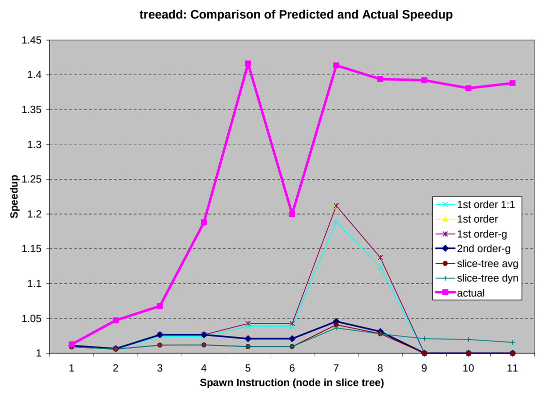
Figure 5.14: Helper Thread Prefetch Slack Distribution for 175.vpr

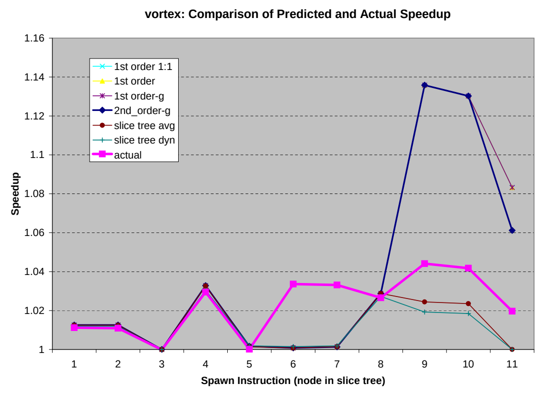












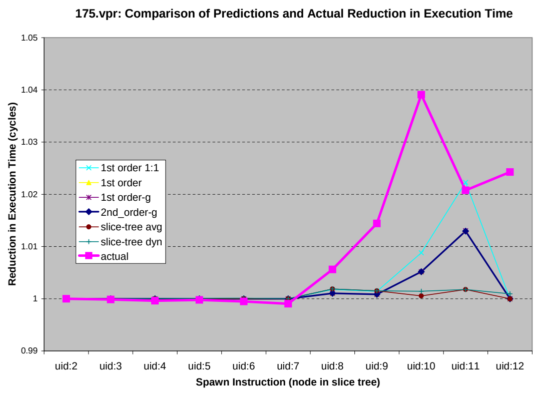


Figure 5.15: Predicted and Actual Speedups with 32 Byte Data Cache Lines

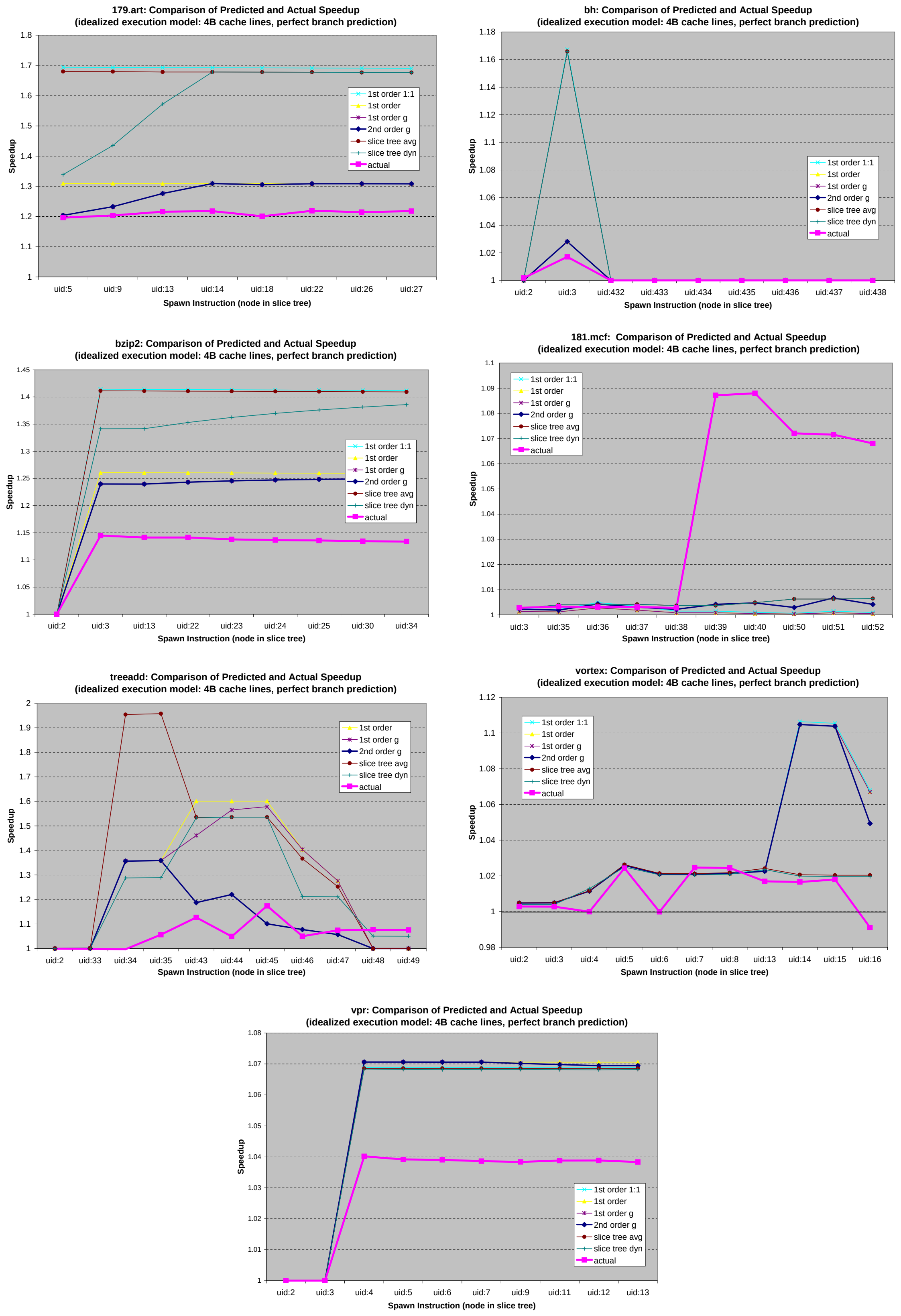


Figure 5.16: Predicted and Actual Speedups with 4 Byte Data Cache Lines and Perfect Branch Prediction

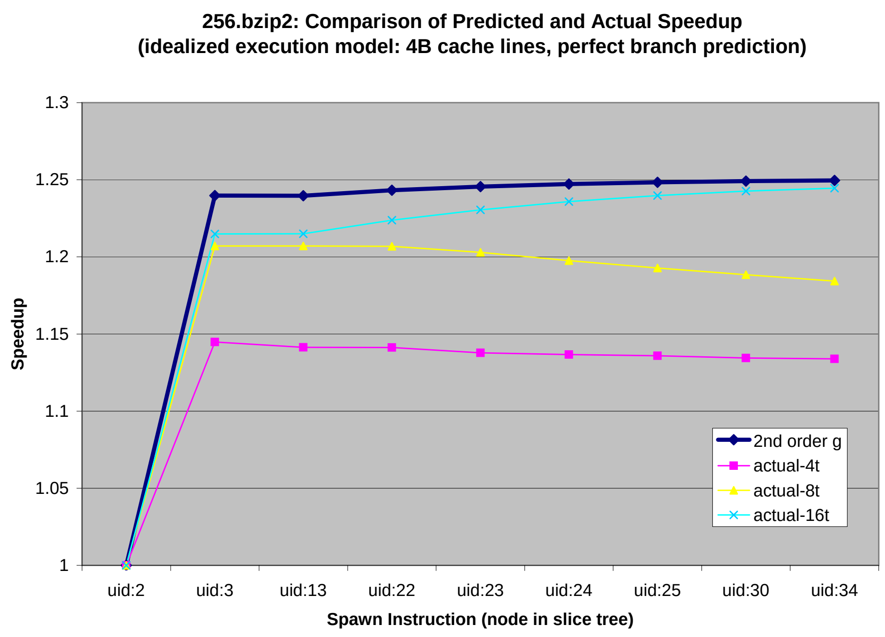


Figure 5.17: Increasing Thread Resources for 256.bzip2 Reduces Prediction Error

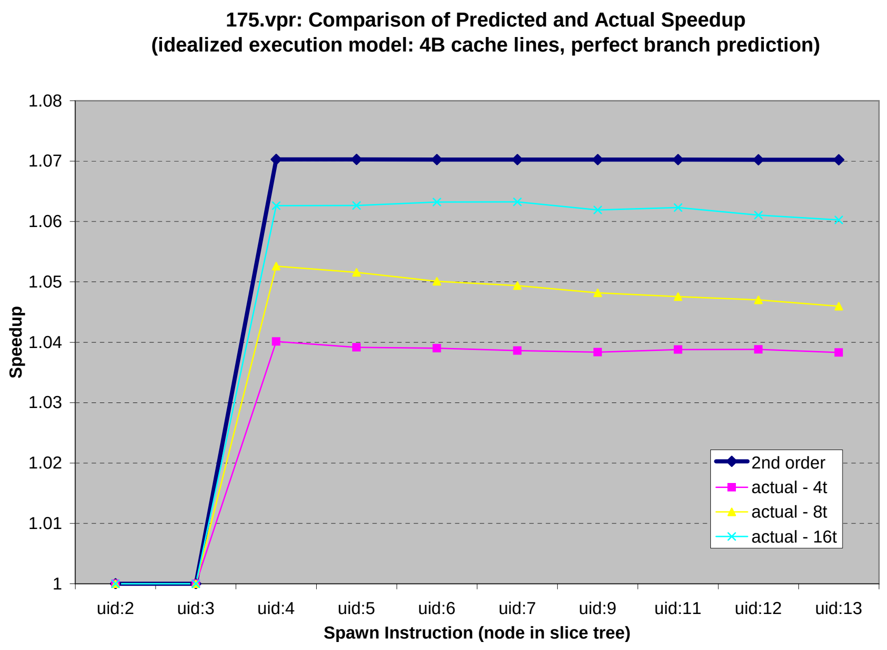


Figure 5.18: Increasing Thread Resources for 175.vpr Reduces Prediction Error

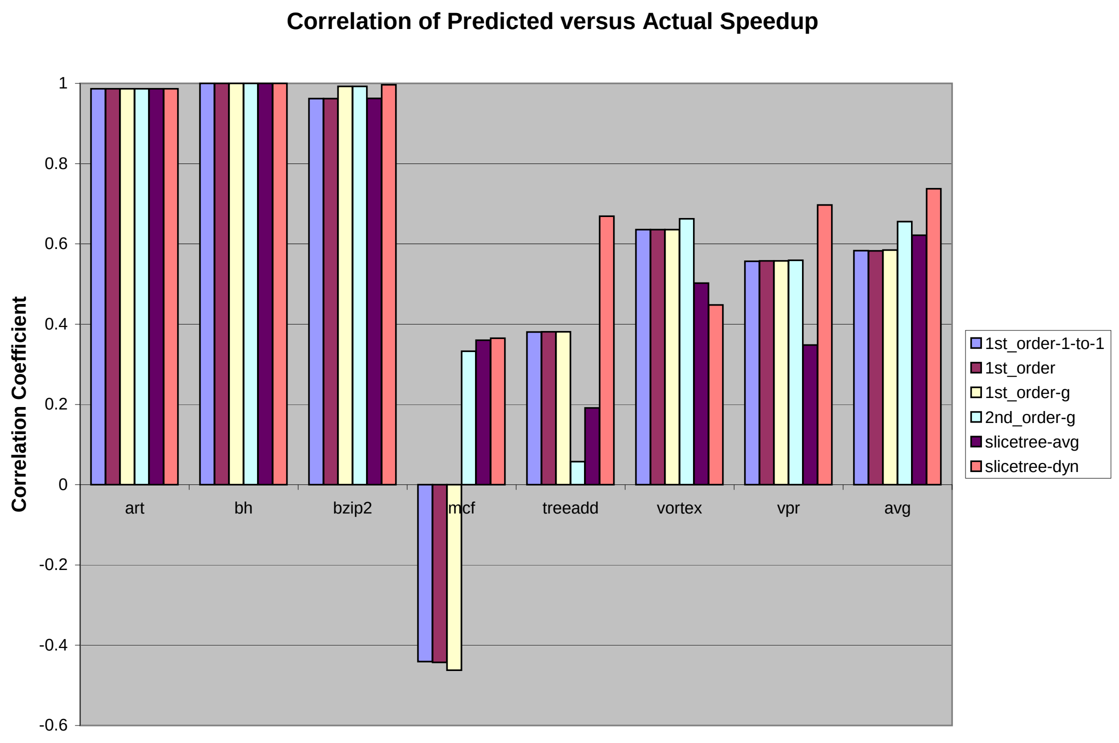


Figure 5.19: Correlation of Predicted and Actual Speedups

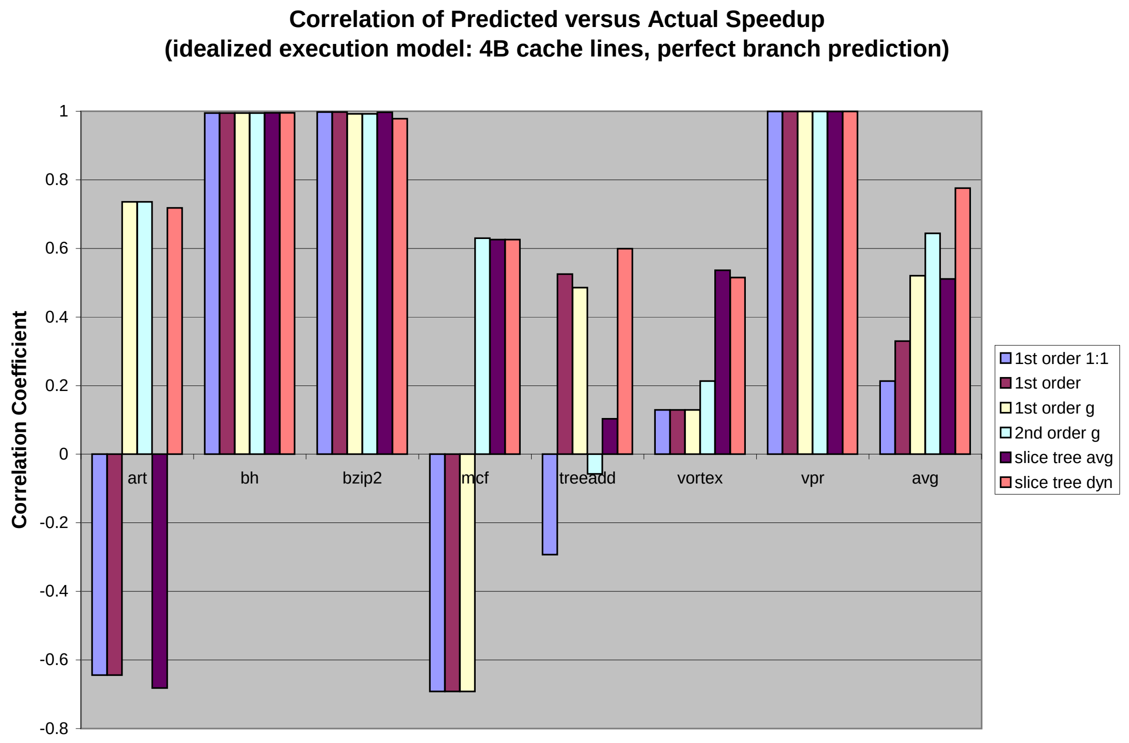


Figure 5.20: Correlation of Predicted and Actual Speedups (4B data cache lines, perfect branch prediction)

Chapter 6

Related Work

This chapter describes related work relevant to this dissertation to place this work in the context of previous and contemporaneous work. In Section 6.1 related work on modeling microprocessors and program behavior is described. Section 6.2 summarizes related work on instruction and data prefetch techniques. Section 6.3 covers related work on speculative multithreading. Finally, in Section 6.4 related work on helper threads is described.

6.1 Modeling Techniques and Applications

This section describes related work on the modeling of microprocessors and program behavior. These techniques have been proposed both to reduce simulation time when designing microprocessors, to provide insight into design tradeoffs, and to improve compilation. The framework proposed in this dissertation essentially attempts to do the latter in a more general and powerful way than previous proposals, and applies such a modeling framework to microarchitecture techniques (helper threads) in a way that has not been extensively investigated before.

6.1.1 Modeling for Program Optimization

Sarkar [Sar89] presents an algorithm for estimating the average and variance of the execution time through a portion of a program by analyzing control flow profile data. The algorithm is applicable to interval regions of non-reducible control flow and was proposed as a way to guide the partitioning of a program into parallel tasks in the PTRAN system [ABC$^+$88]. The framework proposed in Chapter 2 of this dissertation can be viewed as an elegant and efficient unification and generalization of the computation of these quantities with the reaching probabilty calculation of Marcuello Gonzalez [MG02]. The proposed modeling framework is more general than the algorithm proposed by Sarkar [Sar89] because it can be applied to determine statistical quantities between *any* two points in a program rather than just the start and end points of a control flow interval. Further, the control flow does not need to be reducible. Also, in addition to the reaching probability and the path length mean and variance, the proposed framework provides a means of computing the posteriori probability and the expected path footprint. The former is vital to quantifying the coverage of performance degrading events provided by a helper thread, and the latter helps ensure that spawn-target selection does not lead to instruction cache thrashing.

Contemporaneous with the development of the modeling and optimization framework presented in Chapter 2 of this dissertation (and first presented in Aamodt et al. [AMC$^+$02, AMC$^+$03]), Roth and Sohi [RS02] introduced a framework for automated selection of static p-threads (similar to our notion of a simple data prefetch helper thread). The framework consists of a data structure called the *slice tree*, and a formulation of the performance impact of p-threads they call the *aggregate advantage*. These two components are used together to optimize the selection of p-threads by abstractly executing the p-thread to estimate the performance impact. The slice tree contains a record of the dynamic instructions that affect the outcome of repeated executions of a *problem load*. The problem load is the root of the slice tree, and is a load instruction with an address stream that defies conventional address prediction mechanisms (such as per load-address stride prediction [FCJV97]). The aggregate advantage is used to select a slice from the slice tree based upon its estimated impact on execution time. It is based upon the following program statistics that are assumed to be either collected from trace data or determined in some other way: the dynamic execution count of each potential *trigger instruction* used to launch a p-thread (similar to the notion of a spawn instruction described in this dissertation); the number of times a particular sequence of instructions is seen from the trigger to the problem load; and the number of instructions between the trigger and the problem load encountered in the main thread.

6.1.2 Modeling to Predict Microprocessor Performance

Another related area of research is the use of modeling to predict microprocessor performance with the goal of speeding up microprocessor design. Such techniques are motivated because of the growing disparity in the number of instructions executed by a contemporary benchmark application compared with the speed of the simulators used to model the ever more complicated microarchictures of modern processor designs. For example, many SPEC CPU 2000 benchmarks execute several 100 billions of instructions whereas the popular SimpleScalar [BA97] microprocessor simulator achieves a simulation rate of roughly 25,000 instructions per second on contemporary desktop computer systems. At this rate it could take several months to simulate a complete benchmark. Given that processor design is an interative process there is a need to more quickly estimate the performance impact of proposed changes. In the context of this dissertation these techniques are of relevance because the helper thread optimization techniques proposed in Chapter 3 and 5 also employ a modeling framework to estimate microprocessor performance.

Jouppi [Jou89] studies the distribution of program and machine parallelism and suggests a simple way in which they can be used to estimate performance. The essential obervation is that to a first approximation, ILP grows linearly as machine parallelism is increased until it reaches a limit established by program parallelism.

Noonburg and Shen [NS94] build on this work by presenting a more elaborate theoretical model of superscalar performance that accounts for the interactions of program parallelism and machine parallelism in greater detail. Their formulation is based upon a decomposition of program parallelism into control and data parallelism, and a decomposition of machine parallelism into branch, fetch, and issue parallelism. A simple probability distribution is used to represent the parallelism (ILP) at the input (branch parallelism) and at stages within the processor. A series of matrix transfer functions are used to represent the effects of the different forms of machine and program parallelism as instructions flow through the processor.

More recently, Karkhanis and Smith [KS04] proposed a first-order model of superscalar performance that takes into account branch mispredictions and cache misses and achieves accuracy within 5.8% of full simulation on average. The model is based upon first estimating a steady-state IPC, then subtracting the penalty of performance degrading events (i.e., cache misses and branch mispredictions). The later is computed by first estimating the impact of each type of performance degrading event on the number of instructions executed over the duration of an event of that type, then multiplying by the frequency of these events in the dynamic instruction stream.

6.1.3 Sampling

Among techniques that attempt to limit the amount of detailed simulation are the works of Ofelt and Hennessy [OH00], who describe a technique for estimating the running time of an application on a modern superscalar microarchitecture by profiling paths and weighting the simulated execution time of each path, or, to achieve greater accruacy, a short sequence of paths by their dynamic execution frequency. When branch prediction and caches are ignored, their approach provides estimates within 1.25% of full simulation time in several orders of magnitude less time.

More recently, the Simpoints [SPHC02a] and the SMARTS [WWFH03] frameworks have been proposed to aid researchers and processor designers in finding representative regions to simulate out of the entire dynamic instruction stream of a program run.

6.2 Prefetching

In this section related work on instruction and data prefetching is summarized. In *next-N-line* instruction prefetch [Smi78, Smi82] the hardware prefetches N cache lines ahead of the fetch stage. In *target-line prefetching* [SH92] the hardware maintains knowledge of the target of taken branches and uses this to initiate instruction prefetches. In cooperative prefetching [LM98], the compiler and hardware cooperate to improve the effectiveness of instruction cache prefetching by selectively filtering prefetches. In fetch directed instruction prefetch [RCA99], the branch predictor streams ahead of the fetch and prefetch hardware by using a decoupling queue and a special form of branch target buffer called the fetch target buffer.

Data prefetching techniques include hardware only, stride-based data prefetch techniques [Jou90, BC91, PK94, FJC95] and software controlled prefetching [Por89, MLG92].

These techniques, while effective, do not cover all instruction and data cache misses. Helper thread based prefetch techniques, such as those proposed in Chapter 3-4 for instructions and those explored in Chapter 5 for data, tackle those cache misses which tend to foil the above mechanisms.

6.3 Speculative Multithreading

In this section related work on speculative multithreading is summarized. This body of work is related to the prescient instruction prefetch helper threads studied in Chapter 3 and 4 insofar as it exploits both control-independence (sometimes referred to as control-equivalence) to skip ahead in program execution, and bandwidth readily available for multiple fetch streams on multithreading processors. In the Multiscalar [SBV95] architecture, computation is divided up among processing elements that communicate via a ring bus. Data speculation is supported using special hardware structures [FS96, MBVS97, GVSS98], and the division of a single threaded application into tasks is managed by the compiler [VS98]. In *thread-level speculation,* [SM98, SCZM00] memory dependencies between tasks (or epochs) are determined by extending invalidation based cache coherence protocols. In *speculative multithreading* [MGT98], the hardware dynamically creates threads using control flow heuristics.

The use of precomputation for producing live-ins to a future task has been explored in *master/slave speculative parallelization* [ZS02]. However, whereas these techniques primarily achieve speedup by exploiting thread-level parallelism available by reusing the architectural results of speculative computation, prescient instruction prefetch improves performance by reducing memory latency, in particular by reusing the microarchitectural side-effects (i.e., cache warm-up) of speculative precomputation threads.

Whereas *trace preconstruction* [JS00] employs a hardware-based breadth-first search of future control-flow to cope with weakly-biased future branches, prescient instruction prefetch uses precomputation to resolve which control-flow path to follow. Furthermore, as the precomputation frequently contains load instructions, prescient instruction prefetch often improves performance by prefetching data.

6.4 Helper Threads

In contrast to branch predictor guided instruction prefetch mechanisms such as *fetch directed instruction prefetch* [RCA99], prescient instruction prefetch uses a macroscopic view of control-flow gained by a profile-driven statistical analysis of program behavior. Such global analysis allows helper threads to accurately anticipate potential performance degrading code regions from a long distance ahead. For instance, the distance between spawn and target could be thousands of dynamic instructions.

Dubois and Song proposed assisted execution as a generic way to use multithreading resources to improve single threaded application performance [DS98]. Chappell et al. proposed Simultaneous Subordinate Multithreading (SSMT), a general framework for leveraging otherwise spare execution resources to benefit a single-threaded application. They first evaluated the use of SSMT as a mechanism to provide a very large local pattern history based branch predictor [CSK^{+}99], and subsequently proposed hardware mechanisms for dynamically constructing and spawning subordinate microthreads to predict difficult-path branches [CTYP02]. Weiser [Wei81] proposed *program slicing* to find the subset of a program impacting the execution of a particular statement to aid program understanding. Combining these themes, Zilles and Sohi proposed analyzing the dynamic backward slice of performance degrading instructions so as to pre-execute them [ZS00]. They subsequently implemented hand-crafted speculative slices to precompute branch outcomes and data prefetch addresses [ZS01].

Roth and Sohi [RS01] proposed using data driven multithreading (DDMT) to dynamically prioritize sequences of operations leading to branches that mispredict or loads that miss. Concurrent with our work, they propose a framework for optimizing the selection of slices for loads that miss in the second-level cache based upon an analysis of program traces [RS02]. Moshovos et al. proposed Slice Processors, a hardware mechanism for dynamically constructing and executing slice computations for generating data prefetches [MPB01]. Annavaram proposed Dependence Graph Precomputation [APD01], which effectively uses the predicted stream of instructions to produce accurate dynamic slices. Luk proposed software controlled preexecution [Luk01] as a mechanism to prefetch data by executing a future portion of the program.

Collins et al. proposed speculative precomputation [CWT$^+$01], and later dynamic speculative precomputation [CTWS01] as techniques to leverage spare SMT resources for generating long range data prefetches by executing slices that compute effective addresses for loads that miss often. They showed that chaining the precomputation of a load that repeatedly misses can help tolerate long memory latencies. Wang et al. [WWC$^+$02] studied the impact of speculative precomputation on in-order and out-of-order processors, and argue that delinquent loads can be categorized into two classes of criticality [SL98b] by whether or not they appear as part of the associated loop control computation (assuming the delinquent load is part of a loop). Liao et al. extended the work of Collins et al. [CWT$^+$01] by implementing a post-pass compilation tool to augment a program with automatically generated precomputation threads for data prefetching [LWW$^+$02].

One task common to all these techniques is the selection of a trigger point where a helper thread should be spawned off for precomputation leading to a target branch or load. As a matter of convenience, we call this trigger the *spawn* point. In contrast to helper threads for loads and branches, for instruction prefetch the *target* identifies not just a single instruction, but rather the beginning of an entire region of code that the main thread is likely to execute soon. A similar notion of a *spawn-target pair* can be found in speculative multithreading, where a thread is forked to speculatively execute a future portion of the program with the goal of quickly reusing the architected state it generates when the main catches up with it. Traditionally, control flow idioms such as loops and procedure calls have been exploited for identifying spawn-target pairs. Generalizing the near control equivalence between spawn and target common to such idioms, Marcuello and Gonzalez [MG02] proposed using a reaching probability threshold to define a far larger set of candidate spawn-target pairs from which thread-level parallelism may be harvested. They considered refining the selection on the basis of maximal spawn-target distance, minimum inter-thread dependencies, and value predictability.

Chapter 7

Conclusions and Future Work

This chapter concludes the disseration. Section 7.1 provides a brief summary and conclusions, Section 7.2 lists the contributions made by this dissertation, and Section 7.3 indicates important areas for future work in this area.

7.1 Summary and Conclusions

This dissertation proposes and evaluates techniques for improving microprocessor performance. As the gap between main memory latency and processor cycle time continues to widen with each process technology advance, the performance impact due to cache misses will increase. Prefetching is a technique that improves the effectiveness of the memory hierarchy by bringing instructions and/or data into the small but fast cache hardware structures *before* a cache miss is triggered by an application. By overlapping the latency of memory accesses with independent computation, prefetching improves the rate at which instructions from the application are executed.

Prefetching is challenging for applications with irregular control flow and memory access patterns. For such applications it is difficult to initiate the prefetch access early enough to tolerate the latency of the memory heirarchy while simultaneously accurately predicting the memory address to prefetch. The computation performed between the time the prefetch is initiated and the requested information has been transfered to the cache may affect both the address to be prefetched and whether a cache miss would have occurred.

Recently, several researchers [MPB01, CWT^{+}01, CTWS01, ZS01, Luk01, APD01] have proposed techniques for prefetching data to load instructions with hard to predict address streams using a mechanism called a helper thread. This dissertation proposes and evaluates a modeling framework for predicting, based upon simple profile information, the more complex control flow behavior of a program at the higher level of granularity relevant for speculative threads in general and helper threads in particular. It also proposes and evaluates hardware mechanisms for improving the effectiveness of certain forms of helper threads.

The modeling framework enables the selection of instruction prefetch helper threads by finding good spawn-target pairs. Finding good spawn-target pairs for instruction prefetch is challenging because there is a complex tradeoff between the distance between the spawn-target pair and correlation of the spawn point and the target. The framework was also applied to optimize the selection of simple p-threads for delinquent load prefetch. The framework was found to be significantly more effective at the latter task when employing second-order control flow information due to the correlated nature of branch outcomes. The framework, described in detail in Chapter 2, consists of a simple mapping of control flow onto a Markov chain. This simplified model can represent behavior of interest for modeling the performance impact of speculative threads. It does not represent actual program execution perfectly, but has the advantage that "abstract execution", such as that used to perform data flow analysis, can be applied to predict, with a relatively high degree of accuracy, the performance impact of speculative threads. This abstract execution is performed by employing a path expression framework (as can be done for dataflow analysis [Tar81b, TH92]). The framework was found to accurately model the statistical quantities used for optimizing the selection of spawn-target pairs for prescient instruction prefetch.

The form of helper thread used to reduce instruction cache misses is called a *prescient instruction prefetch* helper thread. This dissertation proposes prescient instruction prefetch helper threads and contributes the first detailed study of helper threads for instruction prefetch. A prescient instruction prefetch helper thread is defined by a spawn-point and a target-point. When the main thread reaches the spawn point a helper thread is launched on an unused hardware thread context, reads data values the helper thread needs, executes a short sequence of code that summarizes the impact of the program's execution between the spawn and target, then begins prefetching instructions for a region after the target called the postfix region. When employing direct pre-execution, this prefetching is performed by executing the main thread instructions directly in the helper thread's hardware context; when employing finite state machine recall, the prefetching is performed by a hardware prefetch engine directed by the helper thread that pre-executes hard to predict branches in the postfix region. Speedups of 4.8% to 17% were found to be feasible as measured using an idealized version of prescient instruction prefetch that does not model the overhead of live-in precomputation.

A hardware mechanism for reducing execution resource consumption, called the *YAT-bit*, is proposed and evaluated. This mechanism mirrors earlier proposals for runahead execution but unlike runahead execution, the focus is for reducing contention for execution resources between helper threads and the main thread. This disseration also provides the first evaluatation of a proposal to provide support for store instructions in helper threads using a mechanism called the *safe-store*, and extends it to improve the effectiveness beyond that of the original safe-store proposal. The YAT-bit is shown to be essential for achieving direct pre-execution on microprocessors that use predication, or conditional move operations. Furthermore, a technique called *finite state machine recall* is proposed and evaluated. Finite state machine recall provides speedups of 10% to 22% (depending upon memory latency), which compares favorably to the most aggressive research hardware instruction prefetch technique.

To illustrate the generality of the modeling framework, it is applied to the optimization of simple p-threads used to prefetch data for delinquent loads. Here we show that it is possible to achieve similar optimization results to those found using trace-based optimization when using higher-order control flow profile information. This is significant as the proposal put forward in this disseration fits within the framework of traditional profile guided optimizing compilers, while the trace-based approach does not. Furthermore, this study sheds new light on the influence of control flow statistics upon data prefetch.

7.2 Contributions

This dissertation makes the following key contributions:

1. It proposes and evaluates a rigorous mathematical framework for modeling a program's control-flow behavior and computing statistical quantities of interest using path expressions.

2. It applies this framework to the optimization of a new form of helper thread used for instruction prefetch and provides the first detailed study of instruction prefetch helper threads.

3. Two forms of prescient instruction prefetch are proposed and evaluated: direct pre-execution, and finite state machine recall.

4. It proposes and evaluates implementation techniques and hardware mechanisms for improving the performance impact of instruction prefetch helper threads.

5. It applies the framework to the optimization of simple p-threads used for prefetching data and underlines the necessity of using second-order control flow statistics for selecting good helper threads for data prefetch.

6. It analyzes sources of modeling error in the prediction of the benefit of specific p-threads (as would be estimated during a compiler based profile guided optimization), and highlights their relative sensitivity on a set of memory intensive benchmarks from the SPEC 2000 integer CPU benchmark [Sta] and the Olden benchmark [Car96].

7.3 Future Work

The work in this dissertation is only a starting point for several areas of research. This section provides some ideas for future work.

7.3.1 Speculative Multithreading and Dependence Chains

The framework can be applied to speculative multithreading, which was described in Chapter 6. Marcuello and Gonzlez [MG02] employ the notion of a high reaching probability to define a set of spawn-target pairs from which a subset of speculative threads are defined using additional heuristics. The use of expected path length and expected path length variance in combination with probabilistic notions of data dependence [CHH+03a] could be used to improve the performance benefits of speculative multithreading. Furthermore, using higher order control flow statistics such as that employed for simple p-thread optimization in Chapter 5 could yield additional benefit and should be explored.

Recently several researchers have explored new microarchitectures that execute separate strands of instructions on separate clusters to localize communication and improve instruction level parallelism [KS02, Sat05]. It may be possible to extend the framework proposed in this dissertation to a hybrid data and control dependence graph (such as a control dependence graph [FOW87]), that can be useful, either to generate code for such microarchitectures, or perhaps even to optimize the microarchitecture for a given suite of applications [J.A96, AM99, ARK99, AR00, Gon00].

7.3.2 Compiler Optimizations

The framework proposed in this dissertation could be useful for traditional optimizations such as instruction scheduling across basic block boundaries, or prefetch instruction insertion.

7.3.3 Optimization of Chaining Helper Threads

Chaining helper threads were described in Chapter 6. Using the statistical modeling framework it may be possible to predict the number of prefetches a chaining helper thread will generate on average, before it runs so far ahead that the costs of additional prefetches (such as evicting data that will be referenced before the data being brough into the cache) outweigh the benefits. Similarly the framework could be used to determine the number of times a spawn instruction should be ignored before triggering the next chaining helper thread sequence from the main thread. Employing such mechanisms with manual tuning has been shown to improve the performance impact of helper threads that prefech multiple instances of a static load once spawned from the main thread [KLW$^+$04].

7.3.4 Equivalent Linked Data Structure Traversals

The observation, made in Section 5.5 of Chapter 5, that helper threads may provide beneficial prefetching even when the main threads control flow follows a different path than implied by dependence slicing, leads to the conjecture that significant opportunity for achieving prefetch slack may exist if the benefit of exploiting *equivalent traversals* of linked data structures (i.e., traversals that start and end at the same node in a linked data structure) can be accurately predicted based upon a combination of *data flow frequency analysis* [Ram96], shape analysis [Rep95], and the control flow modeling framework presented in Chapter 2.

Appendix A

Path Length Variance Operators

The union and closure operators for the path length variance are derived below. The commonality in the approach to their derivation is to express the expected value in terms of the conditional expectation over a set of disjoint paths.

A.1 Union Operator

Let Z represent a random variable whose value is the length of a path selected at random from the set of paths between two specific, disjoint vertices s and t in a positively weighted directed graph. Furthermore assume that the corresponding set of paths can be partitioned into two non-empty sets x and y and let X and Y represent corresponding random variables whose value is the length of the paths between s and t in the sets x and y. Then the expected path length variance of Z is given by:

$$\begin{aligned}
\mathrm{VAR}(Z) &= \mathrm{E}\big[(Z-\mathrm{E}[Z])^2\big] && \text{(A.1)}\\
&= \mathrm{E}\big[Z^2\big]-\mathrm{E}[Z]^2 && \text{(A.2)}\\
&= \mathrm{E}\big[Z^2 |\text{ select path in } X\big]\cdot\Pr\big[\text{select path in } X\big]\\
&\quad + \ \mathrm{E}\big[Z^2 |\text{ select path in } Y\big]\cdot\Pr\big[\text{select path in } Y\big]\\
&\quad - \ \mathrm{E}[Z]^2 && \text{(A.3)}\\
&= \mathrm{E}\big[X^2\big]\cdot\left(\frac{p}{p+q}\right)+\mathrm{E}\big[Y^2\big]\cdot\left(\frac{q}{p+q}\right)-\left(\frac{\mathrm{E}[X]\cdot p+\mathrm{E}[Y]\cdot q}{p+q}\right)^2 && \text{(A.4)}\\
\mathrm{VAR}(Z) &= \frac{p\cdot(v_X+m_X^2)+q\cdot(v_Y+m_Y^2)}{p+q}-\left(\frac{p\cdot m_X+q\cdot m_Y}{p+q}\right)^2 && \text{(A.5)}
\end{aligned}$$

Where p and q represent the probabilities of starting at s and following a path in, respectively, X or Y to reach t; m_X and m_Y represent the mean path length starting from s and going to t through a path in respectively X or Y; v_X and v_Y represent the path length variance starting from s and going to t through a path in respectively X or Y. The individual steps of the derivation are elaborated below:

A.1 to A.2 $\mathrm{E}[Z]$ is a constant. The expected value of a constant multiplied by a random variable is the expected value of the random variable multipled by the constant—i.e., $\mathrm{E}\big[Z \cdot \mathrm{E}[Z]\big] = \mathrm{E}[Z] \cdot \mathrm{E}[Z] = \mathrm{E}[Z]^2$.

A.2 to A.3 It is well known that $\mathrm{E}[A] = \mathrm{E}\big[\mathrm{E}[A|B]\big]$ – see for instance Leon-Garcia [LG94, eqn 4.37 on pp. 215]. To transform Equation A.2 into Equation A.3 we consider $A := Z^2$, and define B to be the bernoulli random variable with outcome 'b_X' representing the event that a path in X was selected, and outcome 'b_Y' representing the event that a path in Y was selected. Since B is a discrete random variable we have $\mathrm{E}\big[\mathrm{E}[A|B]\big] = \sum_{b_i} \mathrm{E}[A|b_i] \cdot \Pr(b_i)$ [LG94, pp. 216].

A.3 to A.4 To evaluate $\Pr(b_i)$ we note that $\Pr(b_X) = \Pr(X|Z)$ because we are only interested in the event that a path from s to t is selected from X if it also lies in Z. Then we have $\Pr(X|Z) = \dfrac{\Pr(X \cap Z)}{\Pr(Z)} = \dfrac{\Pr(X)}{\Pr(Z)} = \dfrac{\Pr(X)}{\Pr(X) + \Pr(Y)} = \dfrac{p}{p+q}$. A similar result holds for $\Pr(b_Y)$.

A.4 to A.5 Rearrange and substitute for $\mathrm{E}[X]$, $\mathrm{E}[Y]$, $\mathrm{E}[X^2]$, and $\mathrm{E}[Y^2]$.

A.2 Closure Operator

Let Z represent a random variable whose value is the length of a path selected at random from the set of paths starting and ending at vertex x in a positively weighted directed graph. Then the expected path length variance of Z is given by:

$$\begin{aligned}
\mathrm{VAR}(Z) &= \mathrm{E}\Big[\big(Z - \mathrm{E}[Z]\big)^2\Big] && \text{(A.6)}\\
&= \sum_{n=0}^{\infty} \mathrm{E}\Big[\big(Z_N - \mathrm{E}[Z]\big)^2 | N = n\Big] \cdot \Pr[N = n] && \text{(A.7)}\\
&= \sum_{n=0}^{\infty} \mathrm{E}\Big[Z_N^2 - 2Z_N E[Z] + E[Z]^2 | N = n\Big] \cdot \Pr[N = n] && \text{(A.8)}\\
&= \sum_{n=0}^{\infty} \Big(\mathrm{E}[Z_N^2 | N = n] - 2E[Z_N | N = n]E[Z] + E[Z]^2\Big) \cdot \Pr[N = n] && \text{(A.9)}\\
&= \sum_{n=0}^{\infty} \mathrm{E}\Big[\sum_{i=0}^{n} X_i \sum_{j=0}^{n} X_j\Big] - 2nE[X]E[Z] + E[Z]^2\Big) \cdot \Pr[N = n] && \text{(A.10)}\\
&= \sum_{n=0}^{\infty} \Big(nE[X^2] + n(n+1)E[X]^2 - 2nE[X]E[Z] + E[Z]^2\Big) \cdot \Pr[N = n] && \text{(A.11)}\\
&= \sum_{n=0}^{\infty} \Big(n\sigma + n(n-1)\mu^2 - \frac{2n\mu^2 p}{1-p} + \big(\frac{\mu p}{1-p}\big)^2\Big) \cdot \big(1-p\big)p^n && \text{(A.12)}\\
&= \sigma(1-p)\sum_{n=0}^{\infty} np^n + (1-p)\mu^2 \sum_{n=0}^{\infty} n^2 p^n - (1-p)\mu^2 \sum_{n=0}^{\infty} np^n \\
&\quad -2\mu^2 p \sum_{n=0}^{\infty} np^n + \frac{\mu^2 p^2}{1-p} \sum_{n=0}^{\infty} p^n && \text{(A.13)}\\
&= \sigma(1-p)\frac{p}{(1-p)^2} + \mu^2(1-p)p\frac{p+1}{(1-p)^3} - \mu^2(1-p)\frac{p}{(1-p)^2} \\
&\quad -2\mu^2 p \frac{p}{(1-p)^2} + \frac{\mu^2 p^2}{1-p}\frac{1}{1-p} && \text{(A.14)}\\
&= \frac{\sigma p}{1-p} + \mu^2 p \frac{p+1}{(1-p)^2} - \mu^2 \frac{p}{1-p} \\
&\quad -2\mu^2 \frac{p^2}{(1-p)^2} + \mu^2 \frac{p^2}{(1-p)^2} && \text{(A.15)}\\
&= \frac{\sigma p}{1-p} + \frac{\mu^2 p}{(1-p)^2} - \frac{\mu^2 p}{1-p} && \text{(A.16)}\\
&= \frac{\sigma p}{1-p} + \frac{\mu^2 p - \mu^2 p(1-p)}{(1-p)^2} && \text{(A.17)}\\
&= \frac{(v + m^2)p}{1-p} + \left(\frac{pm}{1-p}\right)^2 && \text{(A.18)}
\end{aligned}$$

Where p is the probability of repeating another iteration after entering the loop, m is the expected path length of a single iteration of the loop body, and v is the path length variance of a single iteration of the loop body. The individual steps of the derivation are elaborated below:

A.6 to A.7	Apply the expansion $\mathrm{E}[A] = \mathrm{E}\big[\mathrm{E}[A\|B]\big]$, with $A := \big(Z - \mathrm{E}[Z]\big)^2$ conditioned on $B = N$, the number of iterations around the loop. The notation Z_N is used to indicate that the value of Z is conditioned on the number of iterations.
A.7 to A.8	Expand the squared expression.
A.8 to A.9	Apply $\mathrm{E}[c \cdot X] = c \cdot \mathrm{E}[X]$, where c is a constant.
A.9 to A.10	Let $Z_N = \sum_{i=0}^{N} X_i$ where X_i is the path length of the i-th iteration of the loop.
A.10 to A.11	Expand the summations and reorganize assuming the X_i are independent random variables so that $\mathrm{E}[X_i \cdot X_j] = \mathrm{E}[X_i]\mathrm{E}[X_j]$ for $i \neq j$ (recall the covariance of two independent random variables X and Y is zero, and that $\mathrm{COV}(X,Y) := \mathrm{E}[XY] - \mathrm{E}[X]\mathrm{E}[Y]$; see [LG94, pp.233-234]).
A.11 to A.12	Substitute σ for $\mathrm{E}[X^2]$ and μ for $\mathrm{E}[X]$. Note that $\mathrm{E}[Z] = \dfrac{p\mu}{1-p}$ as shown in Section 2.2.2.
A.12 to A.13	Expand the product of terms and move constants outside of the summation.
A.13 to A.14	Apply the following simplifications: $\sum_{n=0}^{\infty} p^n = \dfrac{1}{(1-p)}$; $\sum_{n=0}^{\infty} np^n = \dfrac{p}{(1-p)^2}$; $\sum_{n=0}^{\infty} n^2 p^n = \dfrac{p(p+1)}{(1-p)^3}$. The first is the well known geometric series summation. The others can be derived by differentiation.
A.14 to A.15	Cancel identical factors from numerator and denominator.
A.15 to A.16	Expand and cancel terms.
A.16 to A.17	Cross multiply and combine the second and third terms.
A.17 to A.18	Simplify the second term and replace $\mu := \mathrm{E}[X]$ by letting $m := \mathrm{E}[X]$, and replace for $\sigma := \mathrm{E}[X^2]$ by using the relation $\mathrm{VAR}(X) := \mathrm{E}[X^2] - \mathrm{E}[X]^2$ and letting $v := \mathrm{VAR}(X)$.

Bibliography

[ABC+88] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, Oct. 1988.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, 1990.

[ACH+04] Tor M. Aamodt, Paul Chow, Per Hammarlund, Hong Wang, and John P. Shen. Hardware Support for Prescient Instruction Prefetch. In *10th International Symposium on High-Performance Computer Architecture*, 2004.

[AD98] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 226–236. IEEE Computer Society Press, 1998.

[AH90] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246–256, 1990.

[AKPW83] J.R. Allen, K. Kennedy, C. Portferfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[AM99] S.G. Abraham and S.A. Mahlke. Automatic and Efficient Evaluation of Memory Hierarchies for Embedded Systems. In *Proc. 32nd Ann. Int'l Symp. Microarchitecture (MICRO-32)*, pages 114–125, 1999.

[AMC01] Tor Aamodt, Andreas Moshovos, and Paul Chow. The Predictability of Computations that Produce Unpredictable Outcomes. In *5th Workshop on Multithreaded Execution, Architecture, and Compilation*, pages 23–34, 2001.

[AMC$^+$02] Tor Aamodt, Pedro Marcuello, Paul Chow, Per Hammarlund, and Hong Wang. Prescient Instruction Prefetch. In *6th Workshop on Multithreaded Execution, Architecture, and Compilation*, pages 3–10, 2002.

[AMC$^+$03] Tor M. Aamodt, Pedro Marcuello, Paul Chow, Antonio González, Per Hammarlund, Hong Wang, and John P. Shen. A Framework for Modeling and Optimization of Prescient Instruction Prefetch. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 13–24, 2003.

[APD01] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *28th International Symposium on Computer Architecture*, pages 52–61, 2001.

[AR00] S.G. Abraham and B.R. Rau. Efficient Design Space Exploration in PICO. In *Proc. Int'l Conf. Compilers, Architecture Synthesis for Embedded Systems (CASES 2000)*, pages 71–79, 2000.

[ARK99] S. Aditya, B.R. Rau, and V. Kathail. Automatic Architectural Synthesis of VLIW and EPIC Processors. In *IEEE/ACM Int'l Symp. System Synthesis*, pages 107–113, 1999.

[AWSH] Tor M. Aamodt, Hong Wang, John Shen, and Per Hammarlund. Methods and apparatus for generating speculative helper thread spawn-target points. United States Patent Application 20040154019, Filed April 24, 2003 published August 5, 2004.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. http://www.simplescalar.com, 1997.

[BC91] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 176–186, New York, NY, USA, 1991. ACM Press.

[BCC$^+$00] Jay Bharadwaj, William Y. Chen, Weihaw Chuang, Gerolf Hoflehner, Kishore N. Menezes, Kalyan Muthukumar, and Jim Pierce. The intel ia-64 compiler code generator. *IEEE Micro*, 20(5):44–53, 2000.

[BL96] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[Car96] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, Department of Computer Science, June 1996.

[CFA04] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 76–87, June 2004.

[CHH$^+$03a] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jeng Kuen Lee. Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-To Analysis. In *9th Symposium on Principles and Practice of Parallel Programming*, 2003.

[CHH$^+$03b] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-To Analysis. In *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, 2003.

[CHJL04] Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Interprocedural probabilistic pointer analysis. *IEEE Trans. Parallel Distrib. Syst.*, 15(10):893–907, 2004.

[CMH91] Pohua P. Chang, Scott A. Mahlke, and W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software – Practice and Experience*, 21(12):1301–1321, 1991.

[CSK$^+$99] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *26th International Symposium on Computer Architecture*, pages 186–195, 1999.

[CTWS01] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic Speculative Precomputation. In *34th International Symposium on Microarchitecture*, pages 306–317, 2001.

[CTYP02] Robert S. Chappell, Francis Tseng, Adi Yoaz, and Yale N. Patt. Difficult-Path Branch Prediction Using Subordinate Microthreads. In *29th International Symposium on Computer Architecture*, pages 307–317, 2002.

[CWT$^+$01] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *28th International Symposium on Computer Architecture*, pages 14–25, 2001.

[DM97] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. 1997 ACM Int. Conf. on Supercomputing*, pages 68–75, July 1997.

[DS98] M. Dubois and Y. Song. Assisted execution. Technical Report CENG 98-25, Department of EE-Systems, University of Southern California, October 1998.

[Dun98] James Dundas. *Improving processor performance by dynamically pre-processing the instruction stream.* PhD thesis, University of Michigan, 1998.

[Eme99] Joel Emer. Simultaneous multithreading: Multiplying alpha's performance. Microprocessor Forum, October 1999.

[FCJV97] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-System Design Considerations for Dynamically-Scheduled Processors. In *24th Annual International Symposium on Computer Architecture*, pages 133–143, 1997.

[FJC95] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *HPCA*, pages 78–89, 1995.

[FMS88] J. Ferrante, M. Mace, and B. Simons. Generating sequential code from parallel code. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 582–592, 1988.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[FS96] Manoj Franklin and Gurindar S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.*, 45(5):552–571, 1996.

[Gle98] Andrew Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Ideas Session*, 1998.

[Gon00] R. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, 2000.

[GVSS98] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. Sohi. Speculative Versioning Cache. In *4th International Symposium on High-Performance Computer Architecture*, 1998.

[Gwe93] Linley Gwennap. Speed Kills? Not for RISC Processors. *Microprocessor Report*, 7(3):3, 1993.

[HAA+96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, December 1996.

[Hal99] Tom Halfhill. Sun Makes MAJC With Mirrors. *Embedded Processor Watch*, 73, November 1999.

[HD77] D. W. Hammerstrom and E. S. Davidson. Information Content of CPU Memory Referencing Behavior. In *4th International Symposium on Computer Architecture*, pages 184–192, 1977.

[HMR+00] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, 20(5):12–23, 2000.

[HS01] G. Hinton and J. Shen. Intel's multi-threading technology. Microprocessor Forum, October 2001.

[Inc04] GrammaTech Inc. Codesurfer version 1.9 patchlevel 3. http://www.grammatech.com, 2004.

[Inc05] Apple Computer Inc. ipod. http://www.apple.com/ipod/, 2005.

[Int] Intel Corporation. Special Issue on Intel Hyper-Threading Technology in Pentium 4 Processors. Intel Technology Journal. Q1 2002.

[J.A96] J.A. Fisher and P. Faraboschi and G. Desoli. Custom-Fit Processors: Letting Applications Define Architectures. In *Proc. 29th Int'l Symp. Microarchitecture (MICRO-29)*, pages 324–335, 1996.

[JCO98] D.-C. R Ju, J.-F. Collard, and K. Oukbir. Probabilistic Memory Disambiguation and its Application to Data Speculation. In *3rd Workshop on Interaction between Compilers and Computer Architectures (INTERACT-3)*, Oct 1998.

[Jou89] Norman P. Jouppi. The Nonuniform Distribution of Instruction-Level and Machine-Level Parallelism and Its Effects on Performance. *IEEE Transactions on Computers*, 38(12):1645–1658, 1989.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM Press.

[JS00] Quinn Jacobson and James E. Smith. Trace preconstruction. In *ISCA-27*, pages 37–46, 2000.

[KLW$^+$04] Dongkeun Kim, Shih-Wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John Paul Shen. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *2nd Intl. Symposium on Code Generation and Optimization (CGO 2004)*, pages 27–38, 2004.

[KS02] H. Kim and J. Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *Proc. 29th International Symposium on Computer Architecture (ISCA)*, pages 71–81, 2002.

[KS04] Tejas S. Karkhanis and James E. Smith. A First-Order Superscalar Processor Model. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 338–349, 2004.

[KT98] Venkata Krishnan and Josep Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proceedings of the 12th international conference on Supercomputing*, pages 85–92. ACM Press, 1998.

[KY02] Dongkeun Kim and Donald Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *ASPLOS-X*, pages 159–170, 2002.

[LG94] Alberto Leon-Garcia. *Probability and Random Processes for Electrical Engineering - Second Edition*. Addison-Wesley, 1994.

[LM98] Chi-Keung Luk and Todd C. Mowry. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 182–194, 1998.

[LS84] J. K. F. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, pages 6–22, January 1984.

[LS96] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–237, December 1996.

[Luk00] Chi-Keung Luk. *Optimizing the Cache Performance of Non-numeric Applications*. PhD thesis, Department of Computer Science, University of Toronto, January 2000.

[Luk01] Chi-Keung Luk. Tolerating Memory Latency Through Software-Controlled Pre-execution in Simultaneous Multithreading Processors. In *28th International Symposium on Computer Architecture*, pages 40–51, 2001.

[LWW$^+$02] Steve S.W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoflehner, Daniel Lavery, and John P. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 117–128, 2002.

[MBVS97] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 181–193, New York, NY, USA, 1997. ACM Press.

[MG02] Pedro Marcuello and Antonio Gonzlez. Thread-Spawning Schemes for Speculative Multithreading. In *8th International Symposium on High-Performance Computer Architecture*, pages 55–64, 2002.

[MGT98] Pedro Marcuello, Antonio Gonz&#225;lez, and Jordi Tubella. Speculative multithreaded processors. In *Proceedings of the 12th international conference on Supercomputing*, pages 77–84. ACM Press, 1998.

[MH86] S. McFarling and J. Hennessy. Reducing the Cost of Branches. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 396–403, 1986.

[ML97] Todd C. Mowry and Chi-Keung Luk. Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 314–320, 1997.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 62–73, 1992.

[Moo65] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MPB01] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. Slice-Processors: An Implementation of Operation-Based Prediction. In *15th International Conference on Supercomputing*, pages 321–334, 2001.

[MSWP03] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *9th International Symposium on High-Performance Computer Architecture*, 2003.

[NS94] Derek B. Noonburg and John P. Shen. Theoretical Modeling of Superscalar Processor Performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 52–62, 1994.

[OH00] David Ofelt and John L. Hennessy. Efficient Performance Prediction For Modern Microprocessors. In *SIGMETRICS*, pages 229–239, 2000.

[Pap96] David B. Papworth. Tuning the Pentium Pro Microarchitecture. *IEEE Micro*, 16(2), April 1996.

[PK94] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 24–33, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Por89] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Department of Computer Science, Rice University, May 1989.

[PS91] J.C.H. Park and M.S. Schlansker. On predicated execution. Technical Report HPL-91-58, HP Laboratories, Palo Alto, CA, May 1991.

[Ram96] G. Ramalingam. Data flow frequency analysis. In *SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 267–277, 1996.

[RBG$^+$01] Alex Ramirez, Luiz Andr Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code Layout Optimizations for Transaction Processing Workloads. In *ISCA-28*, pages 155–164, 2001.

[RCA99] Glenn Reinman, Brad Calder, and Todd Austin. Fetch Directed Instruction Prefetching. In *32nd International Symposium on Microarchitecture*, pages 16–27, 1999.

[Rep95] Thomas Reps. Shape Analysis as a Generalized Path Problem. In *PEPM '95: Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, 1995.

[RR03] Steven Raasch and Steven Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *The 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[RS01] Amir Roth and Gurindar S. Sohi. Speculative Data-Driven Multithreading. In *7th International Symposium on High-Performance Computer Architecture*, pages 37–48, 2001.

[RS02] Amir Roth and Gurindar S. Sohi. A Quantitative Framework for Automated Pre-Execution Thread Selection. In *35th International Symposium on Microarchitecture*, pages 430–441, 2002.

[SAMC98] K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark. Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. In *31st International Symposium on Microarchitecture*, pages 259–71, 1998.

[Sar89] V. Sarkar. Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 298–312, 1989.

[Sat05] Satish Narayanasamy and Hong Wang and Perry H. Wang and John Paul Shen and Brad Calder. A Dependency Chain Clustered Microarchitecture. In *Proc. 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, 2005.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *ISCA-22*, pages 414–425, 1995.

[SCZM00] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Comput. Archit. News*, 28(2):1–12, 2000.

[SFKS02] Andre Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 295–306, 2002.

[SH92] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[SL98a] Srikanth T. Srinivasan and Alvin R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *International Symposium on Microarchitecture*, pages 148–159, 1998.

[SL98b] Srikanth T. Srinivasan and Alvin R. Lebeck. Load Latency Tolerance In Dynamically Scheduled Processors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 148–159. IEEE Computer Society Press, 1998.

[SM98] 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, February 1998.

[Smi78] A. Smith. Sequential program prefetching in memory bierarchies. *IEEE Computer*, 11(2):7–21, 1978.

[Smi81] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 136–148, 1981.

[Smi82] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[SPHC02a] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[SPHC02b] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[SPR00] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *ASPLOS-IX*, pages 257–268, 2000.

[Sta] Standard Performance Evaluation Corporation. SPEC 2000 CPU benchmarks. http://www.specbench.org/.

[Ste93] B. Steensgaard. Sequentializing Program Dependence Graphs for Irreducible Programs. Technical Report MSR-TR-93-14, Microsoft Research, October 1993.

[Tar81a] Robert Endre Tarjan. A Unified Approach to Path Problems. *Journal of the ACM*, 28(3):577–593, 1981.

[Tar81b] Robert Endre Tarjan. Fast Algorithms for Solving Path Problems. *Journal of the ACM*, 28(3):594–614, 1981.

[TCC$^+$00] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The majc architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[TEE$^+$96] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23rd International Symposium on Computer Architecture*, pages 191–202, 1996.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd International Symposium on Computer Architecture*, pages 392–403, 1995.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit - A Tool for Building Optimizers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 82–93, 1992.

[Tow76] R. A. Towle. *Control and Data Dependence for Program Transformations.* PhD thesis, University of Illinois, Urbana-Champaign, 1976.

[Tul96] D.M. Tullsen. Simulation and Modeling of a Simultaneous Multithreading Processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, December 1996.

[VS98] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 81–92, 1998.

[WCW$^+$04] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, and John P. Shen. Helper Threads via Virtual Multithreading on an Experimental Itanium 2 Processor-based Platform. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 144–155, 2004.

[Wei81] Mark Weiser. Program slicing. In *5th International Conference on Software Engineering*, pages 439–449, 1981.

[WWC$^+$02] Perry H. Wang, Hong Wang, Jamison D. Collins, Ed Grochowski, Ralph-Michael Kling, and John Paul Shen. Memory Latency-Tolerance Approaches for Itanium Processors: Out-of-Order Execution vs. Speculative Precomputation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 187–196, 2002.

[WWFH03] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *30th International Symposium on Computer Archictecture*, pages 84–95, 2003.

[WWK$^+$01] Perry H. Wang, Hong Wang, Ralph-Michael Kling, Kalpana Ramakrishnan, and John Paul Shen. Register Renaming and Scheduling for Dynamic Execution of Predicated Code. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 15–26, 2001.

[YP91] T.-Y. Yeh and Y. N. Patt. Two-Level Adaptive Branch Prediction. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 51–61, 1991.

[ZCSM02] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–183, October 2002.

[ZCSM04] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004)*, pages 39–52, March 2004.

[ZS00] Craig B. Zilles and Gurindar S. Sohi. Understanding the backward slices of performance degrading instructions. In *27th International Symposium on Computer Architecture*, pages 172–181, 2000.

[ZS01] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *28th International Symposium on Computer Architecture*, pages 2–13, 2001.

[ZS02] Craig Zilles and Gurindar Sohi. Master/Slave Speculative Parallelization. In *35th International Symposium on Microarchitecture*, pages 85–96, 2002.