PROGRAM DEMULTIPLEXING: DATA-FLOW BASED SPECULATIVE PARALLELIZATION OF METHODS IN SEQUENTIAL PROGRAMS

by

Saisanthosh Balakrishnan

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2007

Abstract

Processor performance has steadily increased in the past several decades. The continuation of this, along with drop in the price to performance, is critical to accelerating the transformations that will result from applications that demand powerful computing. Unfortunately, the processor industry has faced problems with the incumbent processing model —out-of-order superscalar processor— with increasing design complexity and power consumption combined with diminishing performance gains. Processor manufacturers, en masse, have sidestepped this predicament by moving to multicore systems; the current generation of multicore systems being multiple processing cores in a chip. In the future, applications that demand higher and/or continuously increasing performance must rely on parallel execution in some form to use the processing cores in a multicore system. Multi-threaded applications such as enterprise application servers and scientific programs, can benefit from multicore systems as they are either easily parallelizable due to the nature of the application, or have been written by expert programmers who have carefully orchestrated threads. However, such applications occupy only a small fraction of the end-user market. Parallelizing the large fraction of single-threaded programs would require many experienced multi-threaded programmers, as well as significant time requirements to both develop such applications and debug them. Novel approaches are required for parallel execution of programs for the upcoming generations of multicore architectures.

Speculative parallelization is a class of techniques that achieves parallel execution from a sequential program. The basic idea behind proposals in this class is to compose regions of program code as threads, and speculatively execute these “threads” in parallel. With extra hardware support, violations of dependencies from the sequential program order between the threads are determined, and the threads’ execution serialized. This dissertation proposes Program Demultiplexing or PD, a speculative parallelization model that has two novel contributions.

In PD, a speculative thread is composed of a method in a program. A method (also known as procedure, function or subroutine) is a fundamental programming construct used to express a desired sub-computation in a program. Methods are a good choice for a parallel execution model such as PD because they are specified by the programmers and, therefore allow them to reason about performance and correctness. While there is no mandatory programming rule that a program should be composed of many methods, and each

one should solve a specific sub-task, modern applications are commonly developed by many developers who program their tasks as several methods for easier debugging and reusability, and use methods from one or more libraries and language packages. It is of no doubt that most applications are developed with a considerable level of reasoning that determines how the problem must be solved. Methods are an integral part in this development process.

With this choice, PD is based on the observation that a sequential program is a collection of different methods called by the programmer, one after another, for convenience in expressing the computation, as well as to satisfy the default assumption of execution on a single processing core. Parallelism, even if it may exist between different methods, remains unexploited. In PD, different methods are "demultiplexed" from the sequential program order. The execution of a method, which is performed when it is called in a sequential program, is separated, and speculatively executed at a point earlier, on a different processing core. The execution model, therefore, may be speculatively executing several methods in a program, in parallel.

The second aspect of PD is a refinement to the execution model used by previous speculative parallelization proposals, which was to speculatively traverse the control-flow graph of a program at the granularity of speculative threads, and fork them for speculative execution. This control-flow based speculative parallelization approach has a shortcoming. Data requirements for a speculative thread are not considered and, therefore, the execution model may be prone to data violations. This could result in mis-speculations and discarding of not only the thread that violated the dependency but, because of the speculative control flow traversal, the squashing of all threads that follow. PD analyzes data requirements of a speculative thread and determines the most suitable point the thread can begin execution, which is usually much earlier than the call site of the method that the speculative thread is associated with. It uses the call site in the program only for committing the speculative thread and to preserve the sequential program order.

PD achieves its style of execution with two software generated components. A *trigger* specifies the point in the program when a speculative thread may begin execution. It may be placed directly in the program, or can be provided as conditions (predicates) based on program counters. In the latter case, the conditions are evaluated dynamically by the hardware which, when satisfied, begins the speculative thread. A *handler* provides the explicit live-ins of a speculative thread, which are the parameters of the method that it calls. In

addition, the handler also evaluates branches that the call site may be control dependent on, to ensure that the speculative execution of the method is not wasted.

Proposed hardware support for PD consists of speculative execution of the threads, the foremost requirement for all speculative parallelization proposals. Private caches are used to ensure that stores performed by the speculative thread are tracked. To hold the results of a speculative thread, until it can be used by the program or another speculative thread, and to squash a speculative thread that violated any dependencies, private caches may be used. This dissertation instead, proposes novel and efficient storage structures collectively referred to as the execution buffer pool, to alleviate the contention that speculative threads may have if they use the private cache(s) of a processing core. Finally, to support evaluation of triggers, trigger condition code registers are provided to store the results of predicates of triggers, which are operated on by micro-code that evaluates the conditions specified in the triggers.

A simulation-based implementation of PD is evaluated on integer benchmarks from the SPECCPU2000 suite, programs written in C with no explicit concurrency and/or motivation to create concurrency. Several results of the implementation are examined, notably the methods chosen for PD, their size, read and write sets during speculative execution and the overheads incurred during speculative execution, the utilization of processing cores during speculation, the sizing of proposed hardware structures, performance benefits, and limitations of program ordered forking model in prior speculative parallelization proposals. PD achieves harmonic mean speedup of 1.5x on benchmarks evaluated. The execution model has significant potential to achieve greater performance improvements and scalability on a wide variety of applications.

Acknowledgments

I thank my advisor Guri Sohi, for his excellent mentoring. Guri has always been there to listen and give advice. He has been a great influence and a solid role model. I have learnt a lot from his strong ethics, shrewd thinking, and use of precise language in writing and communication. He has guided me in choosing challenging problems, while at the same time giving me freedom to pursue and tackle them. He has tremendously helped me in organizing my research and expressing it in writing. Guri's unique style of leading students has had great impact on both my technical thinking, and in developing my vision and confidence in pursuing it. I thank him for taking me as his student, and funding me for these many years. Without his help and encouragement, this work would not have been possible.

I thank Professor Charlie Chen, for without his financial support for my first semester of graduate school, I would not have come to Madison. Thanks to Professors Charles Fischer, Mark Hill, Karthikeyan Sankaralingam, James Smith, and David Wood for serving on my preliminary and defense exam committee. Their feedback has been useful in improving this dissertation. I also thank David for patiently proofreading many chapters of this dissertation.

Thanks to former and current graduate students in Guri's research group, Matthew Allen, Adam Butts, Koushik Chakraborty, Jichuan Chang, Vikas Garg, Allison Holloway, Paramjit Oberoi, Philip Wells, and Craig Zilles for patiently reading many of my drafts, listening to my talks, and collobarating with me on building the simulation infrastructure. Jichuan, Tejas Karkhanis, Matthew, Vikas, and Koushik acted as a good computer architecture listening board for me. They helped me turn my rambling into formal arguments and meaningful ideas. I have also immensely benefited from the computer architecture community at Wisconsin, especially by observing the arguments and analyses presented by students and professors during seminars, preliminary, and defense exams.

I am greatly indebted to my excellent mentors, Tin-fook Ngai, Ravi Rajwar, Konrad Lai, Mike Upton, and Haitham Akkary, during my internships at Intel. The internships were important steps in my learning process; thanks to fellow interns and members of the Intel Programming Systems and Microarchitecture Labs for providing a stimulating environment.

I would be remiss without acknowledging my friends. I am fortunate to have many friends from my

alma maters, in Madison, and have met many more in the past seven years, who have all made my graduate school life enjoyable. Many others, outside of Madison, hosted me and served as an effective respite from work. You all know who you are; thanks!

Above all others, I thank my parents, Balakrishnan and Hema, for instilling values and educating me on many topics beyond school. I am blessed for the love, affection, constant support and encouragement that I have from them and my brother, Saikrishna. Words cannot express the thanks I have for them.

Table of Contents

List of Figures

List of Tables

Code Listings

CHAPTER 1

INTRODUCTION

In von Neumann architectures, a sequential program, more commonly referred to as a program, is defined as a stored set of instructions. The execution state of these instructions, specified by the register and memory values, is defined by an instruction pointer or program counter. The simplest means of executing a program is to execute the instruction pointed to by the instruction pointer, which will result in the access and modification of register and/or memory values, including the instruction pointer. After execution, the instruction pointer points to the next instruction to be executed. This process is repeated until the end of program is reached. Performance achieved by the machine is defined by the execution time or run time of the program.

A major accomplishment of the microprocessor industry is the improvement in performance with every generation of processors. For example, Intel Corporation now markets processors that are 5,000 times faster than their first microprocessor in 1971 [147]. Advancements in process and material technology, innovations in the architectural, micro-architectural, and circuit implementations, and efficient design tools have helped achieve this significant gain.

A topic that is of interest, particularly for innovations to improve performance, is parallelism. Parallelism in programs is a fundamental characteristic of the program that denotes the independence of computations in a program. Parallelism provides the ability to perform several computations (instructions) in a program concurrently because of their independence. Exploiting it can help reduce execution time and improve performance of a program. The desired extent of parallelism or concurrency to be exploited by the system will determine the support needed from the hardware and software subsystems, and will determine the improvement in performance that can be achieved. It is necessary to use different means to exploit parallelism at several granularities of instructions to achieve improvements in performance. I describe some of them next.

ILP. Instruction-level parallel (ILP) [146, 176, 179] execution is one of the most popular hardware based parallel execution models to date; almost every processor is built to exploit ILP. A uniprocessor exploiting instruction level parallelism employs techniques to extract parallelism from few hundreds of instructions. By using a variant of Tomasulo's algorithm [192] implemented in hardware (also called dynamic scheduled out-of-order processing), false dependencies between instructions are eliminated and instructions executed from the reorder buffer according to their data dependencies, hence achieving an "out-of-order" execution model. The changes to the register and memory values made by an instruction are held until all prior instructions in the reorder buffer have committed their changes. This ensures that the software system sees the changes of an instruction in the order of the sequential program. Another instruction-level parallel processing model is the Very Long Instruction Word or VLIW processors [34, 51, 52, 66, 158, 159, 172] that uses software to schedule independent instructions in the same cycle by means of an explicit instruction set architecture. Examples of out-of-order processors include Alpha 21264 [100], Intel Pentium Pro [50], Intel Pentium 4 [89], IBM Power4 [190], and VLIW processors such as the Intel Itanium [132, 173]. The development and use of instruction-level parallel execution techniques has been supported by a significant body of research and by commercial microprocessors.

Distributed ILP. Traditional ILP uniprocessors have centralized hardware resources such as the reorder buffer, load and store queues, that limit the scalability of the micro-architecture. In addition, the complexity of such architectures can result in significant power consumption, heat dissipation, and complex and time consuming validation. To tackle these shortcomings, yet design a scalable instruction-level parallel processing system, there have been several recent academic proposals for decentralization of the cycle critical micro-architectural structures that are power efficient. Examples are ILDP [102], TRIPS [169], and RAW [189] processors. All of these proposals consider new instruction sets with low-latency scalar operand networks to communicate values between distributed processing units. The distributed processing units or tiles usually consist of a local storage (register file), execution units, and a programmable switch. Some of these proposals (RAW and TRIPS) require compile time support to assign instructions to processing units and/or support to orchestrate the communication of values between the units. The WaveScalar processor [187] is considered a data-flow based distributed micro-architecture as it divides a program into waves, and executes these waves on processing units that fire dynamically according to data dependencies. Several of these proposals

not only target the instruction-level parallelism targeted by the out-of-order superscalar processor, but also parallelism beyond that, at a coarser granularity.

Traditional Non-Speculative Parallelization. Due to diminishing efficiency in increasing the extent of parallelism that can be exploited by hardware, software techniques are used to exploit parallelism at a very coarse level, say, of the order of several hundred thousand instructions. The multi-threaded programming model has long served this purpose. A multi-threaded program is composed of several threads and can be executed concurrently on several processing cores. Each thread is in itself a sequential program and therefore, with corresponding program counters or instruction pointers. There has been a significant body of work in expressing parallel execution as a collection of multiple sequential programs [4, 5, 13, 15–18, 28, 29, 39, 53, 71–73, 79, 80, 85, 99, 109, 113–115, 127, 130, 141, 149, 156, 160, 195, 197, 206]. This is achieved by writing a parallel program for a problem, either by having the programmer express the parallelism explicitly in the program by managing multiple threads, or by extracting parallelism automatically from a sequential program, thus creating a multi threaded application. I broadly refer to this form of non-speculative parallelization as control-driven parallelization.

There are several software parallelization models that fall under the umbrella of control-driven parallelization, each suitable for a specific set of applications. The decomposition of a problem into multi-threaded application will depend on the form and type of parallelism present in it [186]. In the class of embarrassingly parallel applications, concurrency is easily achieved as threads will have no shared state between them, or frequently access shared data with minimal modifications. For example, server-side programs such as webservers can have many concurrently running threads each servicing one or more web requests. Similarly, parallel compilation application such as pmake can compile several C/C++ program source files concurrently. In these cases, parallelism is easily expressed by the programmer using threads.

Another large class of applications is programs that exhibit structured parallelism such as scientific, multimedia, signal processing, and bioinformatics programs. For example, scientific programs usually have instructions that reference an array, perform some stencil computation on the data values referenced, and store the computed value(s) back in the array. Parallelism can be automatically extracted and represented as a collection of control-driven threads such as distributing chunks of loop iterations in a scientific program between many processors. Libraries such as OpenMP can be used to transform a sequential program into

a parallel program by means of pragmas placed before chosen regions of code to be parallelized. These pragmas are processed by the compiler's pre-processor, which automatically inserts calls to the library for dividing the loop iterations across many processors. At the end of a parallel region, a barrier placed by the compiler ensures that all processing cores have finished their assigned task before proceeding with the rest of the program. Other data parallel means (also known as, single instruction multiple data stream or SIMD [67]) such as the MMX [148], 3DNow! [137], SSE [21], AltiVec [60], and Tarantula [63] instruction set extensions on processors, can be used for expressing structured parallelism.

Many applications however, exhibit unstructured parallelism that is not straightforward to identify and cannot be easily expressed by the programmer. Moreover, many applications use irregular data structures and have data access patterns that are not easy to analyze and are not amenable to automatic parallelization. Features of modern languages and compiler infrastructures further hinder the analysis and parallelization of the entire program. These include, separate compilation of files, dynamic linking of libraries, dependence on the runtime system to perform several tasks for a managed application, usage of language packages, object-oriented practices such as information hiding, multiple inheritance, and so on. Therefore, parallel algorithms to problems often have to be carefully constructed by expert programmers, and then programmed in the desired language with the use of synchronization primitives such as locks to protect any data shared and modified between threads. Expectedly, this is a hard task, as the low-level concurrency primitives provided are difficult to use correctly, and errors in these facilities are difficult to detect and debug. Therefore, newer languages such as Java, ship with standardized and tested concurrency packages that provide features such as atomic variables, time-out locks, task scheduling framework for invoking, scheduling, executing and controlling threads. These eliminate many potential sources of problems such as deadlocks, starvation, race conditions, and excessive context switching between contended threads, increase reliability and maintainability of code, and reduce programming effort. Another issue with lock-based synchronization that the research community has been recently tackling is the serialization of entry into a critical section by many threads. This, if not minimized, can become a serious bottleneck if locks are not judiciously used, compromising scalability and performance of an application. For this, the research community has been studying transactional memory programming [7, 81, 83, 84, 86, 87, 128, 133, 134, 153–155, 174, 205] to ease the creation and programming of multi-threaded applications. Transactional memory introduces the notion

of a transaction, a region of code whose changes are indivisible or atomic. The changes are visible if the transaction is committed, and discarded when it is squashed. Transactional programming allows achieving higher performance as transactional regions can be entered concurrently by many threads. A transaction is executed speculatively and, at the end of the execution, committed if it did not violate any data dependencies, or squashed and re-executed serially, if it did. Transactions, unlike lock-based regions, are also composable, and therefore, two or more transactions can be combined into a larger transaction without knowing their internals.

Transactional programming is still a research proposal with several aspects such as programmability, debugging, and hardware support being actively investigated. While programmers inclined to write multi-threaded applications would benefit from transactional programming, it is not clear if the majority of end-user single-threaded applications will become parallel. In any case, research in performance improvements of sequential programs is important even in multi-threaded applications because the majority of them do not have abundant threads to run on future multicore systems that are expected to have hundreds of cores.

The creation of multithreaded programs remains, and is likely to remain, hard primarily because of the complexity in developing a parallel algorithm for a problem and the difficulty in debugging such a solution. Single-threaded programs, on the other hand, dominate the end-user market because of the relative ease in developing such programs and simplicity in debugging the sequential program execution. In this situation, we are also witnessing the ominous decline of performance improvements through micro-architectural enhancements that end-users have been provided with every generation of processors. With desktop systems already shipping with four processors [46, 70, 98, 106, 131, 188], and many more processing cores expected in the future, the question that system designers face is: How do we use the many cores to improve performance of a sequential (single-threaded) program?

Speculative parallelization. Speculative parallelization is a class of proposals that attempts to use the many processing cores by creating concurrency from a program but also maintaining the sequential program order. Several proposals have been studied for a decade now, and it remains a subject of interest in the research community [3, 40, 41, 45, 61, 81, 82, 125, 139, 150–152, 177, 182, 196]. Proposals in this category overcome the limitations of traditional parallelization by creating threads that are composed from the program and speculatively executing them in parallel. Additional hardware support is used to determine

threads that violate dependencies and squash them, and to enforce sequential program order of concurrently executed speculative threads.

The performance benefits of these proposals greatly depend on several aspects which include, the execution model defined by the composition of speculative threads, the ordering in which the threads are forked for speculative execution, and the hardware and software support needed to implement such an execution model. Among these, the execution model has focused on threads composed of specific regions of program code, which are forked for speculative execution as the hardware traverses through the control-flow graph of the program. I broadly refer to these proposals as control-flow based speculative parallelization.

1.1 Overview of Program Demultiplexing

In this dissertation, I present Program Demultiplexing (PD, in short), an execution paradigm based on speculative parallelization, for sequential programs. In PD, threads composed of methods (also, functions or subroutines), are "demultiplexed" from the sequential order, decoupling the execution of a method from the call site, which is where it is called in the program. In sequential execution, the call site of a method represents the beginning of execution of that method, and happens on the same processing core as the program. However, in PD, the execution of a method occurs on another available processing core, albeit speculatively, before the call site is reached in the program. Several such speculative executions of methods create concurrency in a program. The speculative execution is usually forked after the method is ready, i.e., after its data dependencies are satisfied for that execution instance. Its results are committed, if they remain valid, when the call site is reached by the non-speculative program.

Figure 1.1 illustrates the basic idea of PD. The figure on the left presents the sequential execution of a program with methods `A`, `B`, and `D` called by the program and method `C` called by `D`. The methods are executed in the same sequential order as they are called in the program. Parallelism between methods, even if it may exist, is not exploited. On the right side of the figure is the illustration of PD based execution of the same program. In the PD execution, `C` is forked for speculative execution first, followed by `B`, `A`, and finally `D`; the forking order of a method determined by its dependencies with the program. Speculative execution of `D` is (speculatively) used by `C`. Similarly, the program uses (commits) the speculative threads of methods `A`, `B`, and `D`.

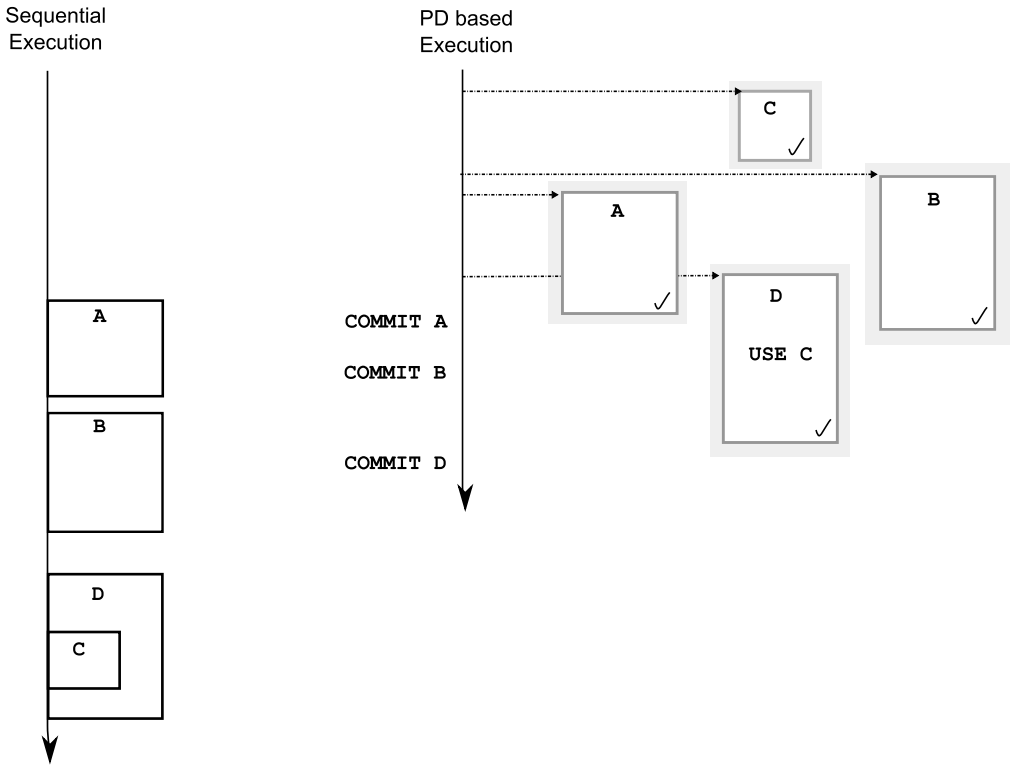


Figure 1.1: Program Demultiplexing overview. The sequential execution on the left represents execution of methods A, B, and D. Method C is called inside D. On the right side is the PD based execution. The methods are spawned for speculative execution (represented by the gray shading over the method). Method D uses the speculative execution of method C. Methods A, B, and D are committed when the call site in the program is reached. The speculative executions do not violate any data dependencies and this is indicated by the tick mark on the bottom right of the method's box.

The highlights of PD are as follows.

No programmer support. Like other speculative parallelization proposals, the conceptual framework of PD and the implementation discussed in this dissertation do not rely on programmer support. The implementation of PD in this dissertation requires automated software support from a compiler or a binary postprocessor to instrument for profile information, and to process the profiles to generate the necessary components for achieving PD based execution.

Suitable granularity of speculation for programs. Threads in PD are at the granularity of methods, a fundamental programming abstraction in modern programs. They are used by programmers to often

solve specific subtasks in a program and provide an interface for the rest of the program thus, acting as a natural means of separation of tasks. Having several methods performing several subtasks in a program is likely to expose parallelism between the methods that PD may exploit. Even though there is no rule that a method should solve a specific subtask, the advocacy of such software engineering guidelines is stricter when developing large scale applications because of their benefits of easier reusability and maintainability of program code.

Efficiently forking speculative threads. PD forks speculative threads for execution without the knowledge of their order with respect to the program. I refer to this model as "unordered forking". Previous systems for speculative parallelization forked threads in program order [40, 41, 45, 61, 81, 82, 125, 139, 150–152, 177, 182, 196] (or hierarchical tree ordered, in case of nested speculative threads [3, 161]) by speculatively traversing the control-flow graph. PD on the other hand, forks threads by also considering data dependencies of the threads and determining suitable points in the program when they may begin execution, thus more efficiently reaching distant parallelism than prior speculative parallelization proposals. The order in which the threads will be used is unknown at the time they are forked for speculative execution.

1.2 Dissertation outline

In Chapter 2, I provide an overview of previous speculative parallelization proposals and other closely related work. In Chapter 3, I introduce the concept of Program Demultiplexing, the reasoning behind the choice of speculating on method granularity, the components to enable PD based execution and examples of opportunities for PD in benchmark programs. In Chapter 4, I provide details of the software support required for this dissertation's implementation of PD. In Chapter 5, I describe the hardware support required for such an implementation. In Chapter 6, I present the evaluation methodology and experimental results. In Chapter 7, I present a summary of this dissertation, and discuss possible future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

Parallelism is one of the key means for improving the performance of computer systems, and Chapter 1 categorizes several means of exploiting it. Figure 2.1 is a spectrum chart of different proposals that exploit parallelism at different granularities; the caption provides more details on their organization. This dissertation is about speculative parallelization and expectedly, in this chapter, I first discuss this category of proposals in detail. I then cover other non-traditional parallelization and parallel programming means. Even though these developments are in the research areas of parallel programming and application design which expect programmers to write correct parallel programs, I discuss them because of conceptual similarities in the means of creating concurrency in a program.

2.1 Control-flow speculative parallelization

Speculative parallelization proposals can be categorized based on several criteria. Some of the key issues that are considered are as follows:

1. **Composition of speculative threads.** What should the threads be? Past proposals have considered loops, iterations of loops, continuations of methods (which is the program executed after the return of a method), or generic tasks obtained by dividing the program.

2. **Forking model of speculative threads.** How are the speculative threads reached and when are they forked for speculative execution? Past proposals have speculatively traversed control flow graph and forked speculative threads in that order or hoisted speculative threads to be forked before they are reached by the program.

3. **Hardware and software support.** This spans a large number of subtopics such as, software support to generate a program with speculative threads, means to perform speculative execution, support to store the speculative threads, ensure their correctness, and commit or invalidate them.

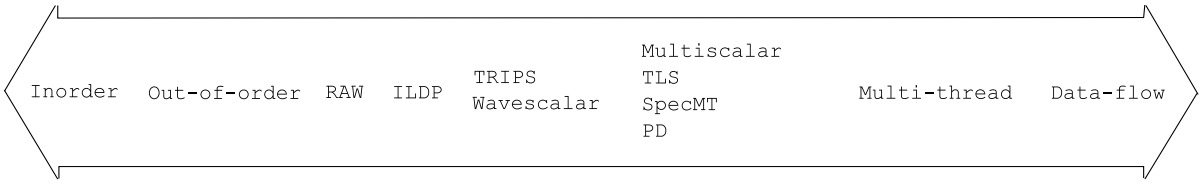


Figure 2.1: Spectrum of proposals that exploit parallelism at different granularities. On the left side is inorder processor that executes one instruction at a time and does not exploit parallelism in a sequential program. On the right side is data-flow processor that exploits all the parallelism in a program by executing instructions in data-flow order. In between the two are several proposals ranging from out-of-order processor that can reach parallelism in a window of few hundred instructions to research proposals on instruction level distributed processing systems such as RAW, ILDP, TRIPS, and WaveScalar, that further extend the reach for parallelism. This is followed by speculative parallelization proposals such as TLS, Multiscalar, SpecMT, and PD all of which have the ability to obtain parallelism thousands of instructions ahead although only at the granularity of a speculative thread defined by the system. Multi-threaded programming models, placed on the far right, could sustain parallel execution of several threads, depending on the application and its developer.

The first two aspects mentioned in the above list define the execution model of a speculative parallelization proposal, and the last aspect defines the implementation of the model. I will next describe several speculative parallelization proposals and in particular, discuss the composition of speculative threads and the forking model. I do not describe the differences (often, subtle) in the hardware support needed for the following reasons:

1. Hardware support does not strictly adhere to a particular concept or proposed execution model. Usually, hardware support for one model can be used to implement many other speculative parallelization models.

2. Hardware support for speculative parallelization has been covered in detail by many previous dissertations.

3. The aim of my thesis research was to identify and substantiate the limitations of reaching parallelism in speculative parallelization proposals. The hardware support is notably not in depth in this dissertation for this particular reason. Discussing the concept of the previous proposals in detail will help uncover the limitations and design new means for alleviating them.

In the following sections, I categorize speculative parallelization proposals into four categories. They are: (i) the generic Multiscalar-based, (ii) loop-based, (iii) method-continuation based, and (iv) transaction

System	Focus	Software	Hardware
Multiscalar [68, 69, 177]	Tasks	Identifying and compiling tasks	Special purpose hardware with processing units with fast operand value communication
SPSM [62]	Generic	Explicit software based speculative parallelization and analysis	Multiprocessor with instruction set extensions
TLS [182–184]	Loops	Profile based analysis	Multiprocessors with support for speculative execution, value prediction
DMT [3]	Loops and method continuations. Hierarchical tree-ordered forking	-	Multi-threaded hardware, support for speculative execution of threads, value prediction, and ordering
Superthreaded [196]	Loops	No data speculation in threads. Data values are sent to consumers with explicit compiler insert instructions	Multiprocessor with instruction set extensions
Hydra [82]	Loops and method continuations	Profile	Multiprocessor with speculative execution of threads
Zhang et al. [208]	Loops	-	Multiprocessor with speculative execution of threads
MAJC [193, 194]	Loops and method continuations	VLIW compilation	Support for speculative execution of threads
Cintra et al. [45]	Loops	Profile based analysis	Hierarchical CMP hardware with support for speculative execution of threads
Clustered SpecMT [122, 125]	Generic	-	Based on clustered microarchitecture with support for speculative execution of threads
Marcuello et al. [123, 124]	Generic	Profile based analysis to determine suitable regions and forking points in program	Multiprocessor with speculative execution of threads
Module-level [200, 201]	Method continuations	Profiling	Multiprocessor with speculative execution of threads
Jrpm [41]	Loops	Java-based with profiling support	Multiprocessor with speculative execution of threads
IMT [144]	Generic	Identifying and compiling tasks	Multithreaded hardware with speculative execution of threads
Du [61]	Loops	Compilation framework for identifying spawning point	Multiprocessor with speculative execution of threads
TCC [81]	Generic, programmer specified	Transactional compiler	Transaction based multiprocessor hardware also used speculative parallelization. Provides programmers with specifying commit ordering
Pinot [138]	Generic	Compiler infrastructure to extract speculative threads from programs	Speculative multi-threading processor that supports fast operand value communication, low latency inter thread communication with an update-based cache coherence protocol and instructions for thread termination
Prabhu et al. [150, 151]	Loops	Analysis of hindrances and opportunities for speculative parallelization	Hydra hardware for speculative execution of threads
Mitosis [152]	Generic	Elaborate compilation framework for speculative parallelization and choosing program points for forking of speculative threads	Multiprocessor with speculative execution of threads
OoO Spawn [161]	Loops and method continuations. Hierarchical tree-ordered forking	-	Hardware extensions to deal with ordered tree based forking
Bulk [32]	Loops and method continuations	POSH compiler	Hardware with speculative execution of threads, signatures used for read and write sets
Subthreads [49]	Generic	-	Multiprocessor with speculative execution of threads, and support for dividing threads into multiple speculative subthreads and checkpointing
PolyFlow [2]	Generic	-	Multithreaded processor with speculative threads spawned from immediate postdominators
IPOT [199]	Generic	PL support, profile driven detection of good candidate for threads	Multiprocessor hardware like TCC

Table 2.1: Summary of speculative parallelization proposals

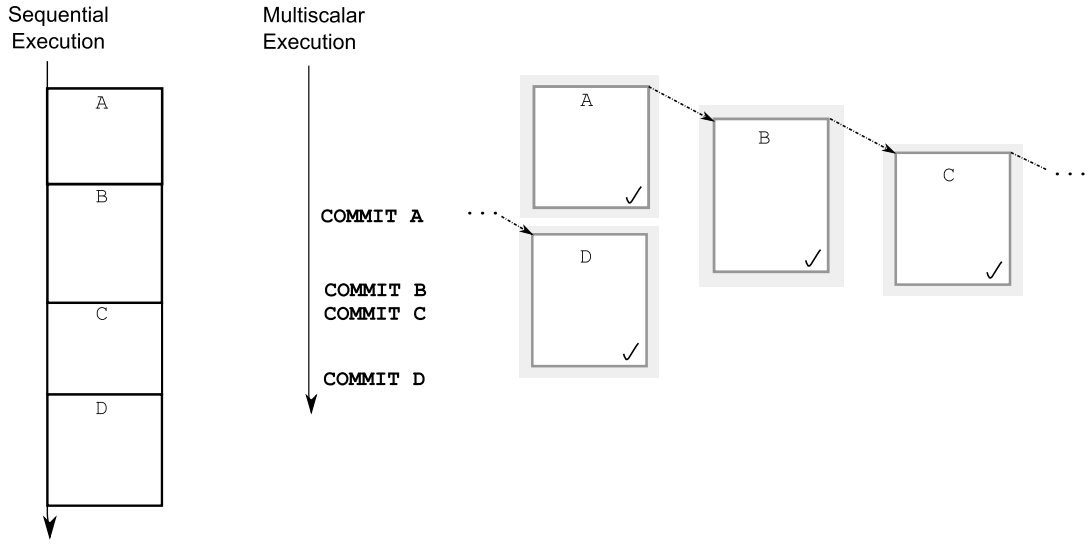


Figure 2.2: Forking of speculative threads in Multiscalar. On the left is the sequential execution of a program with four tasks, A, B, C, and D. On the right side of the figure is the Multiscalar execution, with three processing cores available for speculative execution of tasks. A, B, and C are scheduled one after another with the tasks identified by means of a task predictor. Speculative data is communicated from an older task to younger task. Tasks are committed by the program as they complete. The execution assumes no violation of dependencies; no tasks are therefore, squashed. The gray shading around tasks indicates that they are speculative. The tick symbol placed at the right side corner of a task indicates that the task is valid and did not violate any dependencies. This convention is followed throughout this dissertation.

based speculative parallelization. Table 2.1 lists all speculative parallelization proposals, the regions of program code that are speculatively parallelized, and the hardware and software support needed for the implementation.

2.1.1 Multiscalar-based speculative execution

Multiscalar [69, 177] and other similar proposals [122, 125, 138] dealt with speculative parallelization of an entire program. A Multiscalar system uses special purpose hardware with processing cores connected in a ring topology. The hardware allows communication of register values from one processing core to another. Many other proposals have a similar software model but instead use a typical multiprocessor or multicore system.

The core of these proposals is a software subsystem consisting of a compiler that divides a sequential program into tasks, a task ranging from few instructions to several basic blocks. The hardware steps from one task to another in the sequential program, assigning each of these tasks to processing cores for speculative

execution. The motivation behind this approach is to capture local data dependencies between instructions within a task, and to minimize data and control dependencies between tasks closely coupled in program order. To establish this, a compiler (or some other software such as a binary rewriter) uses program analysis —static or dynamic with profile information— to choose suitable boundaries for tasks in the program to maximize parallelism between them.

Figure 2.2 shows the Multiscalar execution model on a system with three processing cores. The figure shows dynamic execution instances of tasks A, B, C, and D in a program on three processing cores. A task is predicted and forked for execution on an available processing core by a task predictor. In the example, B is predicted from A, C from B, and so on. If for some reason, B is not predictable, task A has to finish execution in order for the control flow to resolve, to identify B. Since the execution model identifies and assigns tasks for speculative execution based on sequential program order, the commit ordering of tasks is the same as the fork order of tasks.

A task is speculatively executed and its speculative changes are committed if it reaches the head of the task queue, or squashed if a dependency is violated. For example, task C may read from a location before task B can write to that same location. Therefore, C has to be squashed and re-executed to ensure that the right value is read by C. Data dependencies may exist between tasks, and executing tasks concurrently may lead to violation of such dependencies especially because the tasks are scheduled only according to the control flow. Many avenues were taken by Multiscalar and related proposals to alleviate this problem:

1. By allowing communication between speculative tasks, data values written by an older speculative task is passed on to newer ones.

2. By value predicting data values, data dependencies between speculative threads are broken.

3. By dynamic insertion of synchronization primitives in speculative threads, to ensure that a newer speculative task proceeds only after the older speculative task has performed the store operation.

Multiscalar allowed multiple outstanding executions per processor, i.e., executions waiting to be committed or squashed, while other proposals required that a task commit or squash to begin executing the next task on that processing core. Having only one active execution per processing core simplifies the requirements of buffering speculative data in cache, unlike having multiple outstanding executions, which

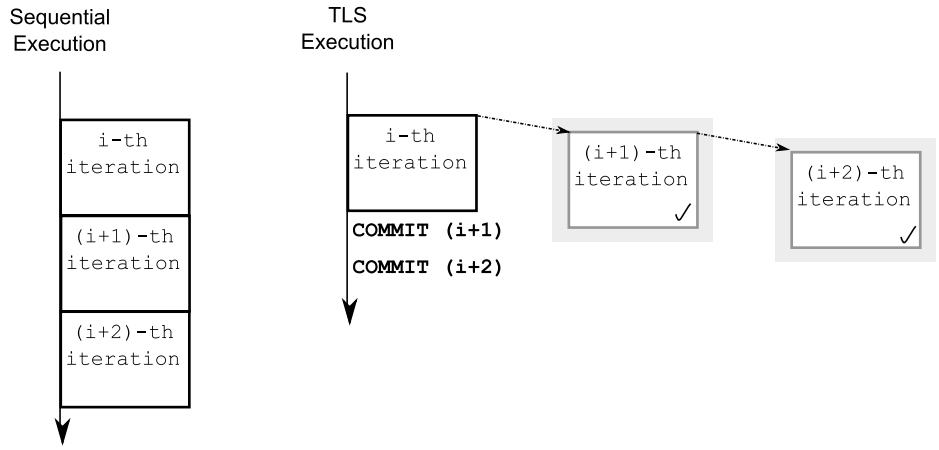


Figure 2.3: Loop-level parallelization. On the left side is the sequential execution of several iterations of a loop. The right side of the figure illustrates loop-level parallelization performed by a TLS-like system. Loop iterations are assigned to processing cores for speculative execution. Dependencies that may exist between iterations can be resolved with one or more of the means described in Multiscalar-like system.

requires cache references from different speculative threads to be identified. On the down side, it may lead to holding up of a processing core until the task executed by that core is committed or squashed, an issue of concern, if tasks in the system are not balanced.

2.1.2 Loop-based speculative execution

Many proposals specialize the Multiscalar-based approach and focus on specific regions of code such as loops since significant portion of a program's execution time is spent in loops [45,82,182,208]. An iteration of a loop or the entire loop is treated as a speculative thread and concurrently executed with many other speculative threads. Figure 2.3 illustrates the TLS system [182], which follows this approach to loop-based speculative parallelization.

2.1.3 Hoisting of speculative threads

An alternate approach to speculative parallelization is the hoist-based speculative execution model commonly used when only selected regions of program code are chosen for speculative threads. The idea is analogous to compilers hoisting a load instruction in the program to tolerate (or amortize) the many cycles that may be taken to obtain the value depending on where the data may be located in the memory hierarchy. Several proposals [61,117,124,152] take a similar approach of hoisting (albeit, speculatively) the forking of a

thread before the thread's head instruction is reached during program execution. When the program reaches the thread's head, it waits until the thread finishes execution, and commits the thread if no dependencies were violated. The execution model is illustrated in Figure 2.4.

Hoist-based speculative parallelization proposals use a profile driven approach to identify candidate program code for speculative threads and the most suitable fork points. Since a thread is hoisted with respect to the program execution, alternate means are necessary to provide the live-in register values that may be accessed by the thread. The common approach is to use a value predictor to predict the live-ins. Another approach is compute-based prediction, in which some instructions are executed to compute the likely live-in values. These values are then provided to the speculative thread. Before a speculative thread can be committed, the used live-in values should match the values generated by the non-speculative program. One such compute-based predictor is the "pslice" used in Mitosis [152]. A "pslice" for a speculative thread is obtained by identifying the live-in registers and constructing a backward slice of instructions from the head of speculative thread back to the fork point in the program. The producers of the live-in values and any transitively dependent instructions compose the backward slice.

The key assumption with the hoist-based speculative execution is that there are several data independent instructions between the thread's fork instruction and its head instruction in the program. Therefore, it is anticipated that the program will (partially) cover the cycles it takes to execute the pslice and the thread speculatively, beginning at the fork point, before the thread's head is reached.

2.1.4 Method-continuation speculative execution

Method-continuation level parallelization (MCLP), also called module-level parallelization is another specialized form of speculative parallelization. The proposals in this category [40, 41, 200] focus on speculating past a method call, i.e., program that follows after a method returns, also called the method continuation. It is a straightforward means of parallelization because of the near definite control flow reachability of the continuation when the method is called (the rare case is when programmer has arbitrary control flow in the program that never returns from the method).

The parallelism that MCLP exploits is the plausible data independence of the method's computation with that of the method's continuation. There are two forms of dependencies that may exist between the method

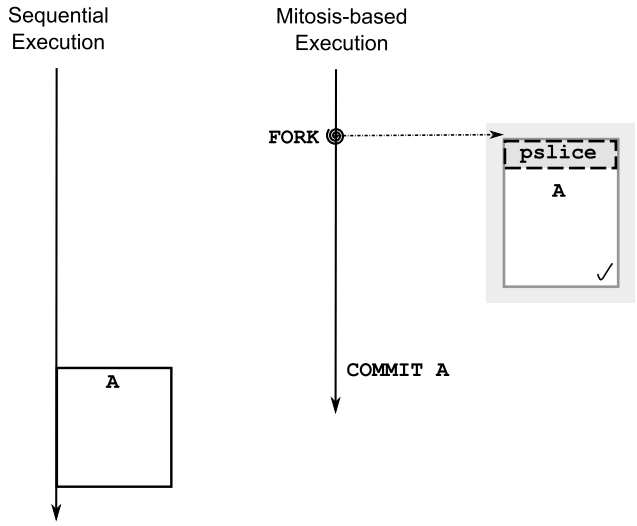


Figure 2.4: Mitosis-based parallelization. On the left side of the figure is sequential execution of a program with A representing loop or task in a program. On the right side is Mitosis-based execution achieved by inserting fork instruction that spawns a speculative thread, pslice that is executed to provide the live-in registers for the thread, followed by the speculative execution of A. The speculative execution is committed when A is reached in the program assuming no dependencies were violated and the live-in registers were computed correctly by pslice.

and its continuation: (i) the return value of the method that may be later used by the program, and (ii) the side effects, i.e., modifications that a method may make to the program's global state that may be accessed by rest of the program. The former dependence can sometimes be circumvented since return values are often discarded by the program, or speculated on, especially when it is highly predictable based on previous values returned by that method.

Out-of-order Forking. One unique aspect of MCLP is the ordering in which speculative threads are forked. In the simple case shown in Figure 2.5, there is only one speculative thread running until the method A finishes execution. The model, when extended to perform speculative execution for every method encountered, may not fork threads in program order.

Consider the example in Figure 2.6 in which threads are forked by both the method and its speculative continuation. The ordering in this model is hierarchical tree-based (also referred to as out-of-order based), in which a speculative thread is ordered sequentially with respect to its parent speculative thread, if one exists. The model is complex and can be detrimental to performance if method-continuations are forked

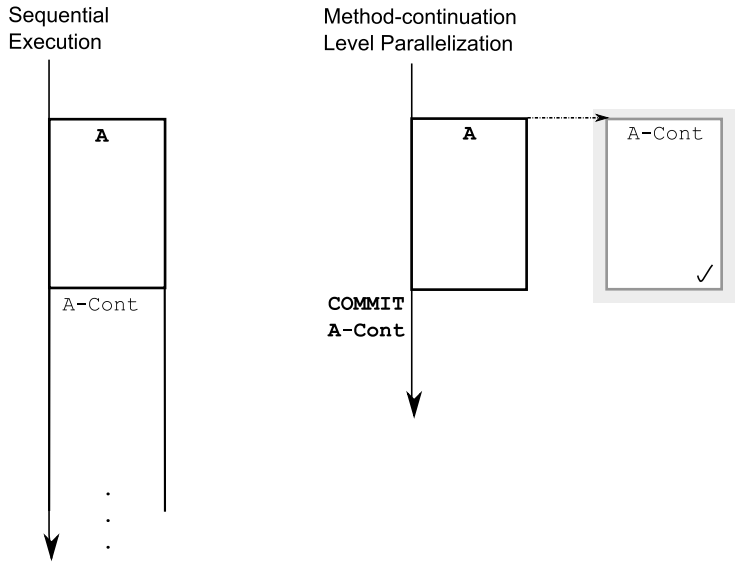


Figure 2.5: Method-continuation level parallelization. The sequential execution shows method A and the rest of program marked as A-Cont. In MCLP, when the method begins execution, a speculative thread that executes the continuation of the method is spawned. Assuming that there no violations of dependencies, the continuation executes until the method returns back to the program. At this point, the results of the speculative thread are committed, and the program continues execution from that point onwards in the program.

indiscriminately for every method encountered [201]. Additional hardware support is also needed to support this model [3, 161].

2.1.5 Transactional memory

Transaction-based execution, a central idea in databases, has been proposed to overcome the difficulty in achieving scalability and correct execution with the use of locks as synchronization primitive. The key feature of a transaction is the notion of atomic execution, i.e., all program state changes made by a transaction has to be either visible or not visible in its entirety to the rest of application. Multithreaded or parallel programs typically use locks for synchronization when multiple threads may conflict, for example, to protect entry to a critical section. Locks serialize the multiple threads to eliminate conflicts—the order of serialization is the order in which the threads acquire the lock. Transactional memory has been proposed to overcome the impediments of locks as a synchronization primitive. While it has commonly been a software implementation [84, 86, 133, 174, 205], more recently researchers have been considering hardware support [7, 81, 83, 87, 128, 134, 153, 154] and hybrid approaches [55, 107, 167] for transactional execution to reduce

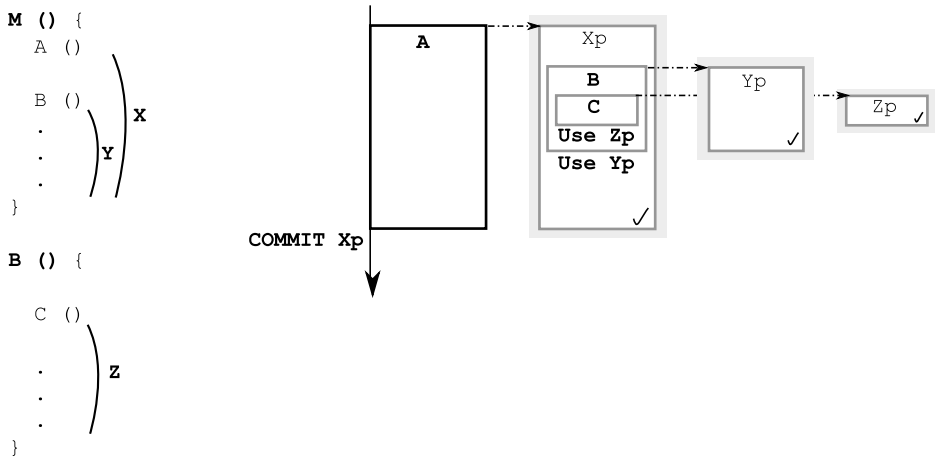


Figure 2.6: Nested forking in Method-continuation level parallelization systems. On the left side of the figure is a snippet of a program. Method M calls two methods A and B. Method B calls method C during its execution. The continuations are marked with X, Y, and Z. Partial executions of these continuations are indicated as Xp, Yp, and Zp. During execution of M, Xp is speculatively executed. Methods encountered in M or Xp spawn more threads. In the example, Xp spawns Yp when B is called, and Zp when C is called by B. When C finishes execution Zp is committed. Similarly, Yp is used/committed when B finishes execution.

the overheads. The hardware support identifies conflicts with several concurrently running transactions. Transactions that violate dependencies are squashed and re-executed serially by the hardware.

Transactional memory and Speculative parallelization. Transactional execution and speculative parallelization have similar requirements in terms of speculatively executing a set of instructions, determining if any instruction violated dependencies, and acting accordingly. However, there are some dissimilarities. Speculative parallelization deals with a sequential program in which the speculative threads are ordered. On the other hand, transactional programming was introduced to deal with multi-threaded programs, and therefore, the transactions from many threads do not have a predetermined order. Likewise, the process of determining if a transaction violated any dependencies occurs only among other concurrently running transactions. However, in speculative parallelization, the conflict detection for a given speculative thread is not only with other concurrently running threads but with all the instructions being committed in the program. Another important distinction between the two is the lack of communication between concurrently running transactions. In speculative parallelization, speculative values are commonly passed between threads.

Some transactional system proposals [32, 81] have extended their systems to perform speculative parallelization. One such proposal is TCC [81] which, unlike other transaction-based systems, requires every

instruction to be associated with a transaction. Programmers divide a sequential program into transactions, and provide the ordering of transactions with two parameters: sequence and phase numbers. These numbers control the ordering of transactions in a program and wait conditions before it can begin its execution; within a given sequence number, transactions are committed in increasing phase order. This allows the programmer to either achieve sequential ordering, completely unordered transactions, for example, when iterations of a loop in a sequential program are independent, or more complex ordering specifications. Like the Multiscalar based model, these transactional systems also spawn speculative threads in program order and commit in that same order or another arbitrary order specified by the programmer.

2.1.6 Discussion

Potential benefits. The performance benefits of a speculative parallelization model depend on several factors. Foremost, is the parallelism that exists in the program (as studied by Austin et al. [12] and Lam et al. [108]) which depends on the characteristics of the program, the algorithm used, and the programming implementation. The second factor is the execution model of the speculative parallelization system. It includes the composition of the speculative threads to maximize performance potential [95, 96] and how the threads are spawned for execution [161]. Finally, the implementation also plays a crucial factor. This includes all the experimental parameters such as communication latencies, number of processing cores for performing speculative execution, and any other resource constraints. The first factor is solely dependent on what problem application developers are trying to solve with a computer program and how they solve it. The second factor is the crucial aspect and determines how the hardware and software support can be provided.

Limitations. An important assumption I have made in my discussion so far is that the speculative threads never violate dependencies and the system is assumed to have no mis-speculations. In practice, this is unlikely to be the case. The general principle in speculative parallelization is to traverse the control flow graph of a program speculatively at the granularity of a task or thread which can usually vary between a few instructions to several basic blocks. This allows the execution model to encapsulate several control flow decisions inside a task and reach parallelism in a program more distant than instruction-level parallel processors. A key limitation of this approach, as illustrated in Figure 2.7, is that a mis-speculation in a

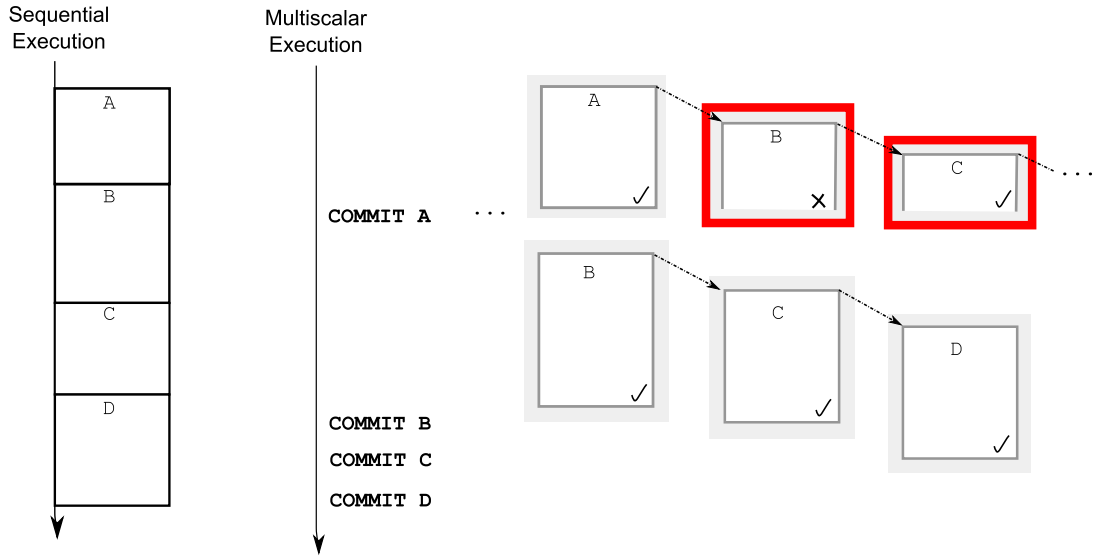


Figure 2.7: Mis-speculations in Multiscalar-based speculative parallelization. Task B violated dependencies and must be squashed. Tasks that follow B, for example, C may not have violated dependencies, but are still squashed as per the execution model.

thread results in squashing that and all other threads that follow it. In addition to this, a thread is scheduled only according to the speculative control flow and without any consideration of its data requirements. This aspect limits the ability to reach “distant” parallelism because intermediate data dependencies may lead to mis-speculations in threads, further increasing the chances of squashing the threads. The probability of occurrence of these violations increases as the number of instructions considered for parallel execution increases. This may result in the discarding of large amounts of (possibly independent) work and thereby, delaying the reachability of a distant independent task even further. For example, indiscriminate usage of method-continuation level parallelization has been shown to hurt performance as the speculative threads are forked on reachability of control flow without any analysis of their data dependencies [201].

The speculative parallelization model has been quite successful in extracting reasonable amounts of parallelism from applications. However, the limitations in the ability to reach “distant” parallelism must be addressed as its exposure and exploitation is likely to be crucial in the future, especially, as the number of processing cores increase.

2.2 Other related speculative execution models

Shadow processing. Shadow processing system introduced the notion of creating a copy of an application with additional code in it referred to as the "shadow" program [145]. This shadow program for example, can be used to perform additional checks such as null pointer checks, initializing and freeing memory allocated, and any other violations that may crash the program. To speed up the execution of the shadow program the system communicates few key values from the main program to the shadow. This eliminates some computation and minimizes the number of instructions in the shadow. Several proposals have enhanced this execution model with different forms of main and shadow programs, for different purposes. For example, Sundaramoorthy et al. proposed the Slipstream system [185] (that originated from AR-SMT [163], a hardware fault detection system), to improve the performance of the program. The main program is speculatively optimized, and the shadow program obtains values from the main program, fetches and executes instructions more efficiently because of the hints provided, and/or verifies if the main program execution was correct. The Master-slave speculative parallelization system [214] further enhances the model, by parallelizing the execution of shadow program, and speculatively optimizing the main program. The parallel shadow program ensures that the speculative main program executes correctly. Like Slipstream execution, data values are passed from main to shadow program to enable parallel execution.

Pre-execution. There have been a number of research proposals investigating execution models that can broadly be classified as speculative data-driven multithreading [36, 47, 48, 64, 101, 118, 135, 164–166, 178, 211, 212] (also commonly referred to as pre-execution, helper, scout, or subordinate threads). This category of schemes typically targets cache misses and branch mispredictions, two performance impediments in processors. In this approach, a thread consisting of a pre-defined chain of dependent instructions leading up to a load or branch instruction (i.e., a data-driven thread), is spawned from specific points in the program. The set of load or branch instructions for which the threads are generated are usually identified by profiling and are those that often miss in the cache or result in branch mispredictions. Each thread creates the performance degrading event (cache miss or branch mispredict) earlier than it would have occurred in normal program execution, thereby allowing its latency to be overlapped with other program instructions (that occur prior to the event in the sequential program). An ensemble of such threads, executing on multiple processing

cores, can effectively overlap the latencies of multiple performance degrading events, thereby ameliorating their performance impact. In order to reach the cache miss or branch misprediction, pre-execution techniques construct backward slices of dependent instructions from the problem causing instruction to the point where the thread can be forked. Several proposals in this category cover various hardware and software means of identifying the problem instructions, generating the slice of instructions to execute, and support for executing them in a system.

2.3 Data-flow machines

An alternative approach to von Neumann machines is data-flow machines, an intuitively appealing data-driven execution model, that have been studied extensively in the past several decades. Data-flow machines [10, 56, 58, 76, 78, 103, 143, 168, 175] are fine-grained data parallelism machines that execute programs expressed as data-flow graphs. Since communication between execution units in the system is fast, scheduling happens at the granularity of instructions. Control-flow is eliminated in programs and unlike von Neumann machines, there is no synchronization required between control dependencies.

The dataflow execution model has many appealing properties, including the ability to expose and exploit arbitrary granularities of parallelism. The parallelism in an application is constrained only by the data dependences in the application, and not by arbitrary control dependences that are an artifact of the imperative programming language. Despite the power and elegance of the dataflow execution model, it has not been widely adopted. An important reason for this is the coupling of the execution model to the data-flow based programming languages [10, 59, 129, 136]. Many of the applications that were easy to express parallelism in such languages had significant inherent parallelism. This parallelism may also have been easily exploited in imperative programming, and similar benefits may be achievable. The data-flow languages, unlike imperative programming languages, were difficult to write a large class of programs due to the lack of available features. The other issue with the fine-grained data flow architectures is the enormous scheduling and communication overhead. To handler this, Sarkar and Hennessy [170] and Iannucci [91] proposed statically partitioning a data-flow program into subprograms and executing them in a data flow order; subprograms by themselves were executed sequentially.

The dataflow execution model also did not explore the notion of speculative execution, which is likely to be the key to extracting parallelism from a wide range of applications. Despite the lack of commercial success of the general dataflow execution model, we can make some observations about the impact of dataflow execution concepts on other program execution paradigms. The ubiquitous dynamically scheduled superscalar paradigm uses dataflow execution principles, coupled with speculation, for a small group of instructions that have been extracted from a sequential program (written in an imperative programming language). The WaveScalar system [187], an instruction level distributed processing system, executes instructions whose firing rules are determined by the data-flow of instructions in a "wave" which represents boundaries in a sequential program specified by a compiler. Similarly, the PD execution model borrows from dataflow execution model. Like dynamically scheduled superscalar processor, it applies dataflow execution principles, coupled with speculation, to programs written in imperative languages. Unlike dynamically scheduled superscalar processor, it uses program methods rather than instructions as program units (i.e., nodes in the dataflow graph), and processing cores rather than functional units to execute the program units. Moreover, the mechanics of how program units are launched for execution (on to processing cores) and how their results are gathered and committed will be different.

2.4 Functional programming

In functional programming languages, computation is represented as a mathematical function. A program's execution is expressed in a functional manner: a function's execution is triggered when its inputs, which denote other functions, are available. Functional programming languages have been very conducive for parallel execution. There have been several projects in concurrently executing methods in purely functional languages, as they do not have any side effects [74, 171]. MultiLisp supports evaluation of parameters in parallel, and allows programmers to express explicitly the concurrency of a method [162]. Knight presented speculative parallelization of Lisp programs [104]; Lisp is not a purely functional language and hence the need for speculative execution.

2.5 Other relevant parallel programming models and languages

PD resembles message-passing based parallel programming paradigms such as the seminal Actors [88] and Linda [31] models. In Actors, concurrent objects are spawned and communicate with each other solely via messages, and allow concurrency even within a single actor. The execution model is message driven, thus allowing latency tolerance. Charm [90] was an implementation of the Actors model. The Linda system does not share messages like Actors. Instead, threads generate results as tuples that are held in the tuple space. The tuples are not intended for a specified receiving thread. A live tuple, at some time during the execution, fires, carries out some computation, and transforms to a data object tuple that can be accessed by another receiving thread.

In the past, numerous parallel language constructs, language designs, and libraries have been proposed for creating parallel programs. These include ABCL [207], Concurrent Smalltalk [54], CA language for J-machine [42], pC++ [25,121], C** [110], a data parallel variant of C, Mentat, a concurrent C++ [77], pSather for Eiffel [65], ESP-C++ that supports concurrent objects [112], transparent remote method invocation, and blocking as well as non-blocking, future based messaging, CC++ that provides parallel constructs for C++ [6, 33], POOL-T [93], Amber [38], OOMDC/C [43], Charm++ [97], and Cilk, a runtime system that manages several threads consisting of Cilk procedures [24].

More recently, several proposals have been studied in the programming languages and applications domain to exploit different forms of parallelism found in applications without any speculation. These include software support to extract parallelism or libraries and/or new languages to express it. Some such proposals are discussed next.

Martel et al. [126] present different parallelization strategies to exploit distant parallelism in the SPECint95 suite. Time-shifted modules [213] is another software-based approach to execute modules which have limited interaction with the program concurrently. The modules communicate by means of message queues. DSWP or decoupled software pipelining [140] exploits fine-grain thread level parallelism in loop bodies of programs. The execution of a single iteration of a loop is subdivided and spread across multiple processing cores in a multicore system. When the compiler can create subdivisions that form an acyclic dependence graph, each subpart can be independently executed forming a pipeline. DSWP allows better utilization of cores and better latency tolerance when such pipeline parallelism can be extracted from

the program. Palatin et al. [142] convert some SPEC integer programs into better software engineered, component based programs, which consist of tightly encapsulated components, each isolated from the rest. Communication between components is explicitly performed in the program. They assume a simultaneous multi-threaded hardware with support of very light-weight locking support. Ranger et al. [157] evaluate the MapReduce model [57], a model created by Google for application development on data-centers with thousands of servers, for multi-core systems. The authors provide an API for efficiently writing code, and using the runtime system for automatically creating threads, scheduling, and partitioning them across processing nodes. Zhong et al. [209] propose Voltron, an architecture that exploits both instruction-level parallelism and fine-grained thread level parallelism by extending multicore system with a low-latency operand communication network between processing cores. The hardware exploits two different modes. The coupled mode is the lock-step operation of a processing core with other cores. In this mode, compiler orchestrated control flow is executed on many processing cores similar to a VLIW processor. In the decoupled mode, cores operate independently on separate fine-grain threads. The threads are used to exploit DSWP [140] and speculative execution of loop iterations.

In the programming languages domain, object-oriented managed languages such as Java and C# have several constructs for creating light-weight threads in an application. The future primitive in Java [116] used in conjunction with a method call, allows a program to spawn a light-weight thread that executes a method, while the program continues to execute beyond the call. The programmer must probe and determine whether the method's execution has completed, and then access the return value or program state produced by the method. The execution is conceptually similar to MCLP model discussed earlier; however, MCLP executes the program along with the method speculatively to ensure no dependencies are violated. The safe futures [204] work achieves MCLP based execution but by means of software based speculative execution within Java virtual machine. Like MCLP, the continuation of the method is executed, and any dependencies violated by this code with the method results in the managed system rolling back all the changes made by the method's continuation. In C# language, the delegates primitive allows programmers to create several tasks/methods, which are executed concurrently, with results of the threads used as and when needed [19, 20]. X10 [37] is an object-oriented programming language intended for creating high-performance parallel programs capable of using several hundred cores in a multicore system. The stream programming model

is based on defining kernel functions or operations which operate on each data element of a stream in a pipelined fashion. The StreamIt [75] and Brook [27] languages are based on this programming model. They are intended for easily writing parallel streaming applications (in particular, DSP, image, and video processing applications), that can run efficiently on multicore systems.

2.6 Chapter summary

In this chapter, I discussed previous speculative parallelization proposals. They were classified into proposals that speculatively parallelized: (i) entire program by dividing them into tasks, (ii) loops and loop iterations, (iii) chosen regions of program code by hoisting their speculative execution and overlapping them with the program, (iv) method continuations, and (iv) transactions specified by programmers. All of these proposals spawned threads only in control flow order and/or did not consider data dependencies of a thread to determine the most suitable point for execution.

I also discussed several parallel programming languages, constructs, primitives, and libraries that help in parallelization of programs. Although these were not speculative techniques, and relied on programmers or software (such as compiler) to express and extract parallelism from a program, many are conceptually similar to PD.

The next chapter will discuss the Program Demultiplexing execution model in detail. It will cover the evolution of programming methodologies and the role of methods in current programs, qualitatively arguing that methods are suitable for speculative execution. It will then cover the execution model and the means to achieve such an execution.

CHAPTER 3

PROGRAM DEMULTIPLEXING MODEL

Over the years, programming methodologies and styles have seen a dramatic change. Early computers were programmed in assembly language, which gave way to programming in first-generation of high-level languages such as FORTRAN and BASIC. FORTRAN has continued to remain a popular language for writing scientific applications because of the advancements in automatic parallelization. These languages were followed by a more structured and modular procedural programming style such as C, which led to modular development, the division of a complex application into multiple files with separate compilation. Recent trends have moved towards object-oriented programming languages such as C++, which further emphasizes modularity. Despite the overheads when compared to a procedural programming style, the object-oriented style has gained popularity as it allows for a streamlined development of large-scale applications. This has helped in cutting down the chances of bugs, ease of maintaining large-scale programs, and facilitating the reuse of code across different applications. The most recent development in programming languages has been the prolific use of managed object-oriented languages such as Java and C#. These languages have been gaining significance because of their object-oriented programming style, and benefits such as automatic garbage collection of allocated but unused memory, and type safety checks obtained from being executed on a runtime system. Any proposed parallel execution model should be suited for the emerging multicore systems and for contemporary programming style and languages. It should match the characteristics of those styles, as programming practices determine the practicality of a parallel execution model.

PD leverages the programming style of encapsulating related computation as a method, to perform speculative execution at that granularity. The focus of this chapter is further discussion of this choice for speculative execution and the framework of PD. This chapter is organized as follows. I present background material on methods, their semantics and memory state, and compilation into a sequential program. I then provide reasoning behind the choice of methods for speculative execution in PD. Then, I discuss the PD

execution framework, implementation sketch, and present some examples of the execution model from benchmark programs.

3.1 Methods

A method, also referred to as a function, procedure, or a subroutine, is a portion of code written to perform subtasks within a larger program. In this dissertation, I will use "method" to denote any of these variants written in any programming language even though the different synonyms have subtle, but not strict differences in connotations depending on the programming language used. The general interpretation of the terms is described next.

Subroutine, commonly used in BASIC, is the most generic term. A *procedure* is used to represent a subtask that does not return any value back to the program, and a *function* is a procedure that returns a value back to the program. Both the definitions are commonly associated with the Pascal programming language. A *method*, commonly used in object-oriented languages, has a stricter definition, and denotes the implementation of a subtask for a given "object" in the program. The purpose of a method is to provide a mechanism for accessing and modifying the private data stored in an object, an instantiation of a class. A method accesses private data of its object in a way consistent with the intended behavior of the object. Rather than thinking of a method as a "sequence of commands" like in subroutines, a programmer using an object-oriented programming language will consider a method to be an "object's way of providing service".

3.1.1 Benefits of methods

A program is written as several methods, with each method separately defined in the program. The methods can be called one or more times in the program and can be shared with other programs through libraries and packages. Methods avoid the undesirable situation of repeating code that performs the same computation in multiple places in the program. This saves space and allows for faster loading of the program into memory and better use of cache space available in the hardware. Methods also form a logical segmentation of the entire problem, enable easier visualization of the structure of a large and complex program, make it easier to debug and maintain, and can (often) be used by people other than the programmer who constructed the method.

```
                                        qsort:
                                       326     pushl   %ebp
                                       327     movl    %esp, %ebp
void qsort (double *a, int l, int r) { 328     pushl   %edi
    int i, m;                          329     pushl   %esi
                                       330     pushl   %ebx
    i = l + 1;                         331     subl    $32, %esp
                                       332     movl    12(%ebp), %eax
    if (l >= r) return;                333     movl    16(%ebp), %edx
                                       334     movl    8(%ebp), %edi
    ...                                335     movl    %eax, -16(%ebp)
}                                      336     movl    -16(%ebp), %esi
                                       337     movl    %edx, -20(%ebp)
                                       338     movl    -20(%ebp), %eax
quick_sort (double *a, int n) {        339     incl    %esi
    qsort (a, 0, n);                   340     cmpl    %eax, -16(%ebp)
}                                      341     jge .L69
```

(a) C program code

```
quick_sort:
426     pushl   %ebp
427     xorl    %ecx, %ecx
428     movl    %esp, %ebp
429     subl    $12, %esp
430     movl    12(%ebp), %eax
431     movl    %ecx, 4(%esp)
432     movl    %eax, 8(%esp)
433     movl    8(%ebp), %eax
434     movl    %eax, (%esp)
435     call    qsort
436     leave
437     ret
```

(b) Compiled assembly code

Figure 3.1: Compilation of methods `qsort` and `quick_sort` written in C, into assembly code. The figure illustrates `quick_sort` passing the three parameters `a`, `0`, and `n` to the method `qsort`, `qsort` accessing them, and `qsort` declaring/using local variables in stack.

3.1.2 Program state of a method

Computation in a method can access and modify two kinds of program state: *local* and *global*. I next describe the typical use and implementation of these two in programs.

Local state The local state is the program data that is not visible outside the method, and is commonly implemented using a stack. The stack is a FILO (first in last out) structure and is a specially reserved part in memory. A stack is used for local variables within a method and sometimes for passing parameters to the method from a program. Every time a method is invoked, it gets a new "frame" on the stack with a new place to store its local variables. The variables are always at the same offset within the frame, but the frame can be at different starting addresses within the stack. When the method exits, the frame is removed from the stack. This deallocates all local variables making cleanup of memory used very easy. One common way for the program to provide parameters to a method is achieved by the caller method writing the parameter values to its stack frame, and the callee accessing them from the caller's frame.

Figure 3.1 illustrates the compilation of a snippet of C program into assembly code.[1] Figure 3.1(a) is the C source code, in which variables `i` and `m` are local to `qsort`, as they are declared inside the method `qsort`. Variable `i` is initialized immediately after the declaration. The second method in the program code

[1]The dissertation will present assembly code and stack layout based on the Intel x86 instruction set architecture to match the simulated machine used for evaluation.

is `quick_sort` and it calls method `qsort` with three parameters `a`, the value `0`, and `n`. In Figure 3.1(b), I present the code compiled into assembly language with the GNU gcc compiler and `-O2` optimization flag. The line numbers with respect to the original source code (not shown in the figure) are on the left side of the assembly lines. First, line 435 calls the method `qsort` from the method `quick_sort`. The parameters are written in lines 431, 432, and 434 at different offsets in the `quick_sort`'s stack frame.[2] In `qsort`, the parameters are accessed in lines 332, 333, and 334 and are stored in registers. Note that the register used for addressing now is the `ebp` register. In line 327, the `ebp` register is assigned the value of the `esp` register, and the method `qsort` reserves its stack frame by subtracting 32 out of the stack pointer `esp` (stack grows downwards) (line 331). The `ebp` register is used by `qsort` to access the parameters. During the call from `quick_sort` to `qsort`, the old `ebp` and `eip` pointers are saved in the stack. The stack layout during the call and the frame semantics is shown in Figure 3.2. Therefore, the parameters of `qsort` accessed by that method are at offsets 8, 12, and 16 from the base pointer `ebp`. The instruction pointer prior to the call to `qsort` is located at offset 4 from `ebp`, and this value is used when the return instruction is executed by `qsort`.

Global state The global state, also referred to as the program state, is visible to all program entities, but may be semantically limited according to the specifications of a programming language, for example, in object-oriented languages. The global state is often accessed by a method by means of variables declared in the global name space or by means of pointers passed to a method as parameters. Typically, it is used for data structures that are needed by several parts of the program rather than temporarily by the method.

The global state is usually implemented in a memory structure known as the heap. The heap is a block of memory that is managed by the heap allocator often implemented in system libraries and operating systems. The heap allocator routines such as `malloc` and `free` (and the object oriented equivalents `new` and `delete`) operate on the heap memory (such as requesting or releasing memory blocks) as per the semantics of the heap, keeping it consistent. A program must explicitly use these routines to acquire memory as and when needed for its computation, and deallocate memory after use. For example, when memory blocks of a certain size are requested by the program, the allocator finds a free memory segment,

[2]This dissertation presents assembly code in AT&T syntax. Source operand appears first, followed by the destination operand of an instruction.

updates its book keeping structures, and returns the memory location(s) of the allocated memory back to the program. The heap allocator is responsible for allocating memory in a consistent manner to all programs running in the system, and achieves it in such a way that a minimum of memory is wasted.

The global program state modified by a method represents the side effects of a method. During a method's execution, the side effects are specified by the write set, and the global data accessed are specified by the read set. The nature of these sets is defined by the programming language. For example, many procedural languages usually allow methods to access and make changes to the entire program state. Therefore, after the execution of a method, its changes could be visible to the remainder of the program. In object-oriented languages, side effects of methods are often more limited as object-oriented programs tend to be more structured, written to access or make changes to an object with which they are associated with. Managed object-oriented languages such as Java and C# place even more restrictions on the memory that can be referenced by the program for guaranteeing the safety of data accessed (type safety) and maintaining compatibility across many architectures.

3.1.3 Semantics and calling conventions of a method

A method can be called from different parts of the program and the location from where it is called by the *callee* is referred to as the *call site*. The method may be called with one or more parameters, which are often used to specialize the computation in the method. A method may return a value back to the program (the caller) and the program continues its execution with the returned value, using it if needed.

Additional instruction(s) need to be placed in the program and the method to transfer control between the program and the method. The call instruction saves procedure linking information on the stack (specifically, the `eip` and `ebp` registers) and jumps to the method (program counter) specified with the destination operand. The return instruction, the last instruction encountered when executing the method, returns the control flow back to the program by popping the contents (`eip` and `ebp` pointers) saved on the stack. The return address is placed on the top of the stack, and the control is returned to the instruction that follows the call instruction in the program.

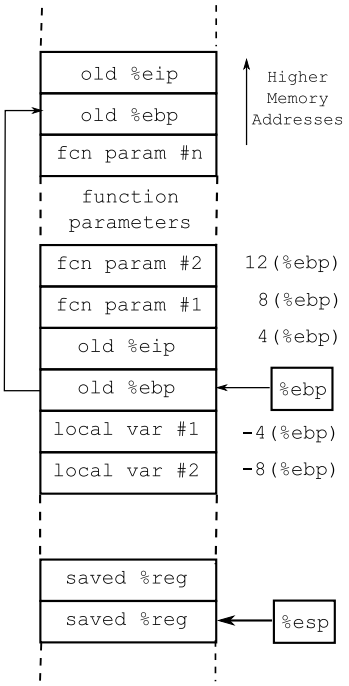


Figure 3.2: Stack layout. The active frame is guarded by the `esp` and `ebp` pointers. The frame space is used for storing local variables (shown local var #1 and #2) and saving registers during the method's execution. The parameters are saved by the caller on its stack frame before the call is made. After the call, the callee method accesses the parameters from the caller's stack with the `ebp` pointer. During the call, `ebp` and `eip` pointers are saved. They are restored (popped from the stack) when the called method returns.

Parameters

The parameters of a method, if any, need to be passed to it by the program before the computation can begin. Depending on the compilation model and the architecture of the hardware, different conventions are followed for this purpose. In general, there are two common means of passing arguments to a method. They are:

1. Parameter values are written to registers by the program, which are then accessed by the method. This is an efficient means of passing parameters, as memory reads and memory writes are not necessary. However, it is limited by the number of registers available for this purpose. For example, in the SPARC architecture, which has a register file window of 32 registers, the caller can write the parameters to eight "out" registers. On executing a call instruction to a method, the register window points to a new set of registers but with the overlap of "out" registers of the callee with the "in" registers of the

caller. Therefore, the method can access the parameters passed from the "in" registers. On the Intel x86 architecture, registers are rarely used for passing parameters because of their limited number. Recent versions of GNU C and C++ compilers for the Intel x86 architecture, perform register based parameter transfer as an optimization, often only within a compilation unit (a program source file).

2. The default approach in case enough registers are not available for passing all the parameters or if the register based convention is not followed, is to use the stack. The compiler generates code that writes the parameters to specific locations in the stack. The method can then access its parameters from the locations on the caller's stack frame. Figure 3.2 illustrates the mechanics of frames in the stack, the locations of the parameters, and the local variables in the stack.

Return value

Almost all architectures pass the return value of a method through a reserved register. For example, in the Intel x86 architecture, the return value is often stored in a reserved register such as the accumulator register (`eax`). Some compilers perform special optimizations within a compilation unit (a program source file), to use multiple registers to pass values between methods.

3.2 Role of methods in a program

Speculative threads in PD are composed of methods. To lay the motivation behind this choice, I first begin by describing the commonly prescribed steps for writing programs and composing methods.

Methods allow programmers to decompose a problem into several subtasks and enable them to write a complex and lengthy program. The decomposition process itself is a matter of programmer's choice and may require experience and skill acquired over time. Algorithms, programming languages, and software engineering textbooks recommend pursuing a set of steps to aid in the process of choosing methods. For example, in the book *Art of Programming* [105], Knuth elaborately describes his recommendations for developing methods and writing a program for a given problem. I briefly describe this next.

The whole program is divided into small number of pieces. Each of these pieces may be considered as methods, even though they may be called only once. These pieces can be

> successively refined into smaller and smaller parts, having correspondingly simpler jobs to do. Whenever some computational task arises that is already occurring elsewhere, the programmer may replace the occurrences with a call to a method, and perform the computation task in that method. All methods constructed can be studied again to determine if any need to be enlarged, for example, computation that is always performed just before or after the use of the method. Similarly, several methods may need to be merged if they are called only once.

With programming languages such as C++, the above recommendations are further supplemented with object-oriented principles, which state that a program should be comprised of individual components or "objects" that coexist and act on each other. Objects are composed of data and methods; methods access the data associated with the object, and any other specially mentioned objects, according to their semantics. (An important exception are static methods that are associated with a class.)

3.3 Role of methods in PD

The above description strongly suggests that methods are an apt choice for speculative execution as they provide an intuitive means for programmers to hold dependent computations. PD's goal is to create concurrency in a sequential program's execution by speculatively executing methods in the program in parallel. The execution of a method in a program is dependent upon another method's execution if a memory location in its read set is directly in the write set of the other method. Dependences between methods result in ordering between methods, and this partial ordering should determine the execution order. However, a compiler, inspite of any parallelism that may be available between methods, cannot automatically parallelize a program into multiple threads because of side effects (i.e., the write set) of methods that are not always identifiable or ambiguous. Since static analysis of a method's side effects (or, lack thereof) is not possible, the compiler assumes that all methods might have side effects and that a method could be dependent upon any prior method. This implies that the methods should be executed in the total order in which they are arranged by the programmer in the sequential program.

In PD, methods in a sequential program are "demultiplexed" from the total sequential order, and executed according to their data dependencies specified by the partial ordering between the methods. However, since the partial ordering is not guaranteed to be correct, and to ensure the sequential program order, the

methods are executed speculatively. The process of demultiplexing is to decouple a given method's call site and its execution. In sequential execution, the call site of a method represents the beginning of execution of that method, while the execution of a method in PD occurs speculatively on another processing core, usually after it is ready, which is expected to be well before the call site in the program.

3.4 Motivating examples

Chapter 1 provided an overview of PD based execution model, and this chapter, thus far, has discussed the reasoning behind the choice of methods for speculative execution. I now present some potential opportunities for PD based execution of methods in an application.

A software application is composed of many layers, each layer presents possibilities for PD based execution. Examples include: (i) library operations on file, I/O, network, and memory buffering, (ii) managed system utilities (depending on the programming language) such as garbage collection, (iii) application modules such as data structure packages, software templates, and (iv) the actual application. Many of these are equally amenable for software based parallelization. In fact, managed runtime system features such as garbage collectors are parallelized. Similarly, Java and C# packages that provide abstract data structures such as linked lists, hash tables, maps, queues, heaps, are often very efficient implementations, which support concurrent execution when invoked by multiple threads. However, as the integration of methods becomes tighter in an application, the process of creating software threads becomes more difficult because parallelism is neither easily identifiable nor readily available. The notion of speculation and speculative parallelization is an important feature that can create concurrency from programs not easily achievable by other means.

I begin with a simple example of a random number generator from `twolf` benchmark in Listing 3.1. The program data that the random number generator method `Yacm_random` accesses are `seed` and `randVarS`, locations that are never touched by the program. The memory references are therefore, clearly partitioned between the application and `Yacm_random` method. For correctness, it is necessary that memory operations to a given address are performed in sequential program order. Therefore, the `Yacm_random` method can speculatively execute, and provide (i.e., commit) the results of the execution when it is called by the program (as shown in Figure 3.3).

```
001 static int randVarS ;               /* random number */
002
003 #define A_RAND 16807L              /* good generator multiplier */
004 #define M_RAND 2147483647L               /* 2 * 31 - 1 */
005 #define Q_RAND 127773L                   /* m / a */
006 #define R_RAND 2836L                     /* m mod a */
007 #define ABS(value)   ( (value)>=0 ? (value) : -(value) )
008
009 /*
010   M_RANDD may have to be changed on different systems. On ultrix
011   it is as below.
012   #define M_RANDD (double) 1.0 / 2147483647.0
013 */
014 #define M_RANDD 4.65661287524579690000000000000000e-10
015
016 /* returns a random number in [0..2*31 - 1] */
017 int Yacm_random()
018 {
019     register int k_rand ;
020
021     k_rand = randVarS / Q_RAND ;
022     randVarS = A_RAND * (randVarS - k_rand * Q_RAND) - (k_rand * R_RAND) ;
023     if( randVarS < 0 ){
024   randVarS += M_RAND ;
025     }
026     return( randVarS ) ;
027
028 } /* end acm_random */
029
030 Yset_random_seed( seed )
031 int seed ;
032 {
033     seed = ABS(seed) ;
034     if( seed == 0 ){
035   seed++ ;
036     }
037     randVarS = seed ;
038 } /* end set_random_seed */
039
040
```

Listing 3.1: Speculative thread for the method `Yacm_random` in `twolf`

Library operations such as dynamic memory management, network, file buffer, input stream manipulation operations are all opportunities for PD. In Figure 3.4, I present an example of program performing memory allocations and deallocations in the program. Memory management is an integral part of any program as it enables using the heap memory space for storing program data of an application. The memory allocator itself has bookkeeping state to keep track of the heap memory that is separate from the application's program state. Execution of these methods rarely interferes with the program except for the parameters (which could create dependencies with the program and limit concurrency) and the value returned by the method. For example, `malloc`'s ordering with other memory allocator calls (such as `free`, `resize`, and

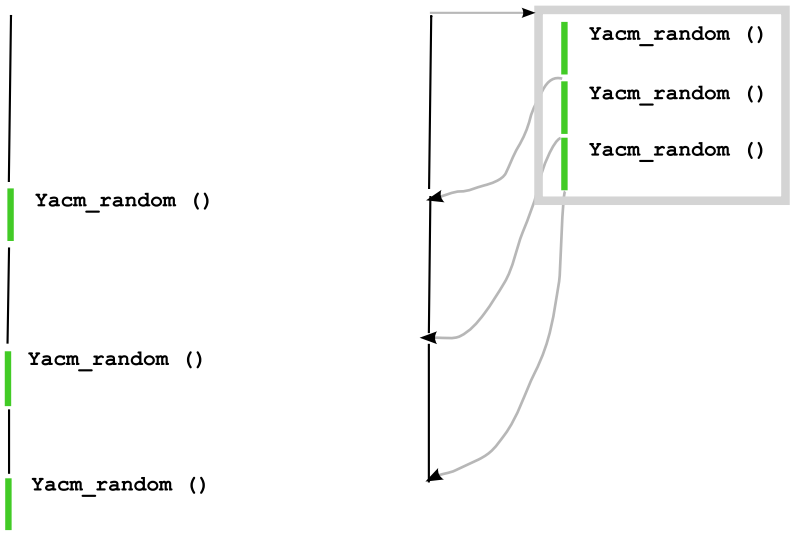


Figure 3.3: On the left is a sequential program with calls to `Yacm_random` in benchmark `twolf`. On the right is a PD based program with calls to `Yacm_random` separated and executed speculatively. The gray box denotes speculative execution.

other methods that modify bookkeeping structures of memory allocator) is the only requirement for correct execution.

Examples from SPEC CPU2000 integer suite I now provide some examples from integer programs in the SPEC CPU2000 suite. `gap` implements a language and library for computing in group theory. In the following example, I consider the method `NewBag` for PD. The method has over 500 call sites in the program, and contributes 17% of the total run time (when run with train inputs) – 7% from `NewBag`, and 10% from the `CollectGarb` method that is executed within `NewBag`. `NewBag` takes two parameters, the `type` of bag to be created and its `size`. The `type` parameter can take 30 possible types but is limited to very few depending on the method that calls `NewBag`. The `size` parameter can also be easily identified depending on the `type`. For example, the `size` is always four when `type = T_LIST`. In executions with train inputs, the method was invoked 6.8 million times and for 99% of the calls the parameters used were the same as that of a previous call. `NewBag` is likely to be a good candidate for PD based execution because of easily identifiable parameters, frequent invocations, and the task of mostly creating and initializing structures for the program, that are unlikely to conflict with the rest of the program.

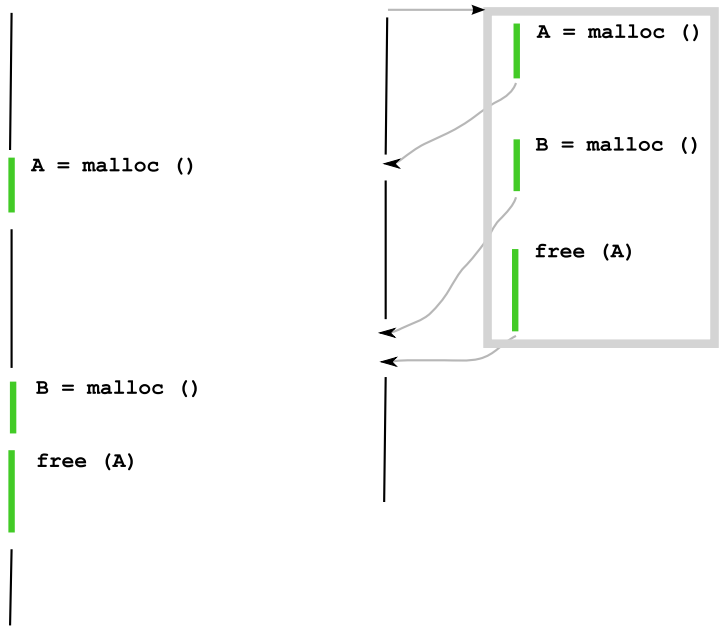


Figure 3.4: On the left is a sequential program with calls to the memory allocator methods. On the right is a PD based program with calls to `malloc` and `free` separated and executed speculatively.

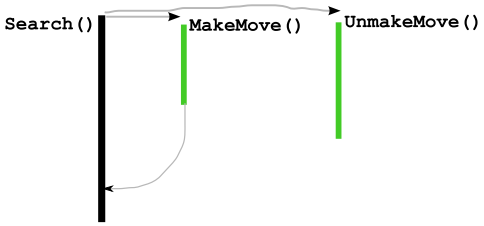
`mcf` spends 22% of its run time in methods `bea_compute_red_cost` and `bea_is_dual_infeasible` with the average number of instructions executed in the methods being 9 and 12, respectively (Listing 3.2). The methods are invoked around 93 million times each in an execution with train inputs (lines 016, 017, 024, and 025). `bea_compute_red_cost` computes the cost (an arithmetic expression on the benchmark's data structure arc) and feeds it to `bea_is_dual_infeasible`, which returns a boolean value based on an expression (lines 001 to 009). `bea_compute_red_cost` is dependent on the arcs and its potential which is updated by the method `refresh_potential` and sometimes by `update_tree`. With PD, the methods `bea_compute_red_cost` and `bea_is_dual_infeasible` can be triggered when an arc (or its potential) is updated; the two methods can concurrently execute for different updated arcs. Data analysis indicates that the distance (measured in the number of dynamic instructions executed) between `refresh_potential` and `update_tree`, to the call sites of these methods in the program, is three times more than the number of instructions the methods execute. Suitably forked, their speculative executions could be overlapped with the program, as illustrated in Figure 3.5(b).

```
001 cost_t bea_compute_red_cost (arc) {
002  return arc->cost - arc->tail->potential + arc->head->potential;
003 }
004
005
006 int bea_is_dual_infeasible ( arc, red_cost ) {
007  return (   (red_cost < 0 && arc->ident == AT_LOWER)
008             || (red_cost > 0 && arc->ident == AT_UPPER) )
009 }
010
011 arc_t *primal_bea_mpp( m, arcs, stop_arcs, red_cost_of_bea ) {
012  if( initialize ) { . . . } else {
013  else {
014   for( i = 2, next = 0; i <= B && i <= basket_size; i++ ) {
015    arc = perm[i]->a;
016    red_cost = bea_compute_red_cost( arc );
017    if ( bea_is_dual_infeasible ( arc, red_cost ) ) {
018      . . .
019    }
020  }
021  . . .
022  for( ; arc < stop_arcs; arc += nr_group ) {
023   if( arc->ident > BASIC ) {
024    red_cost = bea_compute_red_cost( arc );
025    if( bea_is_dual_infeasible( arc, red_cost ) ) {
026     . . .
027    }
028 }
030
```

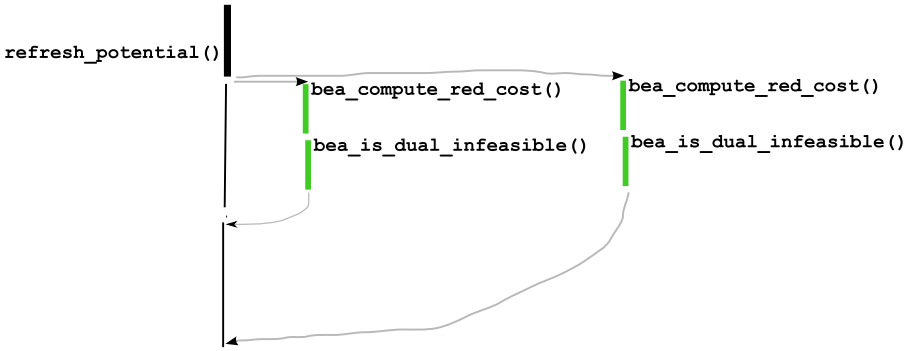
Listing 3.2: Speculative threads for methods `bea_compute_red_cost` and `bea_is_dual_infeasible` in `mcf`.

`vpr` is a FPGA placement and routing application. It spends 86% of its run time in operations on its heap data structures. 7% of its run time is from `alloc_heap_data` (Listing 3.3), a method to allocate memory in the heap structure. The program spends the rest of the 86% in `get_heap_head`, `expand_neighbours`, `node_to_heap`, and `add_to_heap` with its routing inputs. The application calls these methods to alter the value of elements in the heap, get the head of heap, and insert a new node onto the heap, respectively. I illustrate PD, in Figure 3.6(a), with the simple example of method `alloc_heap_data`, shown in line 003 in Listing 3.3. The method allocates a chunk of data if `heap_free_head` is not set; otherwise, it recycles the recently freed chunk of memory by the method `free_heap_data`, as shown in lines 013 to 021. Therefore, speculative execution of `alloc_heap_data` can be forked when the call site of `heap_free_head` or the call site of `alloc_heap_data` during the previous invocation are reached.

`crafty` is a computer chess program. Since it is an automated game playing application, it spends its execution time evaluating the chessboard, planning its moves, and eventually making them. A number of



(a) Speculative threads `MakeMove` and `UnmakeMove` in `crafty`



(b) Speculative threads `bea_compute_red_cost` and `bea_is_dual_infeasible` in `mcf`

Figure 3.5: Illustrations of PD based speculative executions of methods from benchmark programs, `crafty` and `mcf`

```
001
002 static void node_to_heap () {
003  hptr = alloc_heap_data ();
004  . . .
005  add_to_heap (hptr);
006 }
007
008 static void free_heap_data (hptr) {
009  hptr->u.next = heap_free_head;
010  heap_free_head = hptr;
011 }
012
013 static struct s_heap *alloc_heap_data (void) {
014  if (heap_free_head == NULL) {
015     /* No elements on the free list */
016     heap_free_head = my_malloc (NCHUNK * sizeof (struct s_heap));
017     . . .
018  }
019  temp_ptr = heap_free_head;
020  heap_free_head = heap_free_head->u.next;
021  return (temp_ptr);
022 }
025
```

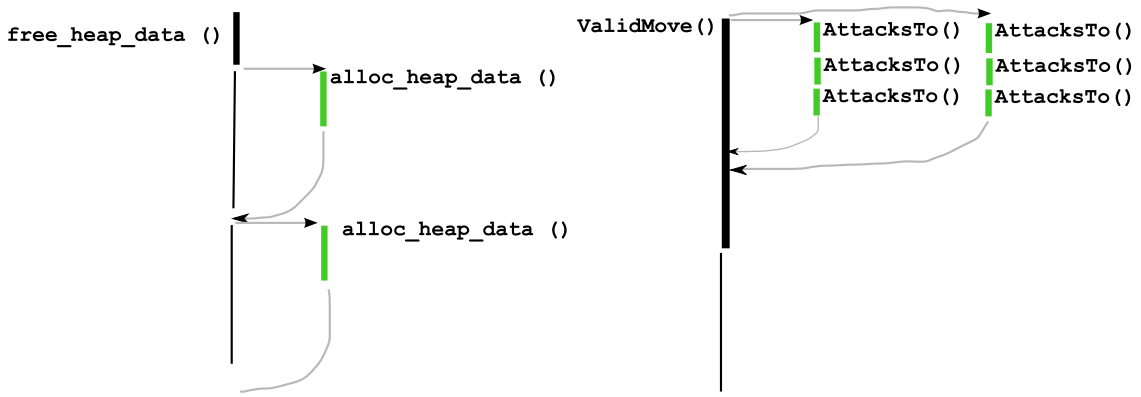
Listing 3.3: Speculative thread for method `alloc_heap_data` in benchmark `vpr`

```
001 BITBOARD AttacksTo(square) {
002    register BITBOARD attacks;
003    . . .
004    attacks=And(w_pawn_attacks[square],BlackPawns);
005    attacks=Or(attacks,And(b_pawn_attacks[square],WhitePawns));
006    attacks=Or(attacks,And(knight_attacks[square],Or(BlackKnights,
007                                                 WhiteKnights)));
008    attacks=Or(attacks,And(AttacksBishop(square),BishopsQueens));
009    attacks=Or(attacks,And(AttacksRook(square),RooksQueens));
010    attacks=Or(attacks,And(king_attacks[square],Or(BlackKing,
011                                                   WhiteKing)));   . . .  . . .
012    return(attacks);
013 }
014
015 int ValidMove (ply, wtm, move) {
016 . . .
017 . . .
018    case king:
019       if (abs(From(move)-To(move)) == 2) {
020  . . .   if ((!(WhiteCastle(ply)&2)) ||
021                   And(Occupied,Shiftr(mask_3,1)) ||
022                   And(AttacksTo(2),BlackPieces) ||
023                   And(AttacksTo(3),BlackPieces) ||
024                   And(AttacksTo(4),BlackPieces)) . . .
025          else if . . .
026                   And(Occupied,Shiftr(mask_2,5)) ||
027                   And(AttacksTo(4),BlackPieces) ||
028                   And(AttacksTo(5),BlackPieces) ||
029                   And(AttacksTo(6),BlackPieces)) . . .
030          . . .
031                   And(Occupied,Shiftr(mask_3,57)) ||
032                   And(AttacksTo(58),WhitePieces) ||
033                   And(AttacksTo(59),WhitePieces) ||
034                   And(AttacksTo(60),WhitePieces)) . . .
035          . . .
036                   And(Occupied,Shiftr(mask_2,61)) ||
037                   And(AttacksTo(60),WhitePieces) ||
038                   And(AttacksTo(61),WhitePieces) ||
039                   And(AttacksTo(62),WhitePieces)). . .
040     }
042
```

Listing 3.4: Speculative thread for method `AttacksTo` in benchmark `crafty`

methods can benefit from PD of which `AttacksTo` is a method where the application spends 6% of its execution time (Listing 3.4, call sites in lines 022 to 039). The `AttacksTo` method, used to produce a map of all squares that directly attack the specified square (lines 001 to 012), is called by several methods in the program with easily identifiable parameters; `ValidMove`, which is used to verify that a move is valid, is one of them. The speculative execution of `AttacksTo` can, therefore, be forked at the beginning of execution of `ValidMove` method (illustrated in Figure 3.5(a)).

Listing 3.5 presents another example in `crafty`. `Search` is a recursive method to implement the minimax search (lines 001 to 014). The method first checks its move, then calls the `MakeMove` method



(a) Speculative thread `alloc_heap_data` in `vpr` (b) Speculative thread `AttacksTo` in `crafty`

Figure 3.6: Illustrations of PD based speculative executions of methods from benchmark programs, `vpr` and `crafty`

(line 016), and then in some cases decides to undo by calling `UnmakeMove` method (line 026). This method accounts for 10% of the application's execution time and is called at several other points in the program; the example in Listing 3.5 is one call site. The parameters for `MakeMove` and `UnmakeMove`, as well as its dependencies, are satisfied at the beginning of the `Search` method. Suitably forked, the speculative executions can be overlapped with the program code executed in lines 002 to 015 (as illustrated in Figure 3.6(b)).

3.5 Program Demultiplexing framework

In this section, I provide a sketch of an implementation of PD that will be further discussed and evaluated in this dissertation. The PD framework is illustrated in Figure 3.7. Suppose, a method `A` at a given call site has been chosen for speculative execution. The *trigger*, a component of PD which, when "fired", i.e., its conditions satisfied, forks a speculative thread for the call site of `A`. In order to begin the speculative execution of `A`, the parameters, if any, are generated by speculatively executing another component of PD, the *handler*. The handler may then invoke `A` with these parameters depending on its control flow. The parameters are recorded, `A` speculatively executes, while the read and write sets of the speculative execution are monitored and gathered at the end of the execution. Several speculative executions may be ongoing on different processing cores. A speculative execution is held until it is used or invalidated. It is invalidated

```
001 int Search (alpha, beta, wtm, depth, ply, do_null) {
002   . . .
003   while ((current_phase[ply]=(in_check[ply]) ? NextEvasion(ply,wtm) :
004                                                NextMove(ply,wtm))) {
005     extensions=(threat) ? 0 : -INCREMENT_PLY;
006     if (Captured(current_move[ply]) && Captured(current_move[ply-1]) &&
007         . . .
008        )) {
009     if (Piece(current_move[ply])==pawn &&
010          ((wtm && To(current_move[ply])>H5 && TotalBlackPieces<16 &&
011            . . .
012            p_values[Captured(current_move[ply])+7]) {
013            . . .
014     }
015
016     MakeMove (ply, current_move[ply], wtm);
017       if (first_move) {
018         if (last[ply]-last[ply-1] == 1) {
019           extended_reason[ply]|=one_reply_extension;
020           one_reply_extensions_done++;
021           if (extensions < 0) extensions+=ONE_REPLY_TO_CHECK;
022         }
023         value=-ABSearch(-beta,-alpha,ChangeSide(wtm),
024                         depth+extensions,ply+1,DO_NULL);
025         if (abort_search) {
026           UnMakeMove(ply,current_move[ply],wtm);
028
```

Listing 3.5: Speculative threads for methods `MakeMove` and `UnMakeMove` in benchmark `crafty`

if the program commits to a location that is in the read set.[3] Outstanding speculative executions available are searched when a PD marked call site is reached by the program or by another speculative thread (called the requestor in this dissertation). Instead of executing the method then at the call site, the results of the execution are used if they have not already been invalidated, and if the parameters that were used for the execution match the ones at the call site. If a speculative execution of A is in progress, the requestor may decide to wait for the execution to complete, or abort the speculative thread and instead execute the method. For an illustration, see Figure 3.8. Next, I enumerate the implementation support that is needed, each of which will be discussed in more detail in the following sections.

1. Choosing methods and the respective call sites for performing PD based execution.

2. Generating handler(s) for a chosen call site so that the handler can set up the speculative thread and provide parameters for the speculative execution of the method that it may call.

[3]This description assumes that the write set is maintained at a byte granularity. However, a practical implementation can only collect the write set at a block level (for example, an entire cache line) and must therefore invalidate a speculative thread if a committed store is in a block that is in the read set and also in the write set.

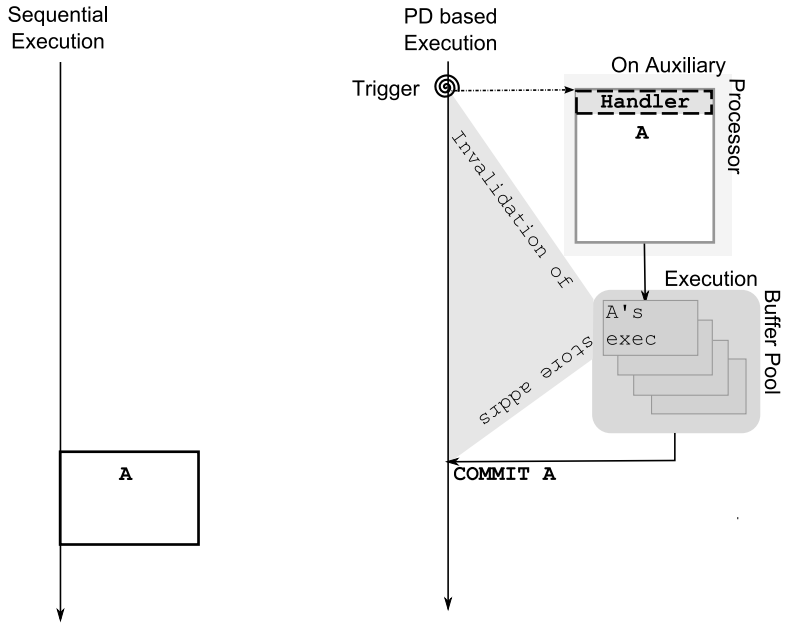


Figure 3.7: Program Demultiplexing framework. The framework illustrates PD based execution of a call site for method A in the program. On the left side is the sequential execution of a program with the call site of A and its execution shown. On the right is the PD based execution with the speculative execution of A when the corresponding trigger is fired, and the call site of A used to commit the speculative execution.

3. Generating trigger(s) for a chosen call site, which is usually set to fire when the dependencies of the handler and the method are satisfied.

4. Determining hardware support to perform speculative executions on processing cores, storing the results of speculative threads, ensuring their correctness, and finally committing the speculative threads.

3.5.1 Methods for PD

Suitable methods and their call sites need to be chosen for PD based execution. It is desirable to speculatively execute all methods in a program. However, this is not always achievable due to two reasons. First, the program may have limited parallelism between methods either because of the characteristics of the problem being solved or because of programming practices. In the latter case, a program may spend significant execution time in a few methods; speculative execution of large methods[4] increases the probability of violating dependencies resulting in wasted executions. Secondly, hardware resources needed for speculative

[4]The size of a method is measured by the number of dynamic instructions executed.

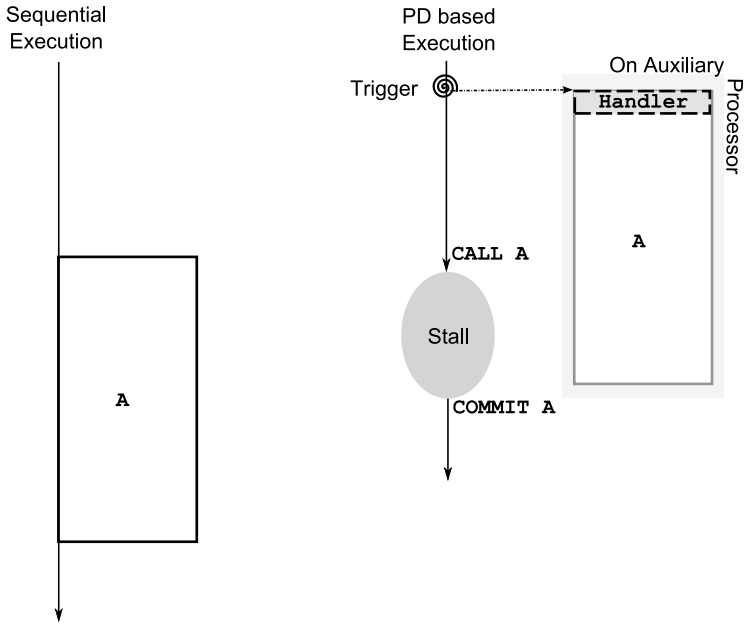


Figure 3.8: Program Demultiplexing illustration in which the program waits for the speculative thread to finish execution before the thread can be committed. The program may instead abort the speculative thread and execute the method at the call site.

execution, such as limited processing cores and cache resources, restricts the extent of speculation that may be performed.

3.5.2 Handler

In sequential execution, a method is invoked at a call site in the program. Any parameters needed for the method are communicated by the callee through the stack, registers, or other alternate means. With PD, a method is speculatively executed when the corresponding trigger fires. Therefore, some other means is necessary to provide the parameters. The handler consists of program code generated specifically for PD with means to generate the parameters and to call the method with the parameters. The speculative thread is said to have completed when the end of the handler is reached.

3.5.3 Trigger

A trigger is used to begin the speculative execution of the associated handler and is chosen to indicate the readiness of a speculative thread. The separation of the trigger site from the call site determines how much of

the speculative execution can be overlapped with the program and the extent of performance benefits. In this dissertation, the trigger is specified as conditions based on the program counters of committed instructions. For this reason, speculative threads can be forked only by the non-speculative program and not by other speculative threads.

3.5.4 Handling speculative threads

When a trigger fires, a free processing core is found, and the speculative execution of the handler begins. Depending on the code (and the control flow) in the handler, it may invoke the method, and begin the speculative execution of the method. Hardware support is used to perform speculative execution, which involves monitoring accesses made during the execution and ensuring that the architected state of the program is not modified. This prevents the execution of instructions that modify privileged state or instructions that may have unspecified side-effects in the speculative thread. During the speculative execution of a method, the read and write sets are tracked. Speculative threads do not communicate data values with each other. Finally, at the end of the execution, the read set, write set and data, have to be identified and retrieved, and stored along with the parameters and stack pointer used, for future use and to ensure that the speculative thread has not violated any dependencies. Additional hardware structures are needed to store the speculative threads, and to determine any conflicts. The program, or any other speculative thread, on reaching a PD marked call site, searches for available speculative executions for that call site. The results (write set and return value) are committed if requested by the program, or integrated into the speculative state of the requesting speculative thread.

3.6 Chapter summary

This chapter provided an overview on the PD based execution model. First, I presented background on the role of methods in programs, the semantics of compiling a method, and the motivation behind the choice of methods for speculative execution. I then presented several examples and opportunities for creating concurrency with PD in benchmark programs. Finally, I presented an implementation overview of PD, describing the software and hardware support needed for it. The next two chapters will focus on the implementation of PD.

CHAPTER 4

AN IMPLEMENTATION OF PROGRAM DEMULTIPLEXING

A key factor that must be considered in order to construct an implementation of PD is to decide the division of work between hardware and software. While some aspects of the implementation, such as profile generation and creation of handlers and triggers, are practical with software support, others such as speculative execution of methods may require hardware support because of lower overheads. In addition, several tradeoffs such as the complexity and cost of hardware implementation versus the benefits of such an implementation must be considered.

Implementations of PD can span several permutations of hardware and software support requirements, each with advantages and disadvantages. For example, a software only implementation of PD can be deployed with no hardware support, in comparison to a hardware-software implementation that requires non-trivial extensions to the current generation of multicore systems. In addition, a hardware-software implementation cannot always achieve the generality of software implementation. It can implement only specific common cases, and may leave the rest for the software to handle gracefully. On the other hand, the benefits achievable from software only approach may be limited due to the potentially prohibitive cost of software based speculative execution. A hardware-software implementation may be able to achieve greater benefits with reasonable implementation cost and complexity.

This dissertation describes a hardware-software implementation of PD that I have chosen. It was chosen not only because of its practicality, but also because of my lack of experience (and perhaps, the lack of evaluation tools) in a software-only approach. The experience and insights gained from the hardware-software implementation may be used to refine the PD execution model and applied to other implementations (such as the software-only approach). With that, this chapter covers the software support needed for the implementation, and is organized as follows. First, I discuss the assumptions made for the implementation. The profiling support needed for the implementation, and the steps to be carried out for generating a PD based application are discussed next. Finally, several examples from benchmark programs are provided.

4.1 Software support

I assume the software support for the implementation of PD is built on top of a binary rewriter or postprocessor. The toolset operates on precompiled binary of the application, and no part of the program code is altered, rewritten, or restructured for PD. A compiler may also be used, but this would require access to the source code which may not be available for some applications. A discussion of the pros and cons of this approach is deferred for later discussion (Section 6.2). Due to the binary-level implementation, the software support has to consider the compilation model of the program. For this purpose, I limit the dissertation to the compilation model used by the GNU gcc compiler for C programs. Due to simplicity of implementation, I also assume that all parameters are passed through the stack; a common case in the Intel x86 architecture due to limited availability of registers. With these assumptions, the following aspects of the implementation are performed by software. First, suitable methods and their call sites for PD based execution are identified. Then, handlers and triggers are generated for these identified call sites. Program analysis, both in the static form as performed by the compiler, and dynamic form by means of profile data (offline and/or online), are crucial for accomplishing the tasks.

4.2 Profile information

The software support for the implementation relies on profile information. In this section, I describe the different types of profile data used. I have broadly classified them as: (i) execution time profile, (ii) memory profile, (iii) branch profile, and (iv) call profile. Details of these profiles are discussed next.

Execution profile. The execution profile, collected at a method granularity, consists of a method's runtime, instruction count, execution cycles, and the different call sites that invoke the method. It is used to determine the run time contribution of a method's execution to the total execution time of a program, and for identifying suitable candidates for PD. A tuple in this profile has the method's program counter—the program counter of the first instruction in the method to which the control is transferred when the call instruction is executed by the program, the program counter of the call site, the number of instructions executed dynamically, and the execution time spent in the method. The tuple may also have debugging

information such as the method's name, the call site's line number, source file name and other debugging information. The list of entries in a tuple has the following:

Call Site PC
Method PC
Execution Time Cycles
Number of Instructions Executed
Call Site Info (source file name and line number)

Memory profile. The memory profile consists of a set of tuples, each tuple consisting of the program counter of a load or store instruction, the type of instruction, i.e., load or store instruction to heap or stack location, address of the read or write operation, and the number of bytes. It is collected for the entire program. A tuple in this profile would therefore have:

Program Counter
Instruction Type
Read or Write Address
Number of Bytes

The read set of a method's execution is aggregated from this profile which is then used to determine the trigger points for the call site. The profile is also used to establish memory dependence between instructions, especially for those that reference the stack as they are needed for handler generation.

Creating a profile with tuples collected for the run of the program is likely to consume significant storage space and offline processing time. On-the-fly memory profiling, where a profile is created in memory and processed for the desired information at run time of the program, is preferred. This may be easily achieved by instrumenting the program to create the profile, as well as to process the profile. A practical way of doing it is through libraries such as Pin [119] or DynamoRIO [26]. This dissertation uses a similar implementation. More details are presented in Chapter 6.

Branch profile. The branch profile, also collected for the entire program, consists of the program counter of a branch instruction, the number of times the branch is executed, and the number of times it is taken or not-taken during execution. The profile also carries the branch targets of indirect branches and calls encountered

during program's execution. These targets are used in place of the address computation operand in the indirect branch during generation of handlers. A tuple consists of:

Program Counter
Target Program Counter (for indirect branches)
Number of times Taken
Number of times Executed

This profile is used for handler generation to partly aid in determining whether a branch has to be included. It is also used to determine the chances of control flow mis-speculation between the trigger site and the call site.

Dynamic call graph. A dynamic call graph is a graph with nodes consisting of methods in a program, in which a method `M` is connected by the set of methods that call `M` during their execution. A dynamic call graph, unlike a static call graph, has only the set of methods that call a method during the execution observed. This information, along with execution time profile of methods, is used determine the call sites of a method for which handlers and triggers need to be generated for PD based execution. The dynamic call graph is also used along with the control flow graph for the construction of handler.

4.3 Static information

Besides relying on profile information, some key compilation structures are also needed for the implementation. These are:

Control Flow Graph. A control flow graph is a graph consisting of basic blocks and arcs representing flow of control between basic blocks. It is used for the generation of handlers and triggers.

Data Dependence Graph. The data dependence graph, specifically the post dominator tree, is required to determine if the call sites chosen for PD are control dependent on any branches between the trigger and call site, and for the generation of triggers.

As a side note, some profiling data used in the implementation may be substituted with compiler based static analysis. For example, memory dependence between instructions is required only for instructions that reference the stack. Stack references are usually not ambiguous and, therefore, the compiler may be used to determine dependent instructions without the need for memory profile information. In addition, precise analysis is not required from the compiler because of the speculative nature of PD. For example, memory profiling may be combined with 'may be' dependence analysis from static alias analysis to determine suitable triggers.

4.4 Overview of the implementation: the different phases

The rest of this chapter will describe the software support for the implementation of PD which comprises of three essential steps: (i) choosing methods for PD, (ii) generating handlers for the chosen call sites, and (iii) generating triggers for the chosen call sites. These three operations are not independent, but are inter-related, as will be evident in the following sections. For example, due to implementation issues with generation of handlers and triggers, or due to lack of performance benefits with speculative execution, some call sites may not be suitable for PD execution. They may be eliminated as candidates for future PD based execution runs. Similarly, determining the suitable handler for a call site will depend on the location of the trigger which, in turn, depends on the read sets of both the handler and the method. To accomplish this, steps (ii) and (iii), i.e., generation of handlers and triggers for the chosen call sites, are iterated twice (for convergence). The second iteration is used to perform some subtasks in a step that requires feedback from other steps, and vice versa. The details are given in the discussion of the generation of handlers and triggers in Sections 4.6 and 4.7.

4.5 Choosing methods for PD

Not all methods in an application are suitable for speculative execution. First, the execution profile is used to determine methods that contribute significantly to the program's total execution time. Then, the profile is used to determine the frequently executed call sites of the chosen candidate methods. The intention is not to consider call sites that are rarely invoked for speculative execution, as they may not contribute to performance improvements. The list of methods and call sites provides an initial list of methods for the rest

of the implementation. The set of call sites for PD based execution may be refined during each step in the implementation, and finally, when benefits of PD for the chosen call sites are available from execution runs (for example, through hardware performance counters). Some call sites may not be suitable for PD due to the following reasons:

1. The generated handlers and triggers for a given call site may not be able to achieve significant separation between the trigger site and call site, thus limiting concurrency between the speculative execution(s) and the program. This could be due to limitations in the handler implementation or due to hardware restrictions on speculative execution. It may also be because of the program code as written by the programmer and/or the execution model's inability to extract parallelism.

2. The probability of the program reaching the call site while at trigger site may not be high. This may result in speculative executions being discarded, thus wasting execution resources.

3. Hardware resource constraints, such as storage limitations in holding the write set of a speculative thread, and the presence of privileged system calls inside a method, may prevent speculative execution of a method.

4. The method called at a given call site may have limited parallelism due to dependencies with the rest of the program.

Several operations may need to be initiated by the hardware when a call site is reached. For example, the non-speculative program or a speculative thread may need to determine if there are any outstanding speculative threads when it reaches a call site. These operations may be performed on encountering every call instruction. Since it is unlikely that every method in the program (including user and system library functions) is going to be chosen for PD, a marker instruction `pdcall` is placed before the call site of a candidate method chosen for PD. The `pdcall` instruction has the upper and lower bound stack addresses of parameters (or equivalently, the number of bytes occupied by parameters) as operands. As shown in the following example, the `pdcall` instruction indicates that there may be speculative thread outstanding for the method `free_heap_data`, called by the following `call` instruction.

```
movl    %eax, (%esp)
pdcall  0x4
```

```
call    free_heap_data
```

The marker instructions are inserted statically to a binary and therefore, are conservative. The presence of `pdcall` indicates that the call site *may* have outstanding speculative threads. The hardware on executing a `pdcall` may trigger micro-code that initiates the search for an outstanding speculative thread.

4.6 Generating handlers

A handler is speculatively executed when its associated trigger is fired. The primary purpose of a handler for a given call site is to predict if the program's control flow would reach the call site, when the program is at the trigger site and, if so, invoke the speculative execution of a method with its parameters. The handler which is not considered as part of the program, therefore, decouples the the method's execution from its call site.

There are several ways for achieving the task of a handler. To evaluate the reachability of the control flow to the call site, the handler may evaluate branches that it predicts will also be executed by the program between the trigger and call site. Another approach would be to use a task predictor, similar to Multiscalar.

To provide parameters for the method's speculative execution, the following approaches may be taken: (i) value-based prediction, (ii) programmer-specified prediction, and (iii) computation-based prediction. All of the three means are predictive because the execution of a handler is speculative. Therefore, the values generated by the handler must be verified with the actual parameters provided at the call site. I next describe these three means.

Value-based prediction. Parameter values for a method's execution can sometimes be highly correlated with values used for invoking that method in previous executions. A value-based predictor may be used to collect the history of previous values and used to predict the parameters for the method's speculative execution.

Programmer-specified prediction. Since programmers have the best idea of the program code in the application, it may be effective to let programmers provide the handlers. Programmer-specified handler requires that the application writer provide alternate means of generating the parameters that can be executed

at the trigger site. This approach would require programmer support and possible language extensions to introduce handlers in the program.

Computation-based prediction. Instead of predicting the parameter values or requiring programmers to specify them, a computation-based predictor obtains the values by extracting some instructions "automatically" from the program and executing that code. The instructions extracted from the program deals only with the computation of the parameter values and, therefore, do not include any independent computation that may be present in the program. The common means of generating this type of a handler is by the process of backward slicing of dependent instructions that provide the parameter values at the call site.

Unlike value-based prediction, computation-based prediction involves recomputing parameters, i.e., performing computation that is also performed by the program, and may have higher overheads because of its execution. The backward slice to generate a predictor may also introduce additional dependencies with the program, which may limit the parallelism. However, value-based prediction is only effective for certain call sites with parameters that are repeatable across multiple invocations. Programmer-specified handlers are also beyond the scope of this work and require altering the program code of the benchmarks used for evaluation. I choose computation-based handlers for the implementation, as it is a generic approach applicable to any call site assuming handlers can be generated to deal with different programming constructs. In the following subsection, I discuss the process of generating handlers by means of backward slicing. I will present the heuristic choices, handling different programming constructs, optional optimizations, and integrating them to an application binary.

4.6.1 Backward slicing

Formally, the backward slice at a program point p is the program subset that may affect p. It is a commonly used technique for understanding, restructuring, and debugging programs, and has been extensively studied in the programming languages community [22, 191, 202, 203]. The goal of this work is to obtain handlers for call sites through backward slicing of the program; the algorithm is not optimized to minimize space and time overheads but merely designed for functionality.

```
001 main:
002   pushl %ebp
003   movl %esp, %ebp
004   subl $24, %esp
005   andl $-16, %esp
006   leal -4(%ebp), %eax
007   subl $16, %esp
008   movl %eax, 4(%esp)
009   movl $.LC1, (%esp)
010   call scanf
011   movl -4(%ebp), %eax
012   movl %eax, (%esp)
013   call m
014   leave
015   ret

016 m:
017   pushl %ebp
018   movl %esp, %ebp
019   subl $8, %esp
020   movl $.LC0, (%esp)
021   movl 8(%ebp), %eax
022   movl %eax, 4(%esp)
023   call printf
024   leave
025   ret
```

Listing 4.6: Assembly listing of a simple program that reads a value from the user and passes the value as a parameter to method `m`.

A collection of backward slices obtained from the program is used as a handler and acts as a computation-based predictor of parameter values. Each slice is used to recompute a parameter value, and the first step is to write the generated parameter value to the stack. (Note that the operation is performed backwards starting from the call site and, therefore, writing a parameter value to the stack is the first step.) The parameter values must be written at the same locations as the call site will write them, as the method will have to access the values from these locations irrespective of whether they are speculatively executed in PD or executed from a sequential program. In the example shown in Listing 4.6, this results in the inclusion of line 012 in the slice of the parameter for method `m` (which is called in line 013). Method `m` is shown to access the parameter value in line 021. The algorithm proceeds as follows.

> The source registers for each of the stack writes of parameter values form the elements of the live-in set for the backward slicing algorithm. For the example in Listing 4.6, it begins with register `eax`.
>
> For every register or condition code in the live-in set, the corresponding instruction(s) that

writes to the register or condition code is determined; they are included in the slice, along with any branches that the instruction(s) is/are control dependent upon.

For every instruction in the slice, depending on its type, the source register, scale, and index registers, if any, are included as elements in the live-in set. In case of a branch instruction, the condition code that is used to determine the decision of the branch is included in the live-in set. In case of a memory load instruction, the address, if in the stack, is included in the live-in set; otherwise, the algorithm terminates.

For every memory address in the live-in set, which can only be a stack location, the corresponding store instruction that writes to that memory address is determined; the instruction is included in the slice.

The goal is to terminate the backward slice of every live-in value with a load instruction to some heap location. If the computation terminates with a load to a stack location, the search continues for a store operation that writes to that address, and so on. The above set of heuristics will produce a handler that may have stack references in it, but will not have any computation involving the heap. This choice is made because stack references are short lived compared to the data in heap, and are usually associated with computations whose results are consumed immediately. In the second iteration, this generated handler is suitably adjusted according to the location of the trigger. This process is discussed later in this section.

Several aspects of the program code need to be dealt with during the backward slicing process. They include: (i) identifying memory handling instructions that are dependent, (ii) handling constant values, (iii) handling branches, (iii) handling other control flow structures such as loops, and (iv) handling inter-procedural dependencies. These issues are discussed next.

Establishing memory dependencies

It is necessary to establish memory dependences between load and store instructions so that they can be analyzed by the algorithm. Of particular importance to the slicing algorithm are instructions that reference the stack. Memory addresses for load and store instructions can be ambiguous at compile time, and available only during execution. The memory profile is used to determine memory dependence between load and store instructions. In some cases, this may not be accurate, as it is difficult to establish dependencies especially

```
001   movl 0x1, (%esp)
002   call m
```

Listing 4.7: Assembly listing of a simple program that calls method `m` with constant value `1`

with changing input sets and phase behavior in an application. However, in this implementation, since the handler generation algorithm deals only with dependencies between instructions that reference stack locations, ambiguity is not an issue. Stack references are rarely ambiguous because they often deal with local variables and the allocation is performed by the compiler, which can easily establish the dependencies.

Constants

The simplest case in the slice generation is handling constants. Listing 4.7 illustrates an example in which a live-in register's source is a constant value. The listing presents a call to method `m` with constant value `1`. The slice will terminate with the inclusion line 001.

Branches

Instructions in a backward slice may be control dependent on one or more branches in the program. This control dependence can be divided into two cases. The first case deals with the reachability of the call site from the trigger site. Consider Figure 4.1 that illustrates speculative execution for a call site. The trigger fires during the execution of the program, forking the speculative thread. Meanwhile, the program may execute several branches and may or may not reach the call site, eventually not using the speculative execution that was fired. To minimize the number of such wasted speculative threads, branches that the call site is control dependent upon, and instructions that compute those branches are included in the handler. For example, in Figure 4.2 the call site for `M` in block B3 is dependent on the branch evaluated in block B1 and, therefore, is included in the handler.

The second case is the control dependence of an instruction (besides the call) included in a slice. To illustrate this case, consider the example shown in Figure 4.3. The parameter value itself is dependent on the branch in block B1. Depending on the path taken, the value of `x` may be `n` or `m`. Both paths and the branch are therefore, included in the handler.

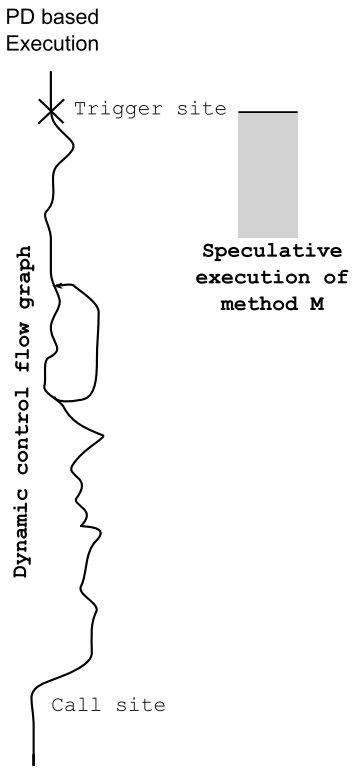
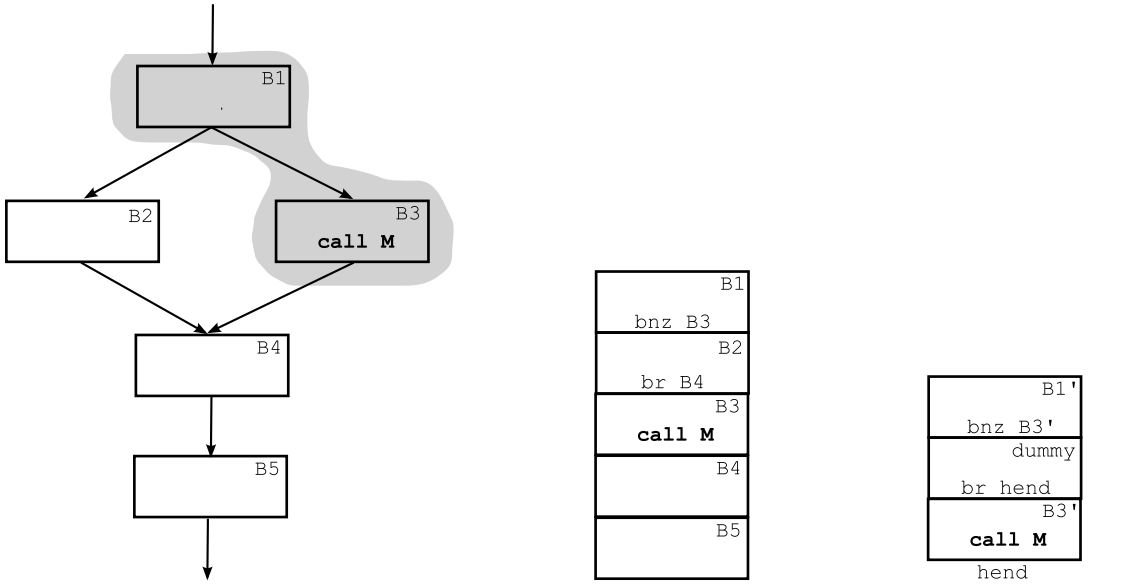


Figure 4.1: Reachability of the call site from the trigger site. Shown in the figure is the dynamic control graph (hence, not a straight line of committed instructions, but branches and loops taken in the program). The reachability of the call site will depend on the intermediate branches executed between the trigger site and the call site.

Loops

Branches may introduce loops in a program. A loop must be specially handled because instructions in a loop body may have cyclic dependencies. In this implementation, handlers do not have loops but instead, include instructions obtained by unrolling the loop in the program.

During the first iteration, a handler for a call site is generated without any knowledge of the trigger points for that call site. In the second iteration, with the knowledge of the location of trigger points, the code in the handler is adjusted. There are three possibilities for the location of the trigger points (refer to Figure 4.4 for the control flow graph). They may be: (i) located outside the loop when the method has no dependencies with the rest of loop body, (ii) located inside the loop when the method has loop carried dependencies, and finally, (iii) located inside the loop because of method's dependence on loop body. Of these, case (iii) is the simplest and requires no special handling. Case (i) can have many implementations, and the



(a) Control flow graph (b) Layout of the program (c) Layout of the handler

Figure 4.2: A call site for method `M` in basic block B3 that is control dependent on the branch in block B1. The control flow graph, the layout of the program, and the layout of the handler in the binary are shown in the figure.

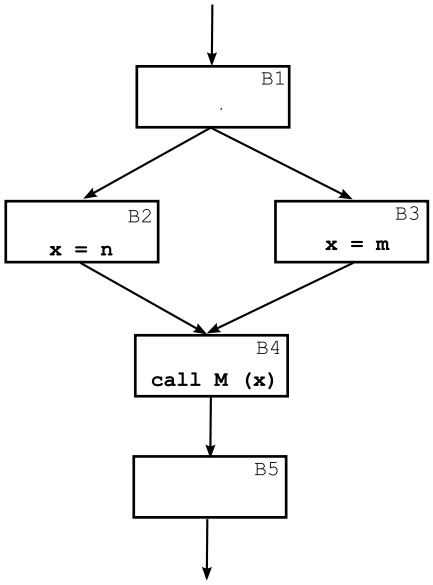


Figure 4.3: A call site for method `M` that takes one parameter `x`. The value of `x` is dependent on the branch in block `B1`.

discussion is postponed to the end. Case (ii) may result in a handler that steps forward several iterations and then speculatively calls the method. Note that due to the algorithm's restriction on not including any store

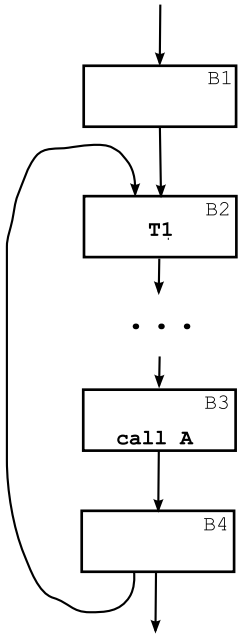


Figure 4.4: Illustration of handler generation for loops. The call site for method A is located basic block B3. The trigger point for the call site is in basic block B2.

instructions to the heap, unrolling is not performed if the iterator is a heap location (the same is true if the iterator is register allocated).

To further explain case (i), consider a simple example of random number generator being invoked inside a loop, which is independent of the computation in the loop body. The best means to capture this parallelism is to speculatively execute several calls to the method in parallel. This may be done by placing several triggers, explicitly or implicitly, by means of a dummy loop, that forks speculative executions of the method. An alternate way is to include the loop in the handler and substitute the call instruction with an "asynchronous call" to the method, which invokes the call and continues with the rest of the code. This implementation takes a simpler approach and does not execute the method in parallel, but invokes in sequential order.

Interprocedural

Two types of interprocedural dependencies may be introduced during the generation of a slice. In the first case, the producer for an element in the live-in set may be the return value of another method invoked by the caller. In the example shown in Listing 4.8, method `m` is called with parameter `x`, which is the return value

```
int g () {
//  do some computation

  return 1;
}

void f () {

  x = g ();

   ...
  m (x);
}

void m (int x) {
 ...
}
```

Listing 4.8: Example code for interprocedural dependencies when generating handler for method `m`. The parameter value `x` is produced by another method `g`.

```
001 m:
002    ...
003    movl 8(%ebp), %eax
004    ...
005    ret
006 g:
007    ...
008    movl $1, %eax
009    ret
010 f:
011    ...
016    call g
017    movl %eax, (%esp)
018    call m
019    ...
```

Listing 4.9: Example handler for code shown in Figure 4.8. The PD call site is in line 018. `g` (which will be copied during relayout) returns the value 1 which is provided as the parameter for `m`.

of another method `g`, called by the caller method `f`. Slicing may proceed to include instructions in method `g`. In the given example, the return value is not dependent with the rest of `g`. The instructions that are part of the handler are shown in Listing 4.9. (The call targets must be adjusted and this process is discussed later.) If slicing of the dependence is not possible, the algorithm terminates at the point of return value dependence and the trigger site for the execution will be set at this point in the program.

In the second case, the producer for an element in the live-in set may be a parameter value. In the example shown in Listing 4.10, the handler is generated for method `m`, called by the caller method `g`. The caller method `g` is also called with the same parameter `x` by another method `h`. Slicing may further extend to all the call sites for the caller method by replicating the handler code for every dynamic call site, and

```
void m (int x) {

//   compute with x
  ... = x ...
}

void g (int x) {

//  ... some computation

  m (x);
}

void h () {
  x =

//   ...
  g (x);
}
```

Listing 4.10: Example code for interprocedural dependencies when generating handler for method `m`. Method `h` calls method `g` with parameter `x` which is passed on to method `m`.

continuing the operation for each one of them. This would produce multiple handlers for the given PD call site. Otherwise, slicing terminates at the head of the caller method and the trigger is set to begin speculative execution from this point.

4.6.2 Termination

An important decision in the generation of the backward slice is determining when to terminate the operation. The factors that need to be considered in the heuristic for termination are the length of the handler, in terms of the number of dynamic instructions, its contribution to the speculative execution, and the separation that can be achieved between the speculative execution and the call site of the method in program. More computation in the handler increases the number of instructions in it. This increases the overheads of speculative execution, but may also further the separation of trigger site and call site and improve the extent of parallelism.

One other aspect that must be considered during slicing is to ensure that a slice does not extend beyond the trigger point for a call site. Figure 4.5 illustrates an example in which a method's read set consists of memory location `X`. Location `X` is written to by the program along the path of the backward slice. After the trigger points for a method's call site are identified, it is clear that the method cannot begin execution before

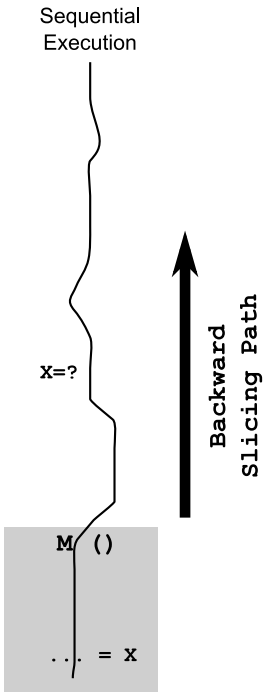


Figure 4.5: Dynamic control flow graph of a program is shown and backward slicing process for a call site of method M is marked. Method M during its execution reads from heap location X, which is written outside of the method by the program and in the path of the handler. The trigger point for the call site of M can occur no earlier than the assignment to X and the slicing process may be terminated here.

the write to X. Therefore, in the second iteration, the handler generated in the first iteration is trimmed to ensure that head of all the slices in the handler do not extend past the corresponding trigger points.

4.6.3 Optimizations

Numerous opportunities exist for choosing the code for the handler; they were briefly discussed earlier in this chapter. In this subsection, I discuss some optimizations to the handler. (None of these are implemented in this dissertation.)

Multiple call sites. The compute-based handler discussed in this dissertation may have high execution overheads, especially if considerable separation (in terms of execution cycles) is needed between the trigger site and the call site to overlap the execution of a speculative thread. The overheads of executing a handler may be amortized if multiple call sites that share similar computation due to their proximity in program use the same handler. For example, consider a hammock with two call sites for method M as shown in Figure

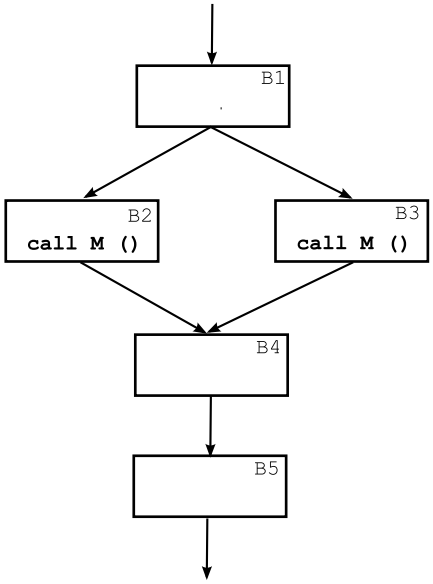


Figure 4.6: Method `M` invoked in both `if` and `else` part of a condition. The implementation generates two handlers for the two call sites.

4.6. In the implementation discussed in this dissertation, two different handlers are generated for the two call sites in blocks B2 and B3. Both evaluate the branch in B1 to determine whether to call `M`. Instead, the two handlers may be combined and the branch speculatively evaluated only once.

Infrequent paths. The handler may be speculatively optimized, such as by elimination of infrequently executed branches and substitution of dependent chains of instructions with the result. Several other optimizations, proposed by other speculative parallelization proposals [61, 152, 210], can also be applied to handlers. In general, these optimizations may affect the accuracy of correctly computing the parameters.

Global data dependencies. The scope of handlers may be expanded and can be used to eliminate some data dependencies (besides parameters) between the method and the program. For example, as shown in Listing 4.11, assume method `m` reads from variable `g`, a global variable written before the method is called by the program. By expanding the scope of handler to include writes to the heap, and by capturing backward slices of not just the parameters but also heap memory locations accessed by the method, the extent for parallelism for PD may be further improved.

```
001 void f () {
002
003   g = val;
004   ...
005   m ();
006 }
007
008 void m () {
009   int temp;
010
011   ...
012   temp = g;
013   ...
014
015 }
018
```

Listing 4.11: Global variable `g` is written before method `m` is called. Method `m` accesses the variable `g` during its execution.

4.6.4 Incorporating handlers into program

Once the handlers are generated, they are incorporated into the binary. The program counters of the head instruction of the handlers are associated with the triggers. The handlers and triggers are laid out in separate segments in the binary (see Figure 4.7). For comparison of parameters generated by the handler and at the call site, `pdcall` instructions along with parameter bounds are inserted before the the call to the method in the handler. Instructions that do not alter the control flow are incorporated in the handler without any alterations. Branch targets are altered to jump to locations within the handler and not back to the program. An example is shown in Figure 4.2, in the form of control flow graph. The branch in block B1 is included in the handler and, therefore, its target is changed from B3 to B3'. If the branch is not taken, the handler falls through to a dummy block (the handler does not have any instructions for the corresponding block B2), and terminates.

Another example in Listing 4.12 illustrates the handler for method `m` in Listing 4.8. Since the handler includes instructions from another method `g`, a dummy method `g-t` is created in the handler. The code in `g-t` is the subset of code from `g` in the program. Note that all stack pointer manipulating instructions (lines `002` to `005`) are also included from the method `g` to ensure that any stack references in `g-t` (all of which obtained from `g`) are not altered. It is necessary that the handler execute the call to `g-t` ensure that the `ret` instruction included in `g-t` is matched. A handler is terminated with a `hend` instruction.

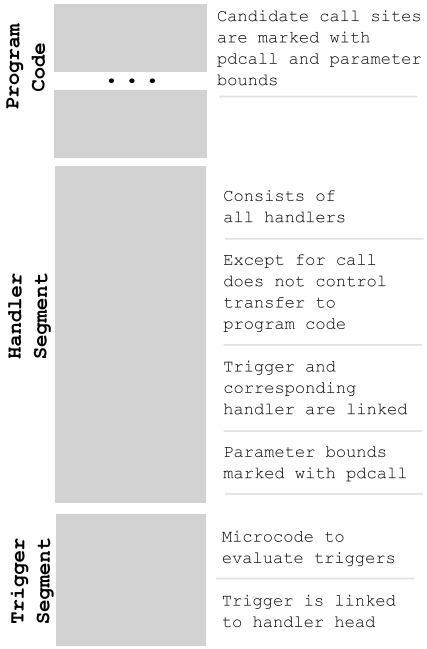


Figure 4.7: Layout of the program with triggers and handlers. Call sites chosen for PD are prepended with `pdcall` instructions in the program code. The handler segment consists of all handlers laid out. Control is transferred to the program only by a call instruction that initiates the speculative execution of a method. The trigger segment has the evaluate and register portions of all the triggers. A trigger is linked to its corresponding handler.

```
001 g-t:
002  pushl %ebp
003  movl %esp, %ebp
004  subl $24, %esp
005  leave
006  movl $1, %eax
007  ret

008 handler_head:
009  call g-t
010  movl %eax, (%esp)
011  call m
```

Listing 4.12: Layout of the handler in the presence of a call for the code presented in Listing 4.8. A dummy call `g-t` returns the value `1` which is saved to the stack, and is passed to the speculative execution of `m`. Note that all instructions that manipulate the stack pointer in `g` are included `g-t` to ensure that the references are to the same location.

4.7 Generating triggers

The notion of a trigger allows initiating the speculative execution of a method at a point in the program different from the call site. The separation of the call and trigger sites provides the capability to break from sequential ordering, execute one or more methods out of program order concurrently, and later use the call sites to determine when the speculative executions of methods, if valid, should be committed. The executions have to be speculative and committed in program order to provide the notion of sequential execution, and because triggers are only indicative of when speculative executions may begin, not definitive. If this had been the case, the compiler could have parallelized the program statically.

A trigger in PD is an expression composed of a set of conditions, formally, predicates. A predicate is constructed from program counter of an instruction with equality and inequality operators. A set of conditions may be logically operated on by `and` operators to create the expression for a trigger. The expression, when evaluated to true, indicates that the trigger has "fired". An example of a simple trigger is,

`(PC = 0x1234578)`

In the example, the trigger fires when instruction at program counter 0x12345678 commits.

`(PC = 0x8495423 and previous PCs != 0x8593251)`

In this example, the trigger fires when instruction with PC 0x8495423 commits, and the program did not commit instruction with program counter 0x8593251 during its execution. (The instructions that are used to match the != condition is dependent on the software micro-code generated for the trigger. This is further described in Section 4.7.4.) One or more triggers are created for a call site of a method in the program that is a candidate for PD based execution. A trigger is used to indicate the "readiness" of a method. The expression of a trigger is constructed to fire when the associated handler, and subsequently the method can begin speculative execution and usually not violate any data dependencies. Choosing the expression for a trigger is one of the key factors that determines the extent of useful speculation that may be performed for the corresponding call site.

Conceptually, triggers for a call site can be derived from the summary, i.e., data requirements of the method, during its executions. Method summaries are commonly used in scientific program parallelization by compilers to analyze and achieve parallel execution. Because PD has to deal with programs with unstructured and ambiguous memory references, summaries cannot be easily generated by a compiler

through static analysis. For this, the implementation relies on dynamic profile information. Even though ambiguous memory references, i.e., pointer references in a program, can access any part of the program state, in reality, programs exhibit bounds in addresses referenced. This aspect either creates stable read sets or stable producer code across several executions of a method. It is also likely to have methods with unrealizable triggers in this implementation.

The steps involved in the generation of trigger for a given call site of a candidate method chosen for PD will be covered in the rest of this section. This process must be repeated for every call site chosen for speculative execution in the program.

4.7.1 Identifying trigger points

The first step in determining the trigger for a call site is to determine what the trigger points are. A trigger point, specified for every execution of a given method's call site, represents the point or instance in the sequential execution of a program when the read set (i.e, data required) for the execution of the method and its handler is available. If the speculative execution begins at this point, the method will be able to execute speculatively without being invalidated for dependence violations. The handler is also included in the read set, and in determining the trigger point, because a method cannot speculatively execute without it.

To determine the trigger point for a given call site's execution, the memory profile is used to collect the read set of the execution of the method and its handler. This summarizes the method's data requirements for speculative execution. The read set is the set of all memory locations that are accessed during execution and are not provided by the execution. Therefore, if in an execution, location X is written first, and then accessed later, that location is not part of the read set. With similar reasoning, the read set will never have a memory location that belongs to the local stack frame. On the other hand, the read set may have references to the caller's stack, due to: (i) passing of a stack pointer as a parameter, (ii) passing a parameter by reference to the method's execution or, (iii) handler's dependencies. The dependence to the caller's stack in cases (i) and (ii) are unavoidable in this implementation, while case (iii) depends on the generated handler.

Having obtained the read set, the memory profile is again used to determine when it is available during the program's execution. The last write that completes the read set's requirements is the *trigger point* for that execution. Figure 4.8 reviews the steps discussed for the generic case when the read set has no references

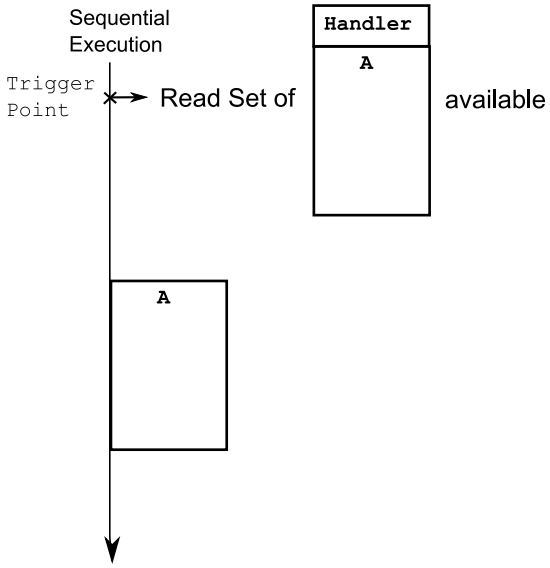


Figure 4.8: Identifying trigger point for an execution of method A. The trigger point is found by collecting the read set of the execution of the method and its handler and determining when the read set is ready during the sequential execution.

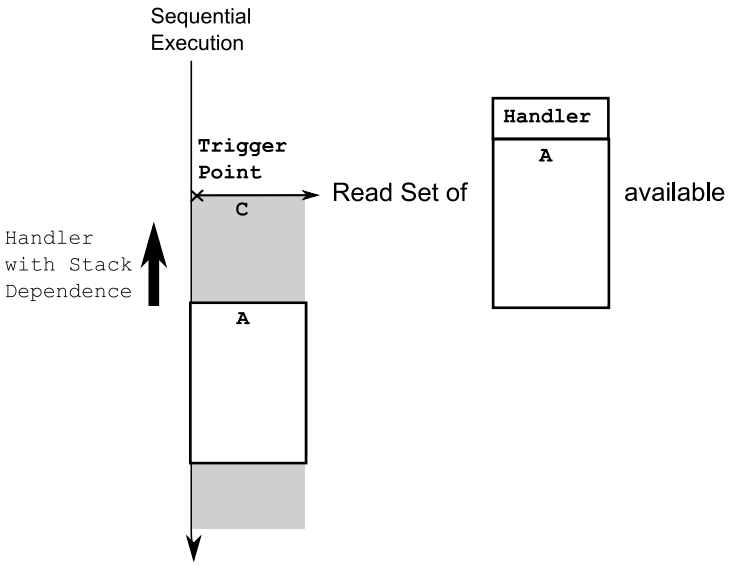


Figure 4.9: Identifying trigger point for an execution of method A which, along with its handler, is dependent on a value in the caller's stack frame (caller method C shaded gray). The trigger point cannot be any earlier than the creation of that value in the stack frame.

to the caller's stack. The trigger point for the execution of A and its corresponding handler is identified and shown in the figure. Figure 4.9 is a similar example, but illustrates trigger point in the case of stack dependence in the read set. Assume that the stack reference is to the caller's stack (the caller is shown with gray shading in the figure), the trigger point cannot be placed earlier than the creation of the dependent value in that stack frame.

4.7.2 Collecting trigger points

From the program's execution, several trigger points are collected for the chosen call site. The goal is not to observe all executions of that call site, but to achieve good coverage by collecting a large set of trigger points. The trigger points may not always be unique due to two reasons. First, executions of the call site may have varying read set requirements depending on the parameters passed and the program state accessed. Second, the program may exercise different control flow paths to reach the call site. The set of trigger points will depend on the application, its characteristics, and its behavior to different input sets.

4.7.3 Specifying triggers

The final step is to take the set of trigger points from the previous step and produce triggers for the call site. The goal is to ensure that the trigger for a call site fires in a timely fashion, i.e., speculatively execute a method before its call site, but also without violating any dependencies of the program. The burden of choosing call sites for PD and determining suitable triggers for useful speculation is on the software implementation.

One of the first requirements that will determine whether a candidate call site can be chosen for PD is the cardinality of the trigger points set. Smaller cardinality implies lesser hardware requirements and easier implementation, and is practical when a method has a stable read set due to control flow convergence and/or stability in program code that produces the read set for the method's execution. Experimental results for the benchmark programs evaluated in this dissertation indicate that methods that are small to medium (relative to the benchmark program's largest method, measured in terms of dynamic instructions executed), and tend to have a small set of trigger points (under three). In addition, the cardinality can be significantly reduced with optimizations that are not studied in this dissertation, but some of which described later.

Another requirement that determines the suitability of a call site for PD is the path taken by the program after the trigger has fired. A low probability of the non-speculative program reaching the call site implies wasted speculative executions.

The trigger points are converted into triggers for a call site that passes the above tests. Before this process, trigger points must be adjusted. First, to simplify the implementation, the program counter of a trigger point is transformed into the program counter of the last instruction in its corresponding basic block. Second, the program counter of a trigger point that is inside another method chosen for PD based execution is moved out of that method and replaced with the program counter immediately after the call site. This is performed because speculative threads can only be forked from the non-speculative program in the current implementation. To maximize benefits, trigger points that are only inside methods that are definitely going to speculatively executed must be promoted outside of the method. This may be achieved by conservatively promoting the trigger points outside of the method, and then based on the feedback from a PD execution, adjusting if necessary.

After the transformations, every element in the trigger point set is converted to a trigger. The trigger for a given trigger point is the 'logical and' of its equality, and the negation of all other trigger points which dominate it. This ensures that multiple triggers do not fire for a given call site and fork many speculative threads. Two examples of this issue are shown in Figure 4.10 and Figure 4.11. Figure 4.10 shows a call graph; methods `A` and `B` call `C`, which calls `M`. The trigger point for `M` is T1 when the program takes the path `A` $\rightarrow$ `C` $\rightarrow$ `M`, and T2 when the path is `B` $\rightarrow$ `C` $\rightarrow$ `M`. Both T1 and T2 should not fire when the program takes the path `A` $\rightarrow$ `C` $\rightarrow$ `M`. Similarly, Figure 4.11 shows trigger points T1 and T2 located in basic blocks B3 and B4. Again, both T1 and T2 should not fire when the program executes basic-blocks B1, B3, B4, and B5. Only T2 should fire; otherwise, the speculative thread forked from T1 is unused.

4.7.4 Incorporating triggers in a program

Having generated the triggers for many call sites, the next step is to specify these triggers and incorporate them in a PD based binary. There are two approaches for performing this, the static and the dynamic approach. I discuss these options next.

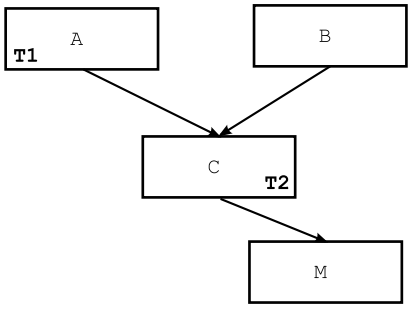


Figure 4.10: Dynamic call graph, in which method `A` and `B` call `C`, which calls `M`. T1 represents the trigger point in the path of `A` to `C` to `M`, and T2 represents the trigger point of `B` to `C` to `M`.

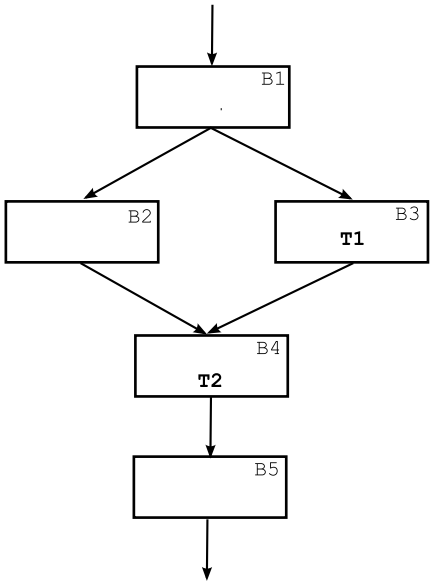


Figure 4.11: Illustration of two trigger points `T1` and `T2` in basic blocks B3 and B4 for a call site (not shown).

Static approach

In the static approach, as the name indicates, the triggers are specified statically, i.e., placed directly into the binary. During program's execution, when an instruction `pdfork handler_pc` is committed, it indicates that a speculative thread has to be forked with execution beginning at program counter `handler_pc`. The trigger is specified by the compiler, but the location may be altered if the application is executed on a managed runtime system such as Java [116], C# VMs, or LLVM [1, 111] or a dynamic code modification system such as DynamoRIO [14, 26]. ISA extension to implement the `pdfork` instruction is the only hardware support that is needed.

Since the static approach specifies a trigger by incorporating it directly into the application, it is most useful for a call site that has only one trigger site. A call site that has more than one trigger points requires the support for generalized expressions constructed by the process described in the previous subsection. To support this, the dynamic approach, described next, is implemented in this dissertation.

Dynamic approach

The dynamic approach is a generic way to support and evaluate triggers. Any form of expression may be specified for a trigger and evaluated with hardware support.

Unlike the static approach that incorporates `pdfork` instructions into a PD based program binary, in this approach, triggers are implemented as micro-code snippets provided by software and evaluated with hardware support. Micro-code of the triggers in a program are laid out in the binary in a separate segment as shown in Figure 4.7. The micro-code is a representation of the trigger's expression in a form that can be executed by the hardware for evaluation. The result of the evaluation will determine if the trigger has been fired.

To support this approach, the fundamental requirements from hardware are: (i) storage for holding the results of the predicates, called trigger condition code registers, (ii) logical operators and access/modify instructions extended or new instructions, to operate on the trigger condition code registers, and (iii) execution resources to evaluate the micro-code by means of execution.

The storage for results of predicates are provided with trigger condition code registers, bit-level storage similar to condition code registers in Intel x86 architecture [92]. The first piece of the micro-code snippet for

```
tsetpc X t0
tsetpc Y t1
```

Listing 4.13: `tsetpc` registers a program counter (first operand) and a trigger condition code register with the hardware. The register is set when the program commits the instruction at the specified program counter.

```
    cmpb    $1, %t1
    je      L1
    testb   %t0, %t0
    jne     L2
L1:
    tend
L2:
    xor     %t0, %t0
    xor     %t1, %t1
    pdfork  handler_pc
    jmp     L1
```

Listing 4.14: evaluate portion for evaluating a trigger. The code checks if `t0` is 1, and `t1` is 0. If true, the trigger condition code registers are reset, and a speculative thread is forked to begin from program counter `handler_pc`. If false, the trigger ends with `tend`.

a trigger is the *register* portion. It deals with registering the program counters of interest with the hardware, so that the corresponding trigger condition code register may be set for further use. For example, assume that the trigger is `PC = X and previous PCs != Y`; the program counters of interest are `X` and `Y` used in the two predicates. Each of these are registered with the hardware using `tsetpc` instructions (shown in Listing 4.13), that instruct the hardware to set condition code `t1` when program counter `X` is committed, and `t2` when `Y` is committed. The working of these instructions is similar to watchpoints implemented in hardware [92, 94, 180] and used to interrupt the execution of the program when it reaches a specified point (for example, to transfer control to a debugger so that the programmer can examine the state of the program and debug).

The second piece of the micro-code for a trigger is the *evaluate* portion, which evaluates the trigger when one of the associated trigger condition code registers are set. For the example, the evaluate portion must determine if `t0` is one and `t1` is zero, and the micro-code is shown in Listing 4.14. If the result of the evaluation is true, a speculative thread is forked which begins execution from the corresponding handler's program counter. Additionally, the evaluate portion must deal with resetting trigger condition code registers, which is usually after the trigger's condition is satisfied.

Similar to the static approach, a managed application will be able to remove triggers for a call site if

the speculative executions are not beneficial, alter them to reduce mis-speculations or increase parallelism, or insert new triggers for additional call sites. These features can also help an implementation take into account phase changes which may alter the trigger points and/or add new methods to the hot path, or limit the hardware resource requirements by removing some triggers and inserting new optimized ones. All of these may be performed during the execution of a PD based program based on the feedback from the speculative threads, similar to hot path optimizations commonly performed in Java based managed applications [8, 9, 11, 14, 30, 44].

4.7.5 Optimizations

The implementation thus far described is a first-cut means for deriving triggers for PD. Optimizations can improve the efficiency of the execution model by minimizing the number of trigger points for a given call site, thus allowing a static approach for the implementation or minimizing the hardware support that may be needed for the dynamic approach.

One form of optimization is to minimize the number of triggers by identifying two or more call sites (same or different methods) that have the same or similar trigger expressions. Such call sites may share the same trigger which, when fired would invoke speculative executions of multiple handlers.

The other form of optimization is to minimize the number of trigger points and optimize the predicates in a trigger. This may be achieved through several means. One or more trigger points for a call site may be eliminated and replaced with a trigger point in the control flow convergent point. The extent of parallel execution, and in turn performance benefits, may be sacrificed with this optimization. Figure 4.11 provides an example in which a call site has two trigger points `T1` and `T2` in the program. Instead of specifying two triggers for the call site's two trigger points, the trigger can be simplified by just specifying a single trigger at the control flow convergent point block `B4`. Note that `T1` is altered and is no longer the earliest point in program when the read set of the method and handler is available.

Another way to minimize trigger points is to use other forms of instruction attributes such as memory read and write addresses in predicates used in the expression of a trigger. In the illustration shown in Figure 4.12, a call site has two trigger points `T1` and `T2` at instructions that are modifying the same memory location `X`. These trigger points can be alternately specified by a single predicate based on memory write

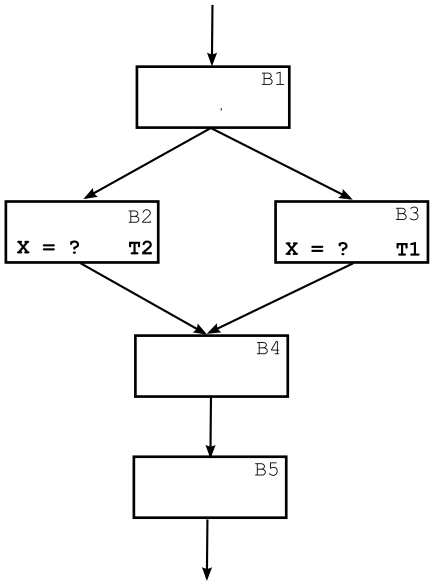


Figure 4.12: Illustration of two trigger points `T1` and `T2` at a memory write instruction to variable `X` for a call site.

address `(MEMWR == X)`. (It is assumed that no other part of the program modifies the memory location `X` in this example.)

4.8 Examples

Having discussed the details of the software implementation, I provide examples of triggers and handlers for some call sites chosen for PD based execution in SPEC CPU2000 integer benchmarks. Each example consists of two parts. The first part presents source code listing with line numbers from the program source file. The second part is the handler generated for a call site identified in the first listing. Triggers for the call site will also be discussed.

Consider the example from benchmark `twolf` in Listing 4.15. The handler generated for method `term_newpos_a`, called at line 96, is shown in Listing 4.16. The handler listing contains the program counter from the compiled binary, the method name, file name, and line number of the instruction included in the handler, and finally, the disassembled instructions. Besides having the call instruction to the method, and writing the parameter values to the stack, the handler includes the branch evaluation in line 95 as the call site is control dependent on it. The handler then performs all the computation necessary for generating the

```
 13 ucxx2( )
 14 {
 15
 32 delta_vert_cost = 0 ;
 33
 34 acellptr = carray[ a ]    ;
 35 axcenter = acellptr->cxcenter ;
 36 aycenter = acellptr->cycenter ;
 37 aorient  = acellptr->corient  ;
 38 atileptr = acellptr->tileptr ;
 39 aleft    = atileptr->left    ;
 40 aright   = atileptr->right   ;
 41 atermptr = atileptr->termsptr ;
 42
 43 bcellptr = carray[ b ]    ;
 44 bxcenter = bcellptr->cxcenter ;
 45 bycenter = bcellptr->cycenter ;
 46 borient  = bcellptr->corient  ;
 47 btileptr = bcellptr->tileptr ;
 48 bleft    = btileptr->left    ;
 49 bright   = btileptr->right   ;
 50 btermptr = btileptr->termsptr ;
 51
 52 newbinpenal = binpenal ;
 53 newrowpenal = rowpenal ;
 54 newpenal    = penalty  ;
 55
 56 new_old( bright-bleft-aright+aleft ) ;
 57
 58 find_new_pos() ;
 59
 60 a1LoBin = SetBin( startxa1 = axcenter + aleft  ) ;
 61 a1HiBin = SetBin( endxa1   = axcenter + aright ) ;
 62 b1LoBin = SetBin( startxb1 = bxcenter + bleft  ) ;
 63 b1HiBin = SetBin( endxb1   = bxcenter + bright ) ;
 64 a2LoBin = SetBin( startxa2 = anxcenter + aleft  ) ;
 65 a2HiBin = SetBin( endxa2   = anxcenter + aright ) ;
 66 b2LoBin = SetBin( startxb2 = bnxcenter + bleft  ) ;
 67 b2HiBin = SetBin( endxb2   = bnxcenter + bright ) ;
 68
 69 old_assgnto_new2( a1LoBin , a1HiBin , b1LoBin , b1HiBin ,
 70           a2LoBin , a2HiBin , b2LoBin , b2HiBin ) ;
 71
 72 a = sub_penal( startxa1 , endxa1 , ablock , a1LoBin , a1HiBin ) ;
 73 b = sub_penal( startxb1 , endxb1 , bblock , b1LoBin , b1HiBin ) ;
 74 c = add_penal( startxa2 , endxa2 , bblock , a2LoBin , a2HiBin ) ;
 75 d = add_penal( startxb2 , endxb2 , ablock , b2LoBin , b2HiBin ) ;
 76
 77 newbinpenal += a + b + c + d;
 78
 79 binpen_chg = newbinpenal - binpenal ;
 80 rowpen_chg = newrowpenal - rowpenal ;
 81 newpenal = (int)( roLenCon * (double) newrowpenal +
 82                 binpenCon * (double) newbinpenal ) ;
 83
 94
 95 if( ablock != bblock ) {
 96     term_newpos_a( atermptr , anxcenter , bycenter , aorient ) ;
 97     term_newpos_b( btermptr , bnxcenter , aycenter , borient ) ;
 98 } else {
 99     term_newpos( atermptr , anxcenter , bycenter , aorient ) ;
100     term_newpos( btermptr , bnxcenter , aycenter , borient ) ;
101 }
```

Listing 4.15: Program code from benchmark `twolf`. PD call site `term_newpos_a`, line 96 and `sub_penal`, line 72.

```
0x8069603 ucxx2 ucxx2.c 34  mov    0x80ea94c,%edx
0x8069610 ucxx2 ucxx2.c 34  mov    0x80eaa54,%eax
0x8069623 ucxx2 ucxx2.c 34  mov    (%edx,%eax,4),%eax
0x806962d ucxx2 ucxx2.c 34  mov    %eax,0xffffffd8(%ebp)
0x8069633 ucxx2 ucxx2.c 36  mov    0xffffffd8(%ebp),%ecx
0x8069643 ucxx2 ucxx2.c 37  movsbl 0x4(%ecx),%eax
0x8069647 ucxx2 ucxx2.c 37  mov    %eax,0xffffffc4(%ebp)
0x806964a ucxx2 ucxx2.c 38  mov    0x20(%ecx),%eax
0x8069654 ucxx2 ucxx2.c 41  mov    0x8(%eax),%eax
0x806965d ucxx2 ucxx2.c 41  mov    %eax,0xffffffd0(%ebp)
0x8069a02 ucxx2 ucxx2.c 95  mov    0x80eaa18,%eax
0x8069a07 ucxx2 ucxx2.c 95  cmp    %eax,0x80eaa4c
0x8069a0d ucxx2 ucxx2.c 95  je     8069e30 <.LBB14>
0x8069a13 ucxx2 ucxx2.c 96  mov    0x80ea9c4,%eax
0x8069a18 ucxx2 ucxx2.c 96  mov    0xffffffc4(%ebp),%edx
0x8069a1b ucxx2 ucxx2.c 96  mov    0xffffffd0(%ebp),%ecx
0x8069a1e ucxx2 ucxx2.c 96  mov    %eax,0x8(%esp,1)
0x8069a22 ucxx2 ucxx2.c 96  mov    0x80ea9f4,%eax
0x8069a27 ucxx2 ucxx2.c 96  mov    %edx,0xc(%esp,1)
0x8069a2b ucxx2 ucxx2.c 96  mov    %ecx,(%esp,1)
0x8069a2e ucxx2 ucxx2.c 96  mov    %eax,0x4(%esp,1)
0x8069a32 ucxx2 ucxx2.c 96  call   8050b80 <term_newpos_a>
```

Listing 4.16: Handler for `term_newpos_a` in benchmark `twolf`. For program code, see Listing 4.15.

```
0x80696d8 ucxx2 ucxx2.c 60  mov    0x80ea694,%ecx
0x80696e9 ucxx2 ucxx2.c 60  mov    0x80ea6e8,%ebx
0x8069751 ucxx2 ucxx2.c 62  mov    0x80ea9b0,%eax
0x8069756 ucxx2 ucxx2.c 62  mov    0x80eaa04,%esi
0x8069762 ucxx2 ucxx2.c 62  add    %esi,%eax
0x8069764 ucxx2 ucxx2.c 62  mov    %eax,0xffffff8c(%ebp)
0x8069767 ucxx2 ucxx2.c 62  sub    %ebx,%eax
0x8069769 ucxx2 ucxx2.c 62  cltd
0x806976a ucxx2 ucxx2.c 62  idiv   %ecx
0x806976c ucxx2 ucxx2.c 62  test   %eax,%eax
0x8069773 ucxx2 ucxx2.c 62  js     8069ee0 <.LBE14+0x60>
0x8069779 ucxx2 ucxx2.c 62  mov    %eax,0xffffffb4(%ebp)
0x8069789 ucxx2 ucxx2.c 63  mov    0x80eaa00,%edi
0x806978f ucxx2 ucxx2.c 63  add    %edi,%esi
0x8069791 ucxx2 ucxx2.c 63  mov    %esi,%eax
0x8069793 ucxx2 ucxx2.c 63  mov    %esi,0xffffff88(%ebp)
0x8069796 ucxx2 ucxx2.c 63  sub    %ebx,%eax
0x8069798 ucxx2 ucxx2.c 63  cltd
0x8069799 ucxx2 ucxx2.c 63  idiv   %ecx
0x806979b ucxx2 ucxx2.c 63  test   %eax,%eax
0x80697a2 ucxx2 ucxx2.c 63  js     8069ed0 <.LBE14+0x50>
0x80697a8 ucxx2 ucxx2.c 63  mov    %eax,0xffffffb0(%ebp)
0x80698dd ucxx2 ucxx2.c 73  mov    0xffffffb4(%ebp),%ecx
0x80698e2 ucxx2 ucxx2.c 73  mov    0xffffffb0(%ebp),%edx
0x80698e5 ucxx2 ucxx2.c 73  mov    %ecx,0xc(%esp,1)
0x80698e9 ucxx2 ucxx2.c 73  mov    0x80eaa18,%eax
0x80698ee ucxx2 ucxx2.c 73  mov    %edx,0x10(%esp,1)
0x80698f2 ucxx2 ucxx2.c 73  mov    0xffffff8c(%ebp),%edx
0x80698f5 ucxx2 ucxx2.c 73  mov    %eax,0x8(%esp,1)
0x80698f9 ucxx2 ucxx2.c 73  mov    0xffffff88(%ebp),%eax
0x80698fc ucxx2 ucxx2.c 73  mov    %edx,(%esp,1)
0x80698ff ucxx2 ucxx2.c 73  mov    %eax,0x4(%esp,1)
0x8069903 ucxx2 ucxx2.c 73  call   8049950 <sub_penal>
0x8069ed0 ucxx2 ucxx2.c 91  movl   $0x0,0xffffffb0(%ebp)
0x8069ed7 ucxx2 ucxx2.c 91  jmp    80697b8 <.LCFI5+0x1ba>
0x8069edc ucxx2 ucxx2.c 91  lea    0x0(%esi,1),%esi
0x8069ee0 ucxx2 ucxx2.c 91  movl   $0x0,0xffffffb4(%ebp)
0x8069ee7 ucxx2 ucxx2.c 91  jmp    8069789 <.LCFI5+0x18b>
0x8069eec ucxx2 ucxx2.c 91  lea    0x0(%esi,1),%esi
```

Listing 4.17: Handler for `sub_penal` in benchmark `twolf`. For program code, see Listing 4.15.

parameters and terminates at line 34. All live values in the handler are terminated with accesses to the heap, as per the handler generation algorithm. Therefore, speculative execution of the handler and the method (if called), will only consist of heap locations in the read and write set. The handler presented in Listing 4.16 is before it has been laid out; branch targets copied from the program binary must be remapped to targets within the handler. For example, line 95 in the handler is pointing to a target back in the program. The target is modified and set to the end of the handler, which is the instruction `hend` placed immediately after the call to `term_newpos_a`. The method has four parameters `atermptr`, `anxcenter`, `bycenter`, and `aorient`. To provide `atermptr`, the handler includes code in line 41, which in turn depends on line 38, and line 34. `anxcenter` is a heap location with no computation in the program code shown. `bycenter` is computed in line 45, but is not included in the handler because `bycenter` is allocated on the heap. The handler generation heuristic forbids store instruction to the heap. `aorient` is computed in line 37 which is in turn computed in line 34. The trigger for the call site in the example will depend on the data needed for the handler and the method.

Listing 4.17 presents the handler for call site in line 72, method `sub_penal`. Lines 62 and 63 in the code are macros and they are included in the handler. The handler terminates at line 60. After studying the trigger points, the execution of `sub_penal` is found to conflict with an instruction inside `old_assgnto_new2` that is called in line 69. Therefore, the trigger is placed immediately after line 69, and the handler is adjusted accordingly. (It will just have the instructions associated with line 73.)

Program code from benchmark `parser` is shown in Listing 4.18. The handler for call site `form_match_list` in line 510 is presented in Listing 4.19. After considering the trigger points for this call site, the method's executions are found to have dependencies within the loop body. The handler, therefore, increments the iterator twice (line 509) (determined by the trigger points), and then calls the method `form_match_list`. Note that in this example, the trigger is set to line 510, with the handler accessing data from the stack frame to perform its computation.

A similar example from benchmark `crafty` is shown in Listing 4.20. The handler for call site `UnMakeMove`, line 134, which is inside a loop, is shown in Listing 4.21. The jump target in line 126 will be altered to jump to `hend` during relayout. The trigger is placed immediately after line 127. Line 127 is

```
509     for (w=start_word; w <= end_word; w++) {
510     m1 = m = form_match_list(w, le, lw, re, rw);
511     for (; m!=NULL; m=m->next) {
512         d = m->d;
513         mark_cost++;
514         /* in the following expressions we use the fact that 0=FALSE. Could eliminate
515            by always saying "region_valid(...) != 0"  */
516         left_valid = (((le != NULL) && (d->left != NULL) && prune_match(le, d->left)) &&
517               ((region_valid(lw, w, le->next, d->left->next)) ||
518                ((le->multi) && region_valid(lw, w, le, d->left->next)) ||
519                ((d->left->multi) && region_valid(lw, w, le->next, d->left)) ||
520                ((le->multi && d->left->multi) && region_valid(lw, w, le, d->left))));
521         if (left_valid && region_valid(w, rw, d->right, re)) {
522         found = 1;
523         break;
524         }
525         right_valid = (((d->right != NULL) && (re != NULL) && prune_match(d->right, re)) &&
526                ((region_valid(w, rw, d->right->next,re->next))    ||
527                 ((d->right->multi) && region_valid(w,rw,d->right,re->next))  ||
528                 ((re->multi) && region_valid(w, rw, d->right->next, re))  ||
529                 ((d->right->multi && re->multi) && region_valid(w, rw, d->right, re))));
530         if ((left_valid && right_valid) || (right_valid && region_valid(lw, w, le, d->left)))
{
531         found = 1;
532         break;
533         }
534     }
535     put_match_list(m1);
536     if (found != 0) break;
537     }
```

Listing 4.18: Program code from `parser`. PD call site `form_match_list ()`, line 510.

```
0x8052eea region_valid parse.c 509  incl   0xfffffffe4(%ebp)
0x8052eea region_valid parse.c 509  incl   0xfffffffe4(%ebp)
0x8052e40 region_valid parse.c 510  mov    0xc(%ebp),%eax
0x8052e43 region_valid parse.c 510  mov    0x14(%ebp),%edx
0x8052e46 region_valid parse.c 510  mov    0x8(%ebp),%ecx
0x8052e49 region_valid parse.c 510  mov    %eax,0x10(%esp,1)
0x8052e4d region_valid parse.c 510  mov    0x10(%ebp),%eax
0x8052e50 region_valid parse.c 510  mov    %edx,0xc(%esp,1)
0x8052e54 region_valid parse.c 510  mov    0xfffffffe4(%ebp),%edx
0x8052e57 region_valid parse.c 510  mov    %ecx,0x8(%esp,1)
0x8052e5b region_valid parse.c 510  mov    %eax,0x4(%esp,1)
0x8052e5f region_valid parse.c 510  mov    %edx,(%esp,1)
0x8052e62 region_valid parse.c 510  call   804d900 <form_match_list>
```

Listing 4.19: Handler for `form_match_list ()` in benchmark `parser`. For program code, see Listing 4.18.

```
126   while (moves--) {
127     current_move[ply]=*(next_move++);
128 #if !defined(FAST)
129     if (ply <= trace_level)
130       SearchTrace(ply,0,wtm,alpha,beta,"quiesce",CAPTURE_MOVES);
131 #endif
132     MakeMove(ply,current_move[ply],wtm);
133     value=-Quiesce(-beta,-alpha,ChangeSide(wtm),ply+1);
134     UnMakeMove(ply,current_move[ply],wtm);
135     if (value > alpha) {
136       if(value >= beta) return(value);
137       alpha=value;
138     }
139   }
```

Listing 4.20: Program code from benchmark `crafty`. PD call site `UnMakeMove ()`, line 134.

```
0x8067d3e Quiesce quiesce.c 126  cmpl   $0xffffffff,0xfffffffe4(%ebp)
0x8067d49 Quiesce quiesce.c 126  je     8067dfd <.LBE4+0xc5>
0x8067dbc Quiesce quiesce.c 134  mov    0x14(%ebp),%edx
0x8067dc1 Quiesce quiesce.c 134  mov    0x10(%ebp),%ecx
0x8067dc4 Quiesce quiesce.c 134  mov    %edx,(%esp,1)
0x8067dc7 Quiesce quiesce.c 134  mov    0x8141660(,%edx,4),%eax
0x8067dd0 Quiesce quiesce.c 134  mov    %ecx,0x8(%esp,1)
0x8067dd4 Quiesce quiesce.c 134  mov    %eax,0x4(%esp,1)
0x8067dd8 Quiesce quiesce.c 134  call   806bc80 <UnMakeMove>
```

Listing 4.21: Handler for `UnMakeMove ()` in benchmark `crafty`. For program code, see Listing 4.20.

```
143 boolean    TmGetObject   (tokentype    *Token,
144                            ft F,lt Z,zz *Status,   addrtype   *Object)
145 {
146 dbheader   *CoreDb  = NullPtr;
147
148   if (TmFetchCoreDb  (Token,   McStat,        &CoreDb))
149   if (Normal(*Status))
150       HmFetchDbObject (CoreDb,  Token->Handle,  McStat,  Object);
151
152   if (*Status == Hm_ObjectNotPaired)
153       *Status  = Tm_ObjectNotPaired;
154
155 TRACK(TrackBak,"TmGetObject\n");
156 return (STAT);
157 }

743 boolean    HmFetchDbObject  (dbheader     *CoreDb,   handletype  Handle,
744                               ft F,lt Z,zz *Status,   addrtype   *Object)
745 {
746 *Object    = NullPtr;
747
748  if (Normal(*Status))
749  if (HmGetObjectAddr  (CoreDb,   Handle,  McStat,  Object))
750  if (*Object == NullPtr)
751  {
752     *Status = Hm_ObjectNotPaired;
753  }
754
755 TRACK(TrackBak,"HmFetchDbObject\n");
756 return (STAT);
757 }
758

743.2 boolean    MemGetAddr  (numtype       Chunk,  indextype   Index,
744.2                         ft F,lt Z,zz *Status, addrtype   *Addr)
745.2 {
746.2 addrtype     ChunkSlotAddr = NullPtr;
747.2
748.2
749.2    if (ChkGetChunk (Chunk, Index, sizeof(addrtype), McStat))
750.2    if (*Status != Set_EndOfSet)
751.2    {
752.2      *Addr = *(((addrtype *)(Chunk_Addr(Chunk))) + Index);
753.2       ChunkSlotAddr = (addrtype )((char *)Chunk_Addr(Chunk) + Index);
754.2    }
755.2
756.2 TRACK(TrackBak,"MemGetAddr\n");
757.2 return(STAT);
758.2 }
```

Listing 4.22: Program code from benchmark `vortex`. PD call site `ChkGetChunk`, line 749.

not included in the handler because of the write to a heap location, `current_move`. (Heap writes in the handler are not allowed in the implementation.)

```
0x80a793c TmGetObject tm.c 145  mov    0x14(%ebp),%ebx
0x80a7970 TmGetObject tm.c 149  mov    (%ebx),%eax
0x80a7972 TmGetObject tm.c 149  test   %eax,%eax
0x80a7974 TmGetObject tm.c 149  je     80a79c0 <.LCFI29+0x7e>
0x80a79c0 TmGetObject tm.c 150  mov    %ebx,0x10(%esp,1)
0x80a79c4 TmGetObject tm.c 150  mov    0x18(%ebp),%eax
0x80a79c7 TmGetObject tm.c 150  movl   $0x0,0xc(%esp,1)
0x80a79cf TmGetObject tm.c 150  movl   $0x0,0x8(%esp,1)
0x80a79d7 TmGetObject tm.c 150  mov    %eax,0x14(%esp,1)
0x80a79db TmGetObject tm.c 150  mov    (%esi),%eax
0x80a79dd TmGetObject tm.c 150  mov    %eax,0x4(%esp,1)
0x80a79e1 TmGetObject tm.c 150  mov    0xfffffff4(%ebp),%eax
0x80a79e4 TmGetObject tm.c 150  mov    %eax,(%esp,1)
0x80a79e7 TmGetObject tm.c 150  call   8072fa0 <Hm_FetchDbObject>
0x8072fa0 Hm_FetchDbObject hm.c 745  push   %ebp
0x8072fa1 Hm_FetchDbObject hm.c 745  mov    %esp,%ebp
0x8072fa3 Hm_FetchDbObject hm.c 745  sub    $0x28,%esp
0x8072fa6 Hm_FetchDbObject hm.c 745  mov    %esi,0xfffffffc(%ebp)
0x8072fa9 Hm_FetchDbObject hm.c 745  mov    0x18(%ebp),%esi
0x8072fac Hm_FetchDbObject hm.c 745  mov    %ebx,0xfffffff8(%ebp)
0x8072faf Hm_FetchDbObject hm.c 745  mov    0x1c(%ebp),%ebx
0x8072fb2 Hm_FetchDbObject hm.c 748  mov    (%esi),%eax
0x8072fb4 Hm_FetchDbObject hm.c 746  movl   $0x0,(%ebx)
0x8072fba Hm_FetchDbObject hm.c 748  test   %eax,%eax
0x8072fbc Hm_FetchDbObject hm.c 748  jne    8073007 <.LCFI47+0x58>
0x8072fbe Hm_FetchDbObject hm.c 749  mov    %ebx,0x14(%esp,1)
0x8072fc2 Hm_FetchDbObject hm.c 749  mov    0xc(%ebp),%eax
0x8072fc5 Hm_FetchDbObject hm.c 749  mov    %esi,0x10(%esp,1)
0x8072fc9 Hm_FetchDbObject hm.c 749  movl   $0x0,0xc(%esp,1)
0x8072fd1 Hm_FetchDbObject hm.c 749  mov    %eax,0x4(%esp,1)
0x8072fd5 Hm_FetchDbObject hm.c 749  mov    0x8(%ebp),%eax
0x8072fd8 Hm_FetchDbObject hm.c 749  movl   $0x0,0x8(%esp,1)
0x8072fe0 Hm_FetchDbObject hm.c 749  mov    0x878(%eax),%eax
0x8072fe6 Hm_FetchDbObject hm.c 749  mov    %eax,(%esp,1)
0x8072fe9 Hm_FetchDbObject hm.c 749  call   8082180 <Mem_GetAddr>
0x8082189 Mem_GetAddr mem10.c 745.2  mov    0x18(%ebp),%ebx
0x808218f Mem_GetAddr mem10.c 745.2  mov    0x8(%ebp),%esi
0x8082195 Mem_GetAddr mem10.c 745.2  mov    0xc(%ebp),%edi
0x8082198 Mem_GetAddr mem10.c 749.2  mov    %ebx,0x14(%esp,1)
0x808219c Mem_GetAddr mem10.c 749.2  movl   $0x0,0x10(%esp,1)
0x80821a4 Mem_GetAddr mem10.c 749.2  movl   $0x0,0xc(%esp,1)
0x80821ac Mem_GetAddr mem10.c 749.2  movl   $0x4,0x8(%esp,1)
0x80821b4 Mem_GetAddr mem10.c 749.2  mov    %edi,0x4(%esp,1)
0x80821b8 Mem_GetAddr mem10.c 749.2  mov    %esi,(%esp,1)
0x80821bb Mem_GetAddr mem10.c 749.2  call   807f960 <Chunk_ChkGetChunk>
```

Listing 4.23: Handler for `ChkGetChunk` from benchmark `vortex`. For program code, see Listing 4.22.

Consider program code from benchmark `vortex` in Listing 4.22. The handler for `ChkGetChunk`, called in line 749.2, is shown in Listing 4.23. After analysis of the trigger points, it is determined that the execution is dependent on line 148. Therefore, the trigger for the call site is placed at this point in the program. The handler first starts with code from `TmGetObject`. The branch in line 149 is evaluated. During layout, a `jmp` instruction to the target `hend` is placed after the `je` instruction. The handler then calls `HmFetchDbObject` (shown as `Hm_FetchDbObject` because of macro), which then calls `HmGetObjectAddr` (which is `MemGetAddr` because of macro), which finally calls the PD call site `ChkGetChunk`. The handler includes all stack pointer manipulating instructions to ensure that stack offsets in the instructions are still valid. The targets of the call instructions in lines 150 and 749 are corrected

```
40   switch (next_status[ply].phase) {
50   case HASH_MOVE:
51     last[ply]=GenerateCheckEvasions(ply, wtm, last[ply-1]);
61     if (hash_move[ply]) {
62       next_status[ply].phase=SORT_ALL_MOVES;
63       current_move[ply]=hash_move[ply];
64       if (ValidMove(ply,wtm,current_move[ply])) return(HASH_MOVE);
65       else printf("bad move from hash table, ply=%d\n",ply);
66     }
```

Listing 4.24: Program code from benchmark `crafty`. PD call site for `ValidMove()`, line 64.

```
0x805f439 NextEvasion nexte.c  40 mov    0x8(%ebp),%edx
0x805f43c NextEvasion nexte.c  40 lea    (%edx,%edx,2),%eax
0x805f43f NextEvasion nexte.c  40 lea    0x0(,%eax,4),%esi
0x805f446 NextEvasion nexte.c  40 mov    0x80f0660(%esi),%eax
0x805f451 NextEvasion nexte.c  40 cmp    $0x7,%eax
0x805f45b NextEvasion nexte.c  40 je     805f486 <.LCFI5+0x4d>
0x805f4a5 NextEvasion nexte.c  51 mov    0x8(%ebp),%edx
0x805f4ab NextEvasion nexte.c  61 mov    0x8179620(,%edx,4),%eax
0x805f4b2 NextEvasion nexte.c  61 test   %eax,%eax
0x805f4b4 NextEvasion nexte.c  61 jne    805f6c9 <.LCFI5+0x290>
0x805f6d0 NextEvasion nexte.c  64 mov    0xc(%ebp),%ecx
0x805f6d8 NextEvasion nexte.c  64 mov    %edx,(%esp,1)
0x805f6e1 NextEvasion nexte.c  64 mov    %eax,0x8(%esp,1)
0x805f6e5 NextEvasion nexte.c  64 mov    %ecx,0x4(%esp,1)
0x805f6e9 NextEvasion nexte.c  64 call   8070df0 <ValidMove>
```

Listing 4.25: Handler for `ValidMove` in benchmark `crafty`. For program code, see Listing 4.24.

during relayout. Their targets are instructions immediately following the call. Executing the call instruction, ensures that the stack operations performed by the speculative thread are identical to the operations that will be performed by the program. Return instructions are not required as the handler terminates immediately after the speculative call to `Chunk_ChkGetChunk`.

Finally, another example from benchmark `crafty` that illustrates more complex control flow is shown in Listing 4.24. The handler for call site `ValidMove`, in line 64, is presented in Listing 4.25. Analysis of trigger points indicates that the trigger can be set at the beginning of the method that the call site is located in (not shown). The handler includes the branch in line 61 and the switch evaluation in line 40. The jump targets are altered during layout, and in the fall through case, jump instructions are inserted to `hend` instruction.

4.9 Chapter summary

This chapter discussed the software support for the implementation of PD. Three inter-dependent steps were covered, namely, identification of call sites suitable for PD based execution, generation of handlers,

and generation of triggers for the identified call sites. These steps are performed with the use of profile information and other program analysis data structures. The chapter concluded with several implementation examples from the benchmark programs. In the next chapter, I present the hardware support required for the implementation. The implementation is evaluated in Chapter 6.

CHAPTER 5

HARDWARE SUPPORT FOR PD IMPLEMENTATION

This chapter describes the hardware support for the implementation of PD and concludes the discussion on implementation. First, I outline the hardware support needed for the implementation, and then expand on each of them further in this chapter.

Speculative execution. The major aspect of hardware support is the speculative execution of threads. In particular, extensions to the processing cores are needed to speculatively execute load and store instructions. Changes performed by stores cannot modify architected state, but must be observed and held separately, so that they can be later used or discarded. Similarly, load instructions must be monitored and provided with speculative data created by prior stores. Private caches in the processing cores are commonly used for this purpose.

Support for handlers. The handlers are generated by the software and incorporated into the application binary. Hardware support is needed to speculatively execute the handler with additional support to deal with any changes made by the handler depending on the code in it.

Holding executions. Each speculative thread consists of the read set, write set and data, return value, parameter values, and stack and base pointers. The hardware must provide some storage structure that holds the speculative threads until they are invalidated in violation of a dependency, or used by the program or another speculative thread. Private caches, commonly used in speculative parallelization systems for this purpose, may be used. Alternately, the threads may be placed in auxiliary storage structures.

Validating executions. Speculative executions must maintain sequential program order and, therefore, a thread that violates a dependency must be invalidated. For this purpose, the read set of a thread must be held along with the results of that speculative thread. If a store to a location by the non-speculative program is

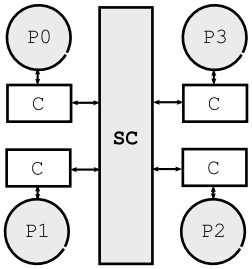


Figure 5.1: Model hardware with four processors connected to two levels of private caches (shown with only one box C), and a shared cache connected to memory.

present in the read and write sets of a speculative thread, that thread is invalidated. Another essential step in the process of validating executions is to determine if the parameters used for speculative execution of the method match the ones provided by the non-speculative program at the call site.

Using executions. The final step of a hardware implementation is to commit or use a valid speculative thread. First, the stack frame of the speculative thread is adjusted to match the stack pointer of the requestor (i.e., program or another speculative thread which is going to use an execution). Then, if the thread is requested by the non-speculative program, the write set data and return value are committed to the architected state. If requested by another speculative thread, the results and the read set are integrated into the speculative state of the requesting thread.

Support and evaluation of triggers. The triggers are generated with software support and incorporated into the binary. Two means, static and dynamic, were discussed in Section 4.7.4, each requiring different hardware and software support. For the generic dynamic case, the hardware must provide suitable register storage for predicates, extensions to the instruction set architecture to access and modify the registers, and evaluate the conditions in triggers efficiently.

5.1 Model hardware

The hardware support for PD can be implemented for any scale of architecture with support for parallel threads such as simultaneous multithreaded processors, chip multiprocessors, and in general, shared-memory multiprocessors. I assume a multicore system with one or more levels of private caches, with a

shared cache connecting all processing cores (refer Figure 5.1). The evaluation deals with a limited, up to eight, number of processors. The caches are kept coherent by means of a snoopy protocol, and the private caches are assumed to be writeback.

5.2 Performing speculative execution

This section deals with the foremost requirement for PD's execution model, which is performing the speculative execution. Speculative execution of threads in PD (and in other speculative parallelization systems), though conceptually similar to speculative execution of instructions in instruction-level parallel processing systems, requires different hardware implementation. In ILP systems, speculative execution is performed within a processing core; an instruction is held in a reorder buffer, executed with results held in some non-architected storage, such as the physical register file and/or store queue. It is eventually squashed, or when it reaches the head of the reorder buffer, committed. This results in discarding the instruction's changes, or applying them to the architected state.

In speculative parallelization systems (PD, inclusive), speculative thread execution is achieved across several processing cores. For simplicity, assume that the system is only running the program of interest. The processing core running this sequential program guarantees its correctness. Speculative threads are executed on other processing cores and are held until they are squashed when a dependency with the program is violated, or committed/used when they are requested by the (non-speculative) program or by other (speculative) threads. Speculative execution of a thread in a processing core is achieved by readying the live-in registers that may be needed, executing the instructions in the thread, but ensuring that the architected memory is unaltered by the execution. This is achieved by holding the changes made by stores separately, usually in the private cache of the processing core, and providing them to dependent loads. Program locations accessed and modified by the thread are tagged into write and read sets, respectively, along with the modified architected registers. These are held in private caches or auxiliary storage structure(s) until the speculative thread is invalidated or committed. The speculative thread is squashed and its data discarded, if it had referenced a location during its execution that is modified by the program, as it indicates violation of data dependencies.

In PD, a speculative thread consists of a method with parameters representing the explicit live-in values. The private cache(s) of the processing core used for speculative execution holds the speculative store data. A peripheral structure, called Speculative Tag Unit or STU, tracks the read and write sets during the execution. A speculative thread is aborted when a speculative cache line in the last level private cache is evicted or when the STU is full, as references may no longer be tracked. On completion of speculative execution, the data of a speculative thread is stored in an auxiliary set of storage structures, collectively referred to as the Execution Buffer Pool.

The accesses and changes made by a speculative thread can be tracked at the granularity of a byte to the entire cache line. Finer granularity requires more state bits to keep track of, increases overheads, but prevents false sharing, and invalidation of speculative data that fall in the same cache line but access different addresses. In this implementation, four writers are supported by dividing a cache line into four sub-blocks. Every sub-block in a cache line has a dirty state bit, and additionally a “speculative access” bit which is used to identify if a sub-block is accessed during the speculative execution (Figure 5.2). The propagation of a speculative access bit across the private cache hierarchy is similar to that of a valid bit in a cache line.

During speculative execution, cache sub-blocks accessed by the method, but not the handler, set the corresponding speculative access bit. Since speculative threads do not communicate data values in the implementation, cache misses obtain data from the memory or from the cache of the processing core running the non-speculative program. Of these, misses due to loads from the method (but not the handler) are recorded in the STU. A store does not request for exclusive access or send invalidate requests to other processing cores for the cache line it is to modify. If the store is to a sub-block that was not previously dirty, and is performed by the method (but not the handler), it is recorded in the STU with the corresponding W bit set. Stores performed by the processing core running the program represents the sequential program order, and sends invalidate messages, as usual, over the bus. A processing core that is running a speculative thread applies the following filtering rules to the invalidation requests: (i) it does not respond to any requests when it is executing a handler and, (ii) it does not respond to requests with addresses in the stack segment that are below the stack pointer communicated at the beginning of speculative execution. An unfiltered request aborts the speculative thread if it invalidates a sub-block with its “speculative access” bit or dirty bit set.

The speculative thread begins with the execution of the handler. An available processing core for

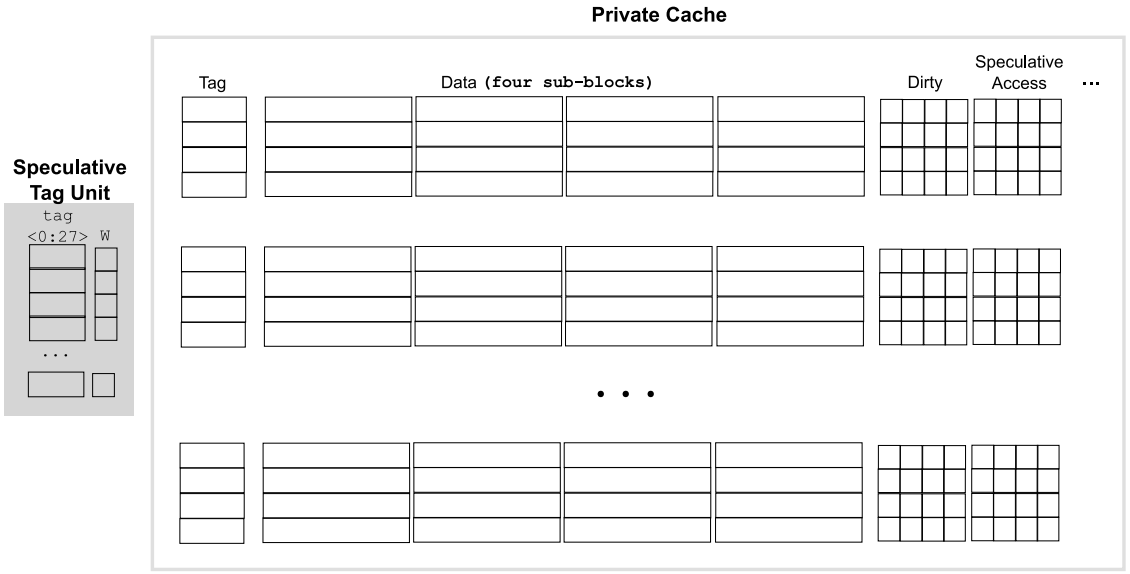


Figure 5.2: Extensions to the cache for speculative execution include four sub-blocks per cache line with each sub-block having a speculative access bit. The speculative tag unit is used to track the read and write sets of a speculative thread.

speculative execution is found. All speculative access bits are cleared in the private caches to indicate that no cache line has been accessed yet. The processing core switches to speculative mode, and receives the call site's program counter, handler's program counter, the stack and base pointers. These are saved in the processing core's special PD state registers, with the values of the handler's program counter, stack and base pointers copied to `eip`, `esp`, and `ebp` registers respectively. The speculative execution begins at the handler's program counter. At the execution of `pdcall` instruction, which is followed by the `call` instruction that begins speculative execution of the method, the parameter bounds are available. The bounds are saved in the PD state registers, the dirty bits set in the sub-blocks of the private caches are all cleared (they may be flash cleared), but remain valid. Note that the speculative access bits are not set during the execution of the handler.

At the end of the method's speculative execution, the read and write sets are collected from the cache. This is achieved by processing the entries in the STU (the write set entries have W bits set in STU). Dirty lines in the cache are all cleared, and the processing core returns to normal mode of operation. Sub-blocks in the write set that belong to the local stack frame (addresses below the stack pointer) are marked. The parameter sub-blocks are also included in the write set and marked. Along with the read set, write set and

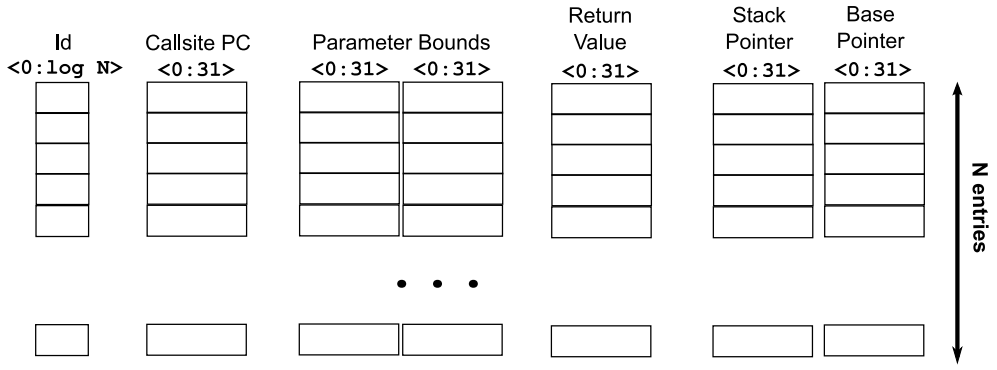


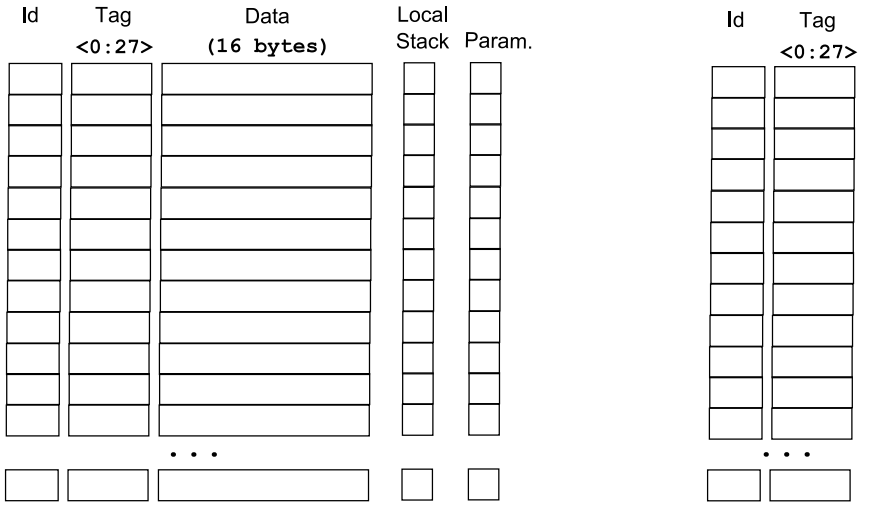
Figure 5.3: Methods table holds all speculative threads. An entry consists of the speculative thread identifier, call site program counter, addresses of the parameters, return value and stack base pointer used.

its data, the return value register, the parameter bounds and its values, the stack and base pointers, the call site's program counter are all collected and form the data associated with the speculative thread.

5.3 Execution buffer pool structures

A speculative thread after its execution may be held in the private cache, like previous speculative parallelization systems, or in an auxiliary cache storage structure. Outstanding speculative threads (i.e., threads that have completed execution waiting to be used or squashed) held in cache might place undue pressure on it. Therefore, this implementation uses auxiliary cache storage, called the Execution Buffer Pool, to eliminate the contention of conflicting accesses from other threads on that processing core.

The data of a speculative thread upon completion is moved into the Execution Buffer Pool. The execution buffer pool consists of the four structures: (i) The root structure is the Methods Table which holds the list of speculative threads outstanding (Figure 5.3). (ii) The Write Set Table holds the write sets of all speculative threads, and their parameters (Figure 5.4(a)). (iii) The Read Set Table holds the read sets of all speculative threads (Figure 5.4(b)). (iv) The Invalidation Cache holds the read and write set tags of all speculative threads for efficient invalidation of speculative threads (Figure 5.5). The implementation in this dissertation assumes a centralized execution buffer pool. For multicore systems with a large number of processing cores, the execution buffer pool may be distributed for a collection of (one to four) processing cores.



(a) Write set table (b) Read set table

Figure 5.4: Write (shown on the left) and read set tables holding the write and read sets of speculative threads. The write set in addition holds parameter values, specifically marked, so that the addresses are not considered for invalidation.

The data of a speculative thread is placed in the methods table and tagged with a thread identifier `tid`. Every entry in the methods table consists of the method's call site program counter, parameter values specified by the upper and lower bound addresses, the return value register, and the stack/base pointers. The methods table is used to search for outstanding execution for a given call site's program counter. Allocation and deallocation of entries is performed for a given identifier. Assume the number of entries in the table to be N for the rest of this discussion. Every entry requires $log\ N$ bits for an identifier, 32 bits for call site's program counter, 32 bits each for return value, stack pointer and base pointers, and 64 bits for parameter bounds.

The read and write set tables are indexed by the thread identifier `tid`. Every entry in the write set consists of the thread identifier, the sub-block address, the data, and local stack frame and parameter sub-block markers. Every entry in the read set consists of the thread identifier and the sub-block address. $log\ N$ bits are used for identifier, 28-bits for sub-block address, and 16-bytes for data for a sub-block of a 64-byte cache line. Allocation and deallocation of entries is performed for a given thread identifier. The hardware for these tables may be organized as N banks with a fixed number of entries for holding the read and write set, so that they may be efficiently accessed with latencies equivalent to accessing the level two cache. The number of entries may be determined based on the hardware complexity, access cycle constraints, and

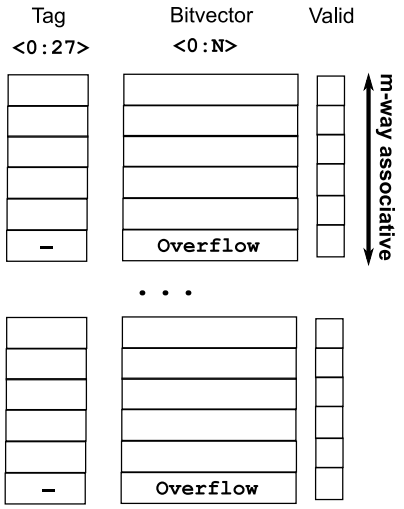


Figure 5.5: Invalidation cache is m-way set associative with address tag, and N bitvector for representing the speculative threads. The last entry of a set is designated as overflow, with the bitvector used as a counter.

studying benchmark programs that will be executed with PD. A speculative thread whose read and write sets do not entirely fit in these tables may be placed in an overflow table, a less efficient resource as it will hold the overflows for all speculative threads and must be associatively searched, but is typically less commonly accessed.

The invalidation cache is a separate set-associative cache-like structure that is used to efficiently determine if an address is in the read or write sets of a speculative thread that is stored in the execution buffer pool. Without the invalidation cache, an address must be searched in the entire read and write set tables of all speculative threads to determine a match. Every line in the cache consists of the sub-block tag to represent an entry in the read or write set, and data consisting of N bits representing the thread identifiers (known as the bitvector). A set in the cache is m-way associative, with the m-th entry's data block used as an "overflow" counter. The invalidation cache may also be separately maintained for the read and write sets and/or distributed for every processing core. Accesses to the invalidation cache are not critical to performance, and may be accessed with latencies equivalent to accessing level two cache.

The operations to allocate and deallocate a speculative thread and the operations on the invalidation cache are described next.

To allocate a speculative thread, an entry in the methods table is allocated (if available, otherwise, the thread must be aborted) and the thread identifier is obtained. The read and write sets are placed in the

corresponding tables. If the number of entries in the read and write set tables are fixed, entries that overflow must be placed in the overflow table. Every entry in the read set, and an entry in the write set that is not specially marked (as parameter or local stack frame sub-block), is passed to the invalidation cache. The set for the sub-block address is determined, and all lines are searched for the address. If the sub-block address is found, the bit that represents the speculative thread identifier in the bitvector is set. If the address is not found, and if an entry besides the reserved overflow entry is available in the set, the tag is installed and the bitvector corresponding to the thread identifier is set. If a free entry is not available, the overflow counter of that set is incremented if the value is less than the maximum, i.e., $2^N - 1$; otherwise, the speculative thread is aborted (deallocated).

To deallocate entries associated with a given thread identifier, which is performed when a speculative thread is squashed or committed, the corresponding entry is invalidated from the methods table. Then every entry in the read and write set tables for the given thread identifier is invalidated, and the sub-block address is sent to the invalidation cache. For every such address, the invalidation cache determines the set, and if an entry is found in the set, the bitvector is modified and the corresponding thread identifier is reset, marking the line invalid if the entire bitvector is zero. If an entry is not found, the overflow counter for the set is decremented.

5.4 Invalidating executions

One of the essential aspects of speculative parallelization is to maintain the validity of the results of the executions. The implementation in this dissertation uses “eager” invalidation, which sends all committed stores performed by the program to the invalidation cache, and as usual to other processing cores. Every address received by the invalidation cache is searched in the corresponding set. If an entry is found, every bit set in the bitvector represents the thread identifier of the speculative thread that violated dependencies, and must be squashed. (The deallocation process was described earlier.) If an entry is not found in the set, the search for a violating speculative thread continues, if and only if, the overflow entry for that set is greater than zero. In such a worst case, entries in the read set table and entries not specially marked in the write set table are searched to determine which speculative thread, if any, accessed the address. If an entry is found, that speculative thread is squashed.

An invalidate request will abort an ongoing speculative thread on a processing core, if the address is not filtered (discussed in Section 5.2) and is found in the private cache with speculative access bit set. System events such as timer and device interrupts also terminate speculative threads.

5.5 Committing and using executions

When the program or a speculative thread (referred to as the requestor) reaches a call site that is annotated with `pdcall` instruction, the call site program counter is sent to the methods table and to other processing cores. The methods table is searched for any matching entries, choosing the oldest one, if more than one entry is found. If no entries are found, and if a speculative thread is ongoing in a processing core, the requestor may stall until the execution completes. (Performance counters that indicate the usefulness of a speculative thread may be used by the requestor to determine whether to abort the speculative thread, and instead, execute the method at the call site on the its processing core.)

Once a speculative thread is found, using or committing the threads proceeds as follows. Data for a completed speculative thread is obtained from the write set table, or from the processing core executing the thread if the thread's execution is ongoing. First, by obtaining the parameter bounds at the requestor's call site (available in the `pdcall` instruction inserted before the call site), and from the speculative thread, values passed by the program and the handler are compared. If the values do not match, the thread is aborted, and the requestor must execute the method.

If the parameters used by the speculative thread and the values provided by the requestor match, the next step is to retag the local stack frame sub-blocks in the write set to the requestor's stack pointer. To hide the latency of all of the above operations, they may be initiated when a `pdcall` instruction is fetched by the front end of the processing core (with the help of some predecoding) and performed concurrently as the `pdcall` and the subsequent `call` instructions move through the pipeline and are committed. Operations performed by the requestor to use a speculative thread after the `call` instruction is committed are in the requestor's critical path.

Finally, the entire write set data for the speculative thread is integrated through the private cache of the requestor. If the requestor is the program, the write set data is non-speculative and committed. This will result in actions that are performed when a store is committed by the program, such as sending invalidate

messages to other processing cores and the invalidation cache. If the requestor is a speculative thread, the data is integrated as part of its speculative write set. The read set is also be integrated to the requestor's read set.

The integration process must be implemented by stalling the requestor for as few cycles as possible. One way to implement this is to stall the requestor until the write set tags (and the read set tags if the requestor is a speculative thread) are transferred. The requestor can proceed with the execution while the data sub-blocks of the write set are transferred. Load and store operations performed by the requestor during this transfer that conflict with the requestee's tags stall the requestor.

Another implementation is to let the requestor continue execution while the tags and data sub-blocks of the write set (and the read set tags if the requestor is a speculative thread) are being transferred. If a conflict is detected between the data of the speculative thread that is used and the instructions that are executed during the transfer by the requestor, the processor's rollback mechanism, which is used for speculative execution, may be used to squash and re-execute the instructions after the call site.

5.6 Supporting triggers

In the last chapter, two means of implementing triggers were discussed. The static approach inserted `pdfork` instructions to fork speculative threads directly into the binary. The hardware support needed is the instruction set extension to support `pdfork`. The generic dynamic approach requires hardware support for: (i) trigger condition code registers, which are used to evaluate predicates and store a true or false boolean value, (ii) instruction set extensions, to access and modify trigger condition code registers and perform logical operations on them, and finally, (iii) support for executing the microcode and determining if a trigger has fired.

To support the register part of a trigger (refer Section 4.7.4 for more details on the software implementation of triggers), when `tsetpc` is executed, an entry consisting of the predicate's program counter, the trigger condition code register, and the location of the evaluate part of the trigger is placed in the Trigger Evaluation Unit. The program counter of the predicate is registered with BF, a Bloom filter [23] (shown in Figure 5.6). The BF has a 1-bit hash bucket with some predefined hash function. During program's execution, all program counters of committed instructions are passed through BF. If the hash bucket for

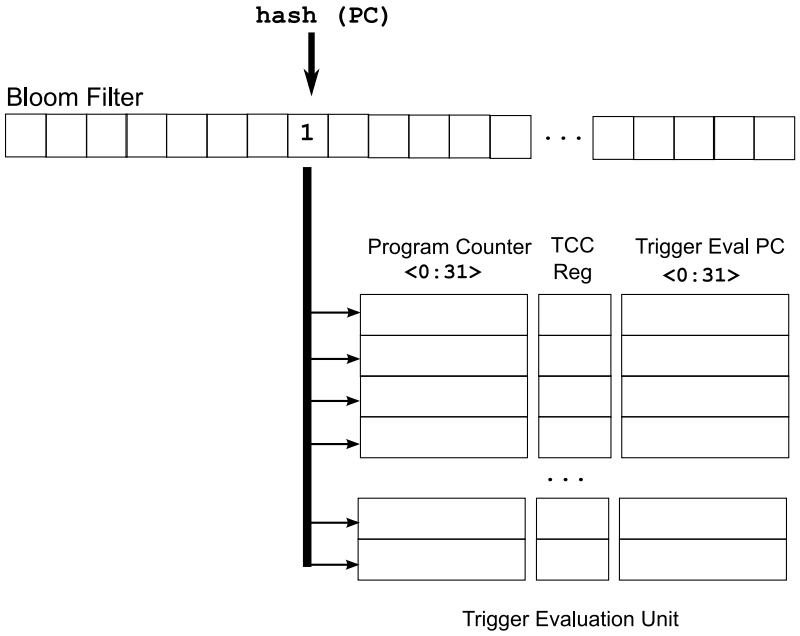


Figure 5.6: Program counters of committed instructions pass through a Bloom filter to determine if Trigger Evaluation Unit must be searched. The Trigger Evaluation Unit holds all program counters used in predicates of triggers, their trigger condition code registers, and program counters of evaluate parts of triggers.

the program counter is not set, the program counter is not part of any trigger's expression. Otherwise, the program counter is sent to the trigger evaluation unit where all entries are searched for a match. If one is found, the corresponding trigger condition code register is set, and the trigger evaluated by executing the evaluate part of the micro-code. The evaluate micro-code is executed either on free thread contexts on the processing core running the program (similar to SSMT [35]), or off-loaded to some other processing core, provided the trigger condition code registers are accessible to it.

The BF is accessed for every committed instruction and, therefore, may be accessed by one or more committed instructions per cycle. A buffering structure may be inserted between the committed instructions and the BF, especially if the BF is not accessible every cycle. The trigger evaluation unit entries are searched only during a hit to the BF, which is typically very infrequent and, therefore, need not be optimized for fast access.

5.7 Discussion of other implementation aspects

In this section, I discuss implementation aspects of Program Demultiplexing that are not considered in this dissertation.

Scheduling of threads

When triggered, speculative threads are scheduled on available processing cores on a first-come first-serve basis. More complex scheduling policies may be able to manage the processing unit resources better. For example, a speculative thread may be more critical to the computation that follows the program and, therefore, may speed up the program more than other threads. Such a speculative thread should be scheduled as soon as the trigger is fired, even if it requires aborting another less critical thread, to free up resources. Complex scheduling policies will only be practical if operations such as choosing a processing core for speculative execution are performed by software.

Storing speculative executions in cache

Private caches may be used to hold the data of speculative threads instead of using the invalidation cache and read/write set tables. No additional hardware storage structure is needed and a speculative thread can be easily committed as the dirty data of the thread is held in the cache. However, the disadvantage is that the cache is used to hold read and write sets of all speculative threads that are outstanding on a processing core. Any thread that causes a speculative cache line (accessed or modified) to be evicted must be squashed. Next, I describe the requirements for using private caches for holding speculative threads.

The methods table is required to hold the list of outstanding speculative threads. Each sub-block in the cache is extended with the thread identifier bitvector, so that the cache can serve as both the invalidation cache and read/write set tables. The mechanics for speculative execution are altered from this dissertation's implementation as follows. First, at the beginning of a speculative thread, cache lines are not invalidated. Second, when a speculative thread accesses or writes to sub-blocks, the corresponding thread identifier bit in the bitvector is set. Finally, if a speculative thread writes or accesses a dirty sub-block, or if a cache line that has a "speculatively accessed" sub-block must be evicted, that thread is aborted. The third requirement

prevents a speculative thread from accessing or modifying a sub-block after it is modified by an outstanding thread. Multiple speculative threads are, however, allowed to access a clean sub-block.

Invalidation and commitment of speculative threads can be achieved by extensions to the coherence protocol as suggested by Steffan [181]. When a speculative thread is invalidated (committed), all dirty references of that thread must be evicted (committed) from the cache. This may be achieved by searching all cache lines and identifying dirty sub-blocks with corresponding thread identifier bitvector set. A more efficient approach would be to supplement this scheme with the write set table for holding just the write set tags of a speculative thread, and populating it with the entries from STU at the end of its execution. The write set table for a given thread identifier may be used to determine the cache sub-blocks that must be evicted (committed).

In order for a speculative thread (requestor) to use the results of another thread (requestee), if the requestor is running on the same processing core as the requestor, the dirty lines of the requestee must be identified (by means discussed in the previous paragraph), and the thread identifier bitvector adjusted by setting the requestor's bit, and resetting the requestee's bit. If on the other hand, the requestor and requestee are on different processing cores, dirty sub-blocks must be retrieved from the requestor's cache, invalidated, and written to the requestee's cache.

Other hardware choices

Program Demultiplexing can be implemented on several other multicore and multithreaded hardware. Each of these choices presents benefits and disadvantages. I discuss some hardware choices next.

An asymmetric multicore system consists of cores with different computational power. Several asymmetric cores have been proposed due to their ability to provide good serial performance by having few high performance (say, out-of-order) cores and good throughput by having many (say, inorder) cores, in a power efficient manner. For PD, asymmetric systems are a good choice because speculative threads may not always need powerful processing cores because of the concurrency that PD may generate. In addition, parallelism may also provide the ability to tolerate long latencies instead of hiding it with out-of-order execution.

Shared memory multiprocessors may not be very suitable for PD because of the costly communication

latencies between processing cores. In typical applications, we may not have parallelism that is suitable for executing on processing cores with hundreds of cycles for communication.

Fine-grained multithreading and simultaneous multithreading hardware are techniques that are commonly used. Fine-grained multithreading time-shares several threads on a processing core and may be used to tolerate long latency events such as cache misses. Every thread executes for a given number of cycles or until an event such as cache miss occurs. On the other hand, simultaneous multithreading shares processing core resources among several threads simultaneously, to better utilize wasted resources. Therefore, instructions from several threads may coexist in a processing core's pipeline. In both these cases, threads scheduled on different thread contexts can share the private cache hierarchy, and benefit from the short communication latencies. The PD implementation must, however, be extended to differentiate multiple threads' contents in the cache, which can be achieved by marking cache lines or sub-blocks with speculative thread identifiers.

5.8 Chapter summary

This chapter discussed the hardware implementation for PD. First, the support required for speculatively executing a thread, much of which already covered in several dissertations on speculative parallelization systems, was discussed. Then, the means of storing, invalidating, and committing speculative threads, by the use of methods table, invalidation cache, and read and write set tables, was described. Finally, I covered the support needed for evaluating the triggers that are provided by the software.

CHAPTER 6

EVALUATION

In this chapter, I present an evaluation of the implementation of PD. The first part of this chapter will describe the hardware simulation infrastructure, software toolchain support for PD, and the benchmarks that are used for the evaluation. The second part of the chapter will present experimental results for the implementation of PD.

6.1 Hardware simulator

I evaluate the implementation with a simulation based machine model. The simulation infrastructure is based on Virtutech Simics [120], a functional, system level simulator that can simulate multiple processors along with the appropriate chipset and motherboard, and any peripherals or devices attached to the system. The simulated system can boot an unmodified operating system and run software installed on the disks of a simulated system. My implementation uses a system based on the Intel x86 instruction set architecture with Pentium 4 processors on an Intel 875P chipset with IDE disks. The system is configured to run the Linux operating system with kernel 2.6.8.

Simics does not model the processor micro-architecture or the memory hierarchy in the system. However, it allows user-written modules to be attached to it. In addition, it provides a micro-architectural programming interface that allows the capability to control the functional simulator inside of Simics. Every instruction executed by Simics is divided into five stages: fetch, decode, execute, memory operation, and commit. Any instruction can be inserted by the user module into the functional simulator and can be stepped through these five stages to eventually commit the instruction, or to squash the instruction and rollback the changes made by the instruction. Several instructions can be inserted into Simics, concurrently stepped through the stages, and may be executed out-of-order according to their data dependencies or before memory dependencies are resolved.

System	Virtutech Simics 2.0. Multiprocessor system with eight Intel Pentium 4 processors on an 875P chipset with IDE disks running Debian Linux (Kernel 2.6.8).
Processor Core	3-GHz out-of-order 4-wide superscalar processor with 7 pipeline stages. No cracking of instructions to micro-ops. 64-entry reorder buffer. 1024-entry YAGS branch predictor, 64-entry return address stack. Load instruction issues only after all prior stores are resolved.
Memory System	Level-1 private instruction cache is 32-KB, direct mapped, with a 1-cycle hit latency and with fetch buffer that prefetches the next line. Level-1 private data cache is 32-KB, 2-way with a 2-cycle hit latency, write-back and write-allocate. Level-2 private, inclusive, and unified cache is 1-MB, 4-way, with a 10-cycle hit latency. Level-3 is a shared cache, 4-MB with a 40-cycle hit latency. Line size is 64 bytes for all caches with four sub-blocks (16 bytes each) and MSI states for cache coherence. Cache-to-cache transfers take 12-cycles. Main memory is 512 MB DRAM with a 400 cycle access latency.

Table 6.1: Details of the simulated hardware

The architectural simulator uses this micro-architectural programming interface. The base processor core is a 3-GHz out-of-order 4-wide superscalar processor with seven pipeline stages. The instructions are not cracked to micro operations but executed as CISC instructions. The micro-architecture has a 64-entry reorder buffer with 32-entries for the load and store queues. A 1024-entry YAGS based branch predictor, and a 64-entry return address stack. Load instructions in the pipeline can issue only after all prior stores in the pipeline have been resolved. Each processing core has 32KB level one private instruction and data caches, followed by a 1MB level two private unified cache, and a level three unified 4MB cache. All levels are inclusive and write-back. More details are provided in Table 6.1.

The experimental infrastructure discussed does not reflect a commercially available chip multiprocessor. The level of details that can be simulated has to be deliberated carefully since it contributes to simulation time, which already is very expensive. A reasonable level of complexity is sufficient for exploring the benefits of PD, specifically to study the importance of ordering, i.e., unordered forking of speculative threads over the control flow based approaches of prior speculative parallelization systems. Complete modeling of the entire system is unlikely to change the insights obtained from this evaluation. It nevertheless, may affect the extent of performance achieved.

Memory profile	A window of 50 previously executed methods with respect to the call site is used to determine the trigger point for an execution of the call site. References are collected at 16-byte granularity and processed online for identification of trigger points.
Execution profile	Collected with train inputs for the entire run of the benchmark program using GNU `gprof`.
Overheads	All profiling information is collected and processed online for trigger and handler generation. The benchmark programs consume 800 MB to 1.2 GB of memory for profiling data, and is processed for handlers and triggers which takes under 5 seconds on an Intel Pentium 4 machine.

Table 6.2: Details of the PD profiling system

6.2 Software toolset and implementation

The software implementation is provided with a compiled binary of the program. First, debugging information such as the source file name and line number for all instructions in the application binary is extracted. The binary is then fed to the software tool chain that is built from Diablo [198], an open source program that is capable of reading Intel x86 program binaries and reconstructing compiler data structures. From this tool, the control flow and program dependence graphs are obtained.

The generation of handlers and triggers for the chosen call sites relies extensively on profile information. One approach is to generate all the profile information through instrumentation and then process the profile information offline to generate the handlers and triggers. An alternate approach that is used in this dissertation is online generation and processing of profile information. The simulator discussed in the previous section, besides performing its core task of simulating the hardware architecture, also collects the necessary profile information. This, along with compiler data structures (control flow and program dependence graphs), are used to generate the handlers and triggers for the list of call sites provided to the simulator. I generated the list of call sites by studying the execution profile of the benchmark programs. Details on the profiling system are listed in Table 6.2.

Using program binaries for constructing the software support for PD, instead of using the program source code or intermediate compiler representation, has its advantages and disadvantages. The primary advantages are that the implementation does not require a compiler infrastructure and can be applied to program binaries without access to source code. The disadvantage is the complexity of dealing with the Intel x86 instruction set architecture and its esoteric features, for example, the stack based access of floating

Benchmark	Description	Input	Input sets used
bzip2	Data compression utility	Train	
crafty	Chess program	Train	
gap	Computational group theory	Train	
gcc	C compiler	Train	
gzip	Data compression utility	Train	
mcf	Minimum cost network flow solver	Train	
parser	Natural language processing	Train	
perl	Perl	Train	Three runs for scrabble, perfect, and diff
twolf	Place and route simulator	Train	
vortex	Object oriented database	Ref	lendian2.raw
vpr	FPGA circuit placement and routing	Train	Two runs for place and route

Table 6.3: Benchmarks simulated from the integer suite of SPEC CPU2000 and input sets used

point registers. With the availability of source code and suitable compiler infrastructure, software support for PD may be implemented in the middle phases of the compiler where an intermediate representation of the program is available. In the middle phases, the intermediate representation usually has semantics as rich as the program itself and hence can be used to identify a loop, switch statement, and other control flow constructs easily. This can be useful in the generation of handlers and triggers.

6.3 Benchmarks

I use the integer programs in SPEC CPU2000 benchmark suite compiled for the Intel x86 architecture using the GNU gcc compiler version 3.3.3 for evaluation. The PD based program is compiled with optimization flags -O2 -mregparm=0 -fno-inline -fno-optimize-sibling-calls, and the sequential program, which is used for performance evaluation, is compiled with -O2 -mregparm=0.[5] The benchmarks are wrapped with additional libraries along with minor additions to program code to enable speculative execution in the presence of the operating system and system events. This is discussed in the next section. The benchmarks are run for 200 million instructions after the initialization phase, except for the run to collect execution

[5]Intel x86 program binaries commonly use the stack for passing parameters. GNU gcc provides a flag -mregparm=N that allows using N registers for this purpose instead. However, it requires recompilation of the entire system (including, libraries) with the same flag. This is beyond the scope of this dissertation. Another flag -funit-at-a-time has been introduced since GNU gcc 3.4. This flag, among other optimizations, uses registers for passing parameters only within a compilation unit (i.e., source file) of functions. The gcc task force has documented that this flag improves performance by 1% SPEC CPU2000. This flag is also not used for compiling the benchmark programs in this dissertation.

profile, which is performed for the entire run with profile data collected using GNU gprof. The benchmarks evaluated and input sets used are listed in Table 6.3 [6].

The SPECCPU2000 integer programs were chosen because they are commonly used in speculative parallelization publications and, therefore, facilitate comparison of opportunities across different proposals. The benefits with these programs could be significantly less than what we might see in future applications due to the following reasons. First, many of the benchmarks are tightly coupled and perform one specific task. Realistic applications may be significantly larger and may perform several tasks. For example, even the new SPEC CPU2006 has 30 times more source code than the SPEC CPU2000 programs. Second, the benchmark programs are written without any serious consideration of good software engineering principles. Several applications spend significant fraction of execution time on a few, very large methods. Others have an esoteric programming style that hinders the opportunities for parallelism. On the other hand, large scale applications are built with the use of software packages, libraries, and significant reuse of code. All of these indicate modular development and superior software engineering, which means that the opportunities for PD style of execution are expected to be higher.

Floating point programs are also not considered for the evaluation because they tend to exhibit structured parallelism and, therefore, are easily parallelizable with software libraries, for example, OpenMP. The floating point programs in SPEC CPU2000 in particular, have been successfully parallelized with minimal programmer effort, as in the SPEC OMP suite.

6.4 Creating speculative threads

An important aspect of this work is implementing speculative threads on a full-system simulator. While such an evaluation is not needed for PD and can astronomically increase the execution time of simulating the target multicore system, the issues in implementing speculative parallelization in the presence of an operating system must be studied. This section briefly discusses these issues.

The Linux operating system running on the simulated multicore system tries to use the available processing cores for scheduling other processes in the system or at least, to run the idle loop. Hiding the processing

[6] eon is a C++ program in the SPEC CPU2000 integer suite that is not evaluated since my software tool chain does not support C++ programs.

```
001
002 void t1 (void) {
003     unsigned long mask = 2;
004     if (sched_setaffinity (0, sizeof (unsigned long), &mask) != 0)
005         printf ("sched_setaffinity (): t1 failed\n");
006 label1:
007     goto label1;
008 }
010 void t2 (void) {
011     unsigned long mask = 4;
012     if (sched_setaffinity (0, sizeof (unsigned long), &mask) != 0)
013         printf ("sched_setaffinity (): t2 failed\n");
014 label2:
015     goto label2;
016 }
018 void main () {
019     pthread_t thread1, thread2;
020     pthread_attr_t t1_attr, t2_attr;
021     unsigned long mask = 1;
022
023     pthread_attr_init (&t1_attr);
024     pthread_attr_init (&t2_attr);
025
026     if (pthread_create (&thread1, &t1_attr, (void *) &t1, 0) != 0)
027         printf ("pthread_create (t1): failed\n");
028
029     if (pthread_create (&thread2, &t2_attr, (void *) &t2, 0) != 0)
030         printf ("pthread_create (t2): failed\n");
031
032   // ... have as many pthread_create's as number of processors ...
033
034     if (sched_setaffinity (0, sizeof (unsigned long), &mask) != 0)
035         printf ("sched_setaffinity (): failed\n");
036   ...
037 }
038
```

Listing 6.26: Use of pthreads to create wrapper threads that run idle loops usually, and are hijacked to run speculative threads.

cores or speculative threads running on them from the operating system is not a realistic system solution for the following reasons:

1. The processing core on receiving a timer or device interrupt that is intermittently delivered by the operating system invokes a service handler to process it. A scheduler determines which processing core gets to service the interrupt, and that is determined based on the system load. If a speculative thread is not visible to the operating system, the operating system may deliver the interrupts to the processing cores that are executing speculative threads, since the OS believes the processors are idle. This interrupts the speculative thread and aborts it.

2. A speculative thread may incur misses to the TLB during its execution. The Intel Pentium 4 architecture services some TLB misses (those that do not require querying page tables) without any support needed from the software. However, if the speculative threads and/or the processing cores they run on are invisible to the operating system, the global segment registers are not initialized by the operating system. The hardware, therefore, will not know the process generating the TLB misses on the hidden processing cores. This affects the operating system, which will panic and crash, instead of servicing the misses.

On the other hand, running speculative threads on OS-visible processors without the operating system's knowledge is catastrophic to the system; the OS may panic and crash. For example, the OS may try to schedule a process on a processing core, which may be already running a speculative thread. To deal with these issues, this dissertation transforms a single-threaded program into a multi-threaded program, with one thread running the SPEC integer application, and several other threads running a dummy idle loop, inserted into the program source code. This is shown in Listing 6.26. Lines 026 to 030 in the example creates two dummy threads executing methods `t1` and `t2`. Each of these threads executes `sched_setaffinity` to set its affinity and attach to a processing core (cores 2 and 3, in this example), and then executes the idle loop. A speculative thread hijacks a processing core that is running the idle loop, for its execution. The processing core, after finishing the speculative execution, returns to its idle loop. In addition to this, to prevent speculative threads from being interrupted from device timers, disk timers, and other intermittent events in the system, the IRQ load balancer's affinity is altered to prefer a processing core in a system that does not have an idle thread attached.

6.5 Evaluation

Several aspects of the implementation are presented in this section. They are broadly divided into the following categories, and expanded in the following subsections.

1. Methods and call sites chosen for PD based execution.

2. Potential for performance improvements with the chosen call sites with PD.

3. Results on handlers generated for the call sites.

4. Results on triggers generated for the call sites, and the hardware implementation results on the trigger evaluation unit.

5. PD implementation data which include, speculative execution overheads, results on the methods table, read and write sets, and invalidate cache, processing core utilization for speculative execution, cycles wasted, and the effect on private caches during speculative execution.

6. Performance improvements with PD including evaluation with latencies modeled between execution buffer pool and processing cores, hardware resource limitations, and a restricted program ordered forking model.

6.5.1 Methods

Table 6.4 presents the execution profile of the benchmarks[7]. For every benchmark evaluated, the table presents the methods that execute for more than one percent of the program's execution time. The first column presents the run time contribution of a method, the second lists the number of times it is called, and the third, the name of the method.

The methods in the table are used to obtain the initial set of candidate methods that PD can focus on because speculative execution of these methods, if achievable, may improve program's performance. I choose the frequently called methods as the candidate methods for PD. Examples include `Evaluate`, `EvaluatePawns` in `crafty`, `bea_compute_red_cost` in `mcf`; these methods are called tens of thousands to millions of times by the program. They are usually hundred to thousands of instructions, making them ideal opportunities for speculative execution. Methods infrequently called, such as `sortIt` in `bzip2` and `deflate` in `gzip`, are not chosen due to their large size (they can be inferred to be large, otherwise, they will not be in the execution profile table), which makes speculative execution unlikely due to limited speculative storage, and dependencies with the rest of the program, which limits parallelism. Studying the tables, benchmark programs `bzip2` and `gzip` have very few opportunities for PD, whereas, benchmarks such as `crafty`, `gap`, `perl`, `parser`, and `vortex` have many.

The number of methods and their call sites chosen for PD are listed in Table 6.5, rows 1 and 3. For comparison, the second row presents the total number of methods, including libraries, in the program. Only

[7]The results presented in this subsection are for entire runs on GNU gprof. For more details, see Section 6.2.

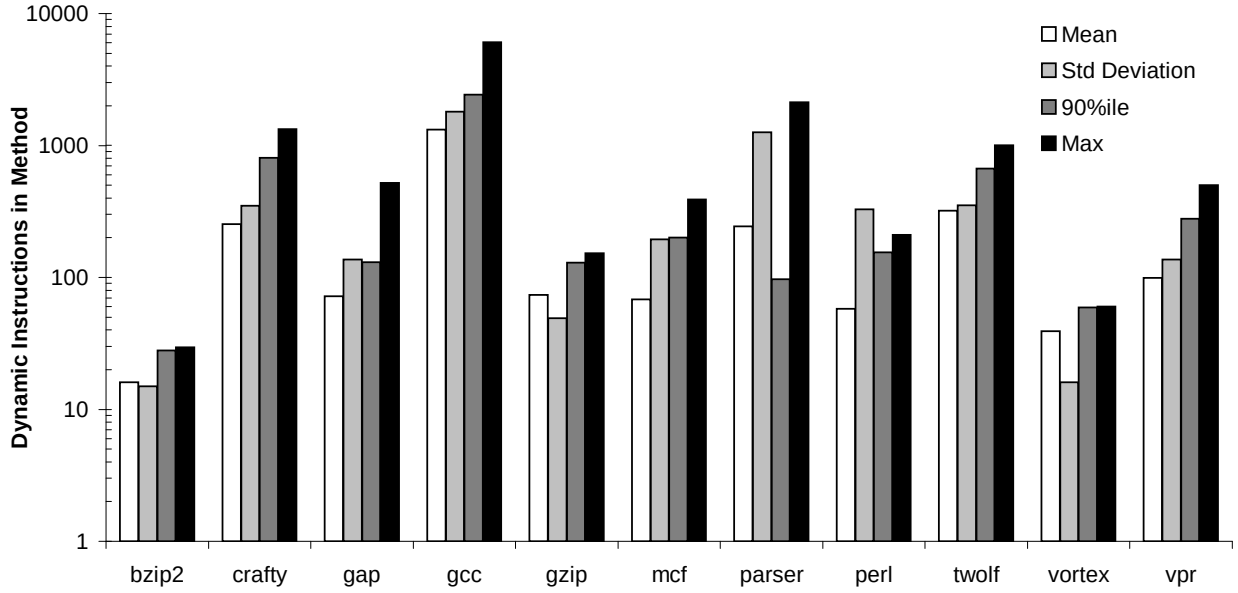


Figure 6.1: The number of dynamic instructions executed in the methods considered for PD

0.2 to 8% of methods are chosen for speculative execution with PD. The percentage execution time that the program spends in the call sites selected is listed in row 5. This execution time provides the upper bound on the percentage of program's execution time that can be concurrently executed with the program and, therefore, the maximum attainable performance improvements.

The methods selected are then provided as inputs to the software tool chain (See Section 6.2). By means of profile information, handlers and triggers for the call sites of the methods are generated. The total number of call sites that the software tool chain selected is listed in row 4 of the table. Some call sites are eliminated from the initial list due to implementation issues when generating handlers and triggers.

I conclude this subsection by presenting the number of dynamic instructions executed by the methods selected for PD in Figure 6.1. The bars represent the mean, standard deviation, 90-th percentile, and maximum of the method sizes. For the two compression programs, `bzip2` and `gzip`, none of the infrequently called (large) methods were included for PD, and the frequently called methods in them are typically less than a hundred instructions. The number of times they are invoked contributes to a large percentage of the program's execution time. The rest of the benchmark programs has average method sizes in fifties to several hundreds of instructions. Benchmark `gcc` is the exception with method sizes usually in the thousands of instructions. The performance results later presented indicate that the large size of the methods in `gcc`

bzip2	% Exec.	Calls	Name
	30.75	24204832	fullGtU
	18.07	31	sortIt
	14.40	22	generateMTFValues
	12.99	22	getAndMoveToFrontDecode
	9.19	22	undoReversibleTransformation_fast
	4.11	22	sendMTFValues
	2.16	46709960	spec_getc
	1.71	30004816	spec_putc
	1.58	840223	simpleSort
	1.30	17549524	bsW
	1.10	16705144	getRLEpair

crafty	% Exec.	Calls	Name
	19.41	12255414	Evaluate
	8.78	4594853	EvaluatePawns
	7.10	4413628	GenerateCaptures
	6.93	11954229	MakeMove
	6.88	119382079	FirstOne
	6.71	19750559	Attacked
	6.20	94568670	PopCnt
	4.95	11954225	UnMakeMove
	4.86	1272	Search
	4.17	7709098	Swap
	3.74	8610885	Quiesce
	3.61	9233312	NextMove
	3.57	8659517	AttacksTo
	2.67	12063583	SwapXray
	2.32	27302935	LastOne
	2.19	2493056	LookUp
	1.42	315353	GenerateCheckEvasions
	1.42	281116	GenerateNonCaptures

gap	% Exec.	Calls	Name
	9.15	4	CollectGarb
	8.94	42974935	EvVar
	6.24	6860751	NewBag
	4.57	7251117	EvElmList
	4.16	3431400	EvFunccall
	3.53	1863720	ChangeEnv
	3.12	6113342	ExitKernel
	3.12	204565	DiffVecFFEVecFFE
	2.70	2304118	EvRecElm
	2.49	3167288	LtPP
	2.49	314194	EvFor
	2.29	3108312	EvIf
	2.29	2154630	EvAssList
	2.08	3711457	EvVarAss
	2.08	2811377	Ne
	2.08	2372634	EqFFE
	2.08	1350063	ProdInt
	2.08	215030	ProdFFEVecFFE
	1.87	2791812	Diff
	1.66	1333693	Prod
	1.66	344475	FunAppend
	1.46	1759148	EvAnd
	1.46	904183	Resize
	1.46	174946	QuoIP
	1.25	298978	FunIsBound
	1.04	6113342	EnterKernel
	1.04	430753	MakeList

gcc	% Exec.	Calls	Name
	20.10	25313	propagate_block
	19.52	26454	new_basic_block
	9.30	25064	schedule_block
	4.26	28	life_analysis
	3.29	56	thread_jumps
	3.10	215	force_movables
	2.52		memset
	2.42	44161	record_one_conflict
	2.08	114066	sched_analyze_insn
	1.69	215	count_loop_regs_set
	1.21	5960	find_reg

gzip	% Exec.	Calls	Name
	33.63	82599334	longest_match
	11.17	5	deflate
	9.56	122481874	ct_tally
	8.50	5115	fill_window
	8.47	3668	inflate_codes
	6.71	125222294	send_bits
	5.52	2	deflate_fast
	5.02	3668	compress_block
	4.67	10250	updcrc
	3.44		memcpy

mcf	% Exec.	Calls	Name
	35.31	5235	refresh_potential
	21.88	104573	primal_bea_mpp
	18.64	93235666	bea_compute_red_cost
	9.14	6	price_out_impl
	4.29	93235666	bea_is_dual_infeasible
	2.93	104567	sort_basket
	1.88	1022350	replace_weaker_arc
	1.76	92516472	compute_red_cost
	1.00	104567	update_tree

parser	% Exec.	Calls	Name
	10.18	11361658	hash
	7.09	10084223	table_pointer
	6.44	11515427	prune_match
	6.19	25993211	xalloc
	5.67	2589707	form_match_list
	5.15	25820721	xfree
	3.74	684389	region_valid
	3.09	8038607	possible_connection
	2.96		__ctype_b_loc
	2.58	467	free_table
	2.19	43340	clean_table
	2.19		__pthread_internal_tsd_address
	2.06	9788229	power_hash
	2.06	2588929	catenate
	2.06	8410	build_clause
	1.93	9504250	table_lookup
	1.80	5006344	match
	1.55	1384118	hash_S
	1.55	1237	count
	1.42	4885415	right_table_search
	1.42	663342	left_connector_list_update
	1.29	3511965	fast_match_hash
	1.03	2545165	left_table_search
	1.03	467	init_table

Table 6.4: List of methods that contribute greater than one percent of the execution time, the number of times the methods are called, and their names

perl	% Exec.	Calls	Name
	27.33	19493591	regmatch
	8.04	14459657	Perl_my_bcopy
	6.54	58517594	regrepeat
	4.88	24674440	Perl_pp_padsv
	4.20	11109892	Perl_sv_setsv
	3.06	17382160	Perl_pp_nextstate
	2.85	19493591	regtry
	2.54	3977825	Perl_regexec_flags
	1.92	6672736	Perl_pp_and
	1.66	5556109	Perl_sv_upgrade
	1.45	5300124	Perl_leave_scope
	1.40	6513307	Perl_sv_setpvn
	1.40	1218778	Perl_pp_match
	1.35	161	Perl_runops_standard
	1.14	5423898	Perl_sv_clear
	1.14	2900601	Perl_pp_gvsv
	1.14	1148534	Perl_pp_entersub
	1.04	9083326	Perl_pp_const
	1.04	7322234	Perl_pp_sassign
	1.04	2429929	Perl_mg_get

twolf	% Exec.	Calls	Name
	30.30	3189275	new_dbox_a
	14.60	2023349	new_dbox
	8.69	2730486	ucxx2
	7.93	120	uloop
	6.58	1724967	term_newpos_a
	4.47	1464308	term_newpos_b
	3.21	5855046	sub_penal
	3.12	6596085	XPICK_INT
	3.12	5855046	add_penal
	2.45	3011773	acceptt
	2.28	12983918	Yacm_random
	2.19	2730486	old_assgnto_new2
	2.19	2023349	term_newpos
	1.52	468153	dbox_pos_2
	1.10	3124560	new_old
	1.01		_IO_vfscanf

vortex	% Exec.	Calls	Name
	12.63	481266209	Chunk_ChkGetChunk
	10.11	286162243	Mem_GetWord
	8.49	144000	Part_Delete
	7.28		memcpy
	5.73	144692956	Mem_GetAddr
	4.63		_int_malloc
	3.70	18185478	OaGet
	3.66	176132271	TmFetchCoreDb
	3.06	101130315	TmGetObject
	2.85	101130315	OaGetObject
	2.30	101130315	Hm_FetchDbObject
	2.11	48751402	Mem_GetBit
	2.10	156232	SaFindIn
	1.79	13352195	OaCompare
	1.77	50458735	TmIsValid
	1.30	4128498	Tree_CompareKey
	1.22	288001	SaDeleteNode
	1.10	7385700	DbmGetVchunkTkn
	1.00	41196116	Ut_MoveBytes

vpr	% Exec.	Calls	Name
	34.02	15491273	get_heap_head
	24.68	8845923	expand_neighbours
	11.31	28972981	add_to_heap
	9.49	43692353	node_to_heap
	6.38	10746	route_net
	6.23	28972981	alloc_heap_data
	1.59	28972981	free_heap_data
	1.21	10746	reset_path_costs

Table 6.4: List of methods that contribute greater than one percent of the execution time, the number of times the methods are called, and their names... contd

Benchmarks	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Methods	5	23	14	10	5	4	11	37	10	10	15
Total Methods	279	307	1051	2464	321	214	527	1297	386	1156	471
Call Sites	23	188	48	32	25	5	133	57	26	174	30
PD Call Sites	13	109	42	20	16	5	44	38	26	110	12
Execution Time	35.0	65.9	51.9	50.6	54.2	53.5	56.5	79.1	89.4	55.5	63.6

Table 6.5: Number of methods and call sites considered for PD based execution

makes them a poor choice for speculative execution because of limited parallelism and hardware resource constraints.

Standard deviations for the benchmark programs are close to or higher than the mean, with 90-th percentile more than the mean, indicating that the methods sizes are widely distributed. Exceptions include the benchmark `parser` whose mean (= 243) is much higher than the 90-th percentile (= 97), indicating the

benchmark has a few very large methods (thousands of instructions) and many small methods with tens of instructions.

6.5.2 Potential for PD-based execution

This subsection presents the potential for PD, which is the ability to begin speculative execution of a method before its corresponding call site in the program. This is specified by the ratio R defined as follows:

$$R = \frac{Cycle_{Call\ Site} - Cycle_{Trigger\ Point}}{Execution\ Cycles}$$

R is the ratio of the difference of the cycle when the method and its handler are ready to execute, i.e., the trigger point, and the cycle when the program reaches the call site, to the number of cycles to execute the method. In the case when the method (caller) being evaluated for PD calls another method (callee) during its execution, the trigger point is computed also including the callee, only if the callee is not considered for PD. All the cycle times are obtained from the sequential execution of the program.

The ratio R is not a direct measure of performance benefits for two reasons: (i) The execution cycles used for the computation is measured on the processing core running the program. In PD based execution, this number is expected to be different as the speculative thread executes on another processing core (refer Section 6.5.6 on overheads of speculative execution). (ii) The data is collected for every call site individually, which is without the influence of other methods. In a PD based execution, however, the cycle time for a given call site and its trigger point used in the ratio will vary and depend on other call sites speculatively executed.

The ratio, however, gives an indication of how much a method interferes with the rest of the program, as it determines the parallelism that is available and how well the method's speculative execution could be overlapped. A ratio close to 0 indicates that a method and its handler's execution is ready only just before the method is called, therefore no overlap is possible with the program. A ratio close to 1 indicates the method can be executed immediately after it is ready; it may be able to finish execution just before it is called when overheads of speculative execution are considered. A ratio $>>$ 1 indicates possible opportunity for completely overlapping the execution even with any overheads.

Figure 6.2 plots the ratio R averaged over all executions of a given method's call site on a logarithmic Y axis. Each point in the graph represents a call site chosen for PD. From the graph we see that many call sites could begin execution well before they are called. Note that the call sites plotted in the graph were chosen after analyzing their execution time contribution and size. These metrics have already pruned methods that may be unsuitable for PD.

A more insightful graph is the cumulative plot of the percentage of execution time, plotted on the Y axis, over ratio R on logarithmic X axis, as shown in Figure 6.3. This graph gives an idea of the percentage of the program's execution time and the extent to which that percentage may be overlapped with the rest of the program. For example, 46% of benchmark `parser`'s execution time has a ratio R of 1, `twolf` with 50% of its execution time with ratio greater than 0.8, and `crafty` with over 60% greater than 0.8. From this graph, reasonable performance benefits from benchmarks `gap`, `parser`, `crafty`, `twolf`, and `vpr` are expected. As pointed out earlier, the data cannot be used as a first order measure of performance benefits because of the interaction of speculative threads in PD with the program's critical path.

Sensitivity of R to varying handlers and triggers. The results presented so far are for a specific implementation of a handler. R may vary with other implementations. The rest of this subsection will focus on the effect on ratio R with different implementations of handlers and restrictions on trigger points. Figure 6.4 plots the ratio R assuming the trigger points cannot occur earlier than the beginning of the method that calls the method being considered for PD. This prevents the generation of any interprocedural slices in the handler (discussed in Section 4.6.1). The following are the noteworthy points from this graph. Overall, the ratio has not dramatically changed, even though aberrations in some benchmarks are clearly visible. For `mcf`, the limitation of handler has reduced the ratio for one call site drastically (10s to 0.1). In general, this form of handler can severely restrict the separation of trigger site and call site if the program invokes many methods during its execution. For example, benchmarks `crafty`, `parser`, and `vortex`, all have the points in the graph located lower than the points in Figure 6.2 because these programs spend their execution time over many methods. Benchmark `vpr` also has some perturbations and some points shift closer to (and below) the X axis.

Another study, shown in Figure 6.5, plots the ratio R assuming that the handlers do not evaluate any branches. This reduces the dependencies that the handler may have with the program and could affect the

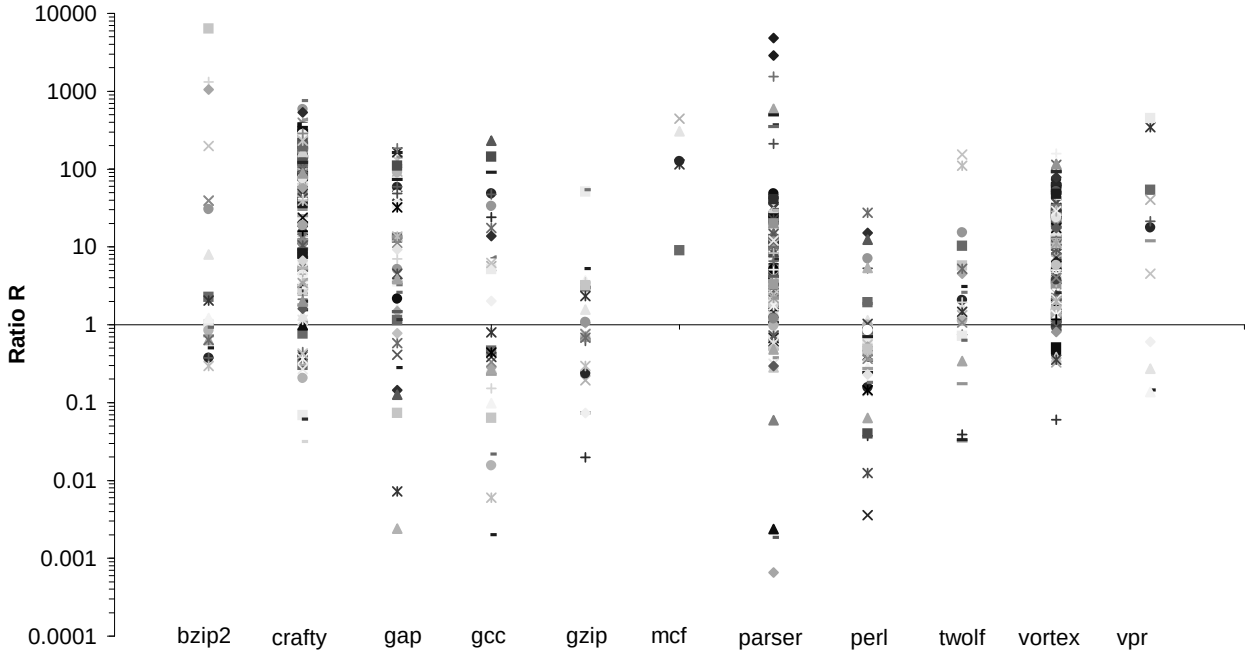


Figure 6.2: Ratio R (described in the text) plotted for different call sites in the benchmark. The trigger point for a call site in this study is the earliest the method and its handler can begin speculative execution.

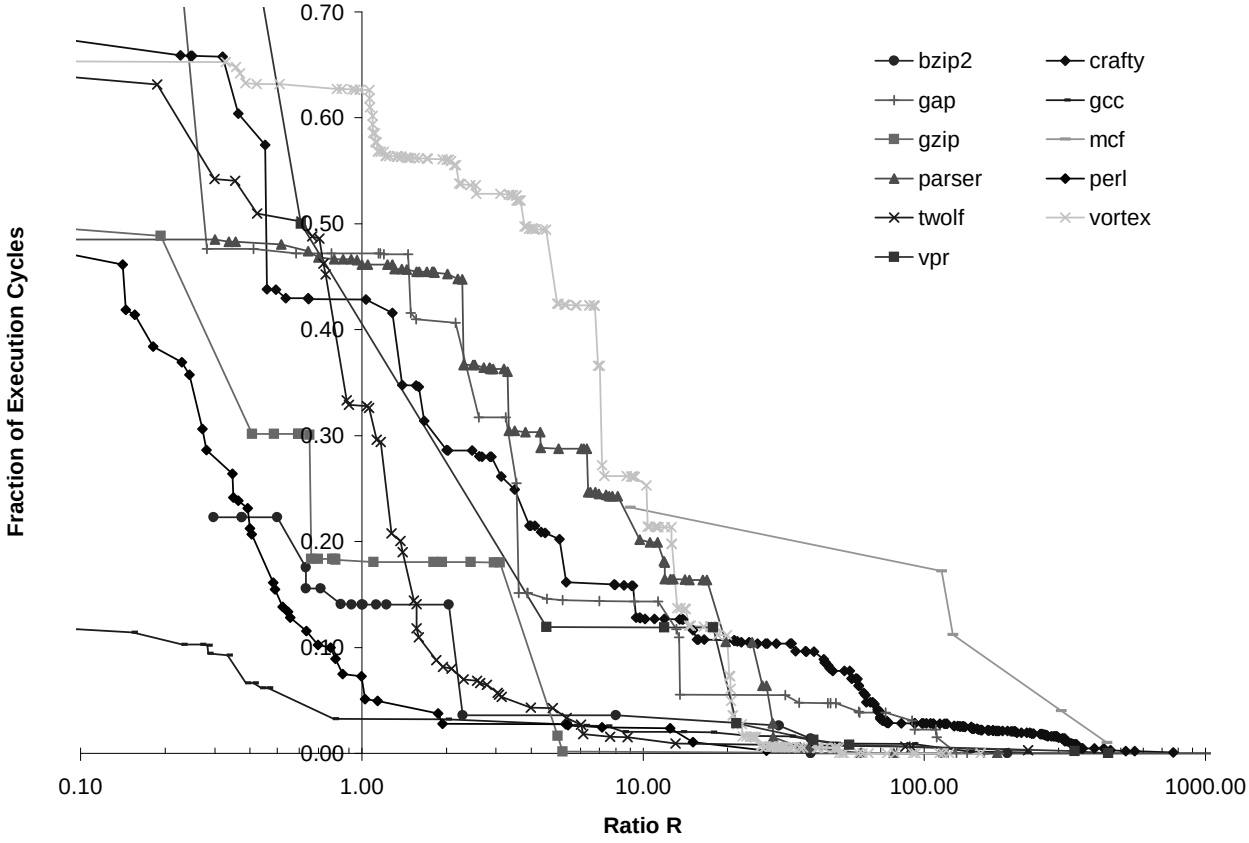


Figure 6.3: Cumulative plot of the execution time versus the ratio R plotted in Figure 6.2

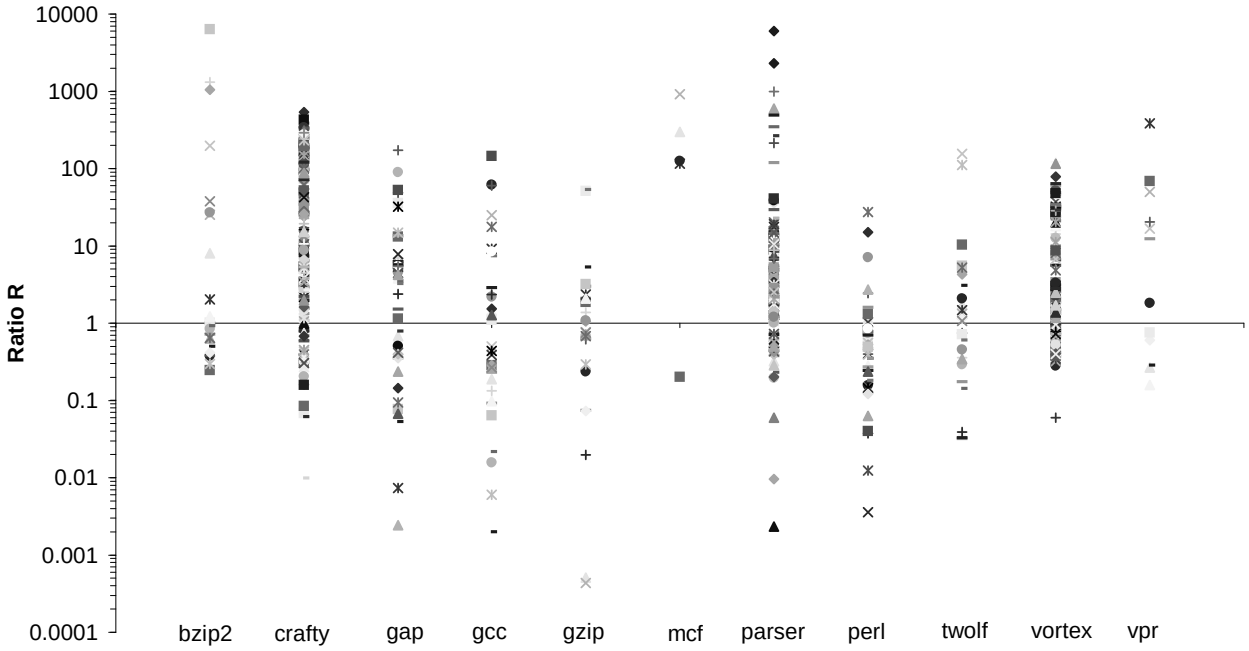


Figure 6.4: Ratio R assuming the trigger points cannot be beyond the scope of the method that has the call site of the method being considered for PD

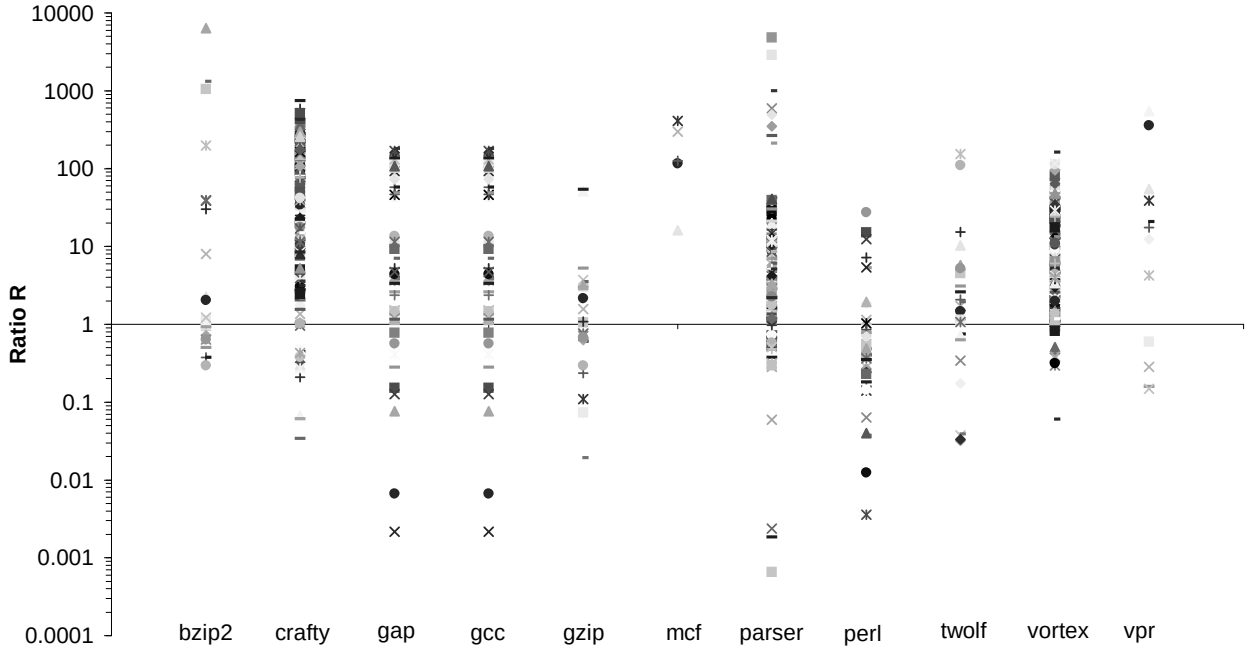


Figure 6.5: Ratio R assuming the trigger points are based on handlers that do not compute any branches

	Mean	St. Dev	90%ile	Max
bzip2	1.28	0.39	1.81	2.25
crafty	12.61	13.02	34.20	53.80
gap	4.46	7.08	8.64	38.70
gcc	27.67	96.64	19.92	477.00
gzip	1.90	4.06	1.27	21.37
mcf	10.68	5.64	14.71	15.48
parser	1.92	2.13	4.47	11.46
perl	1.00	0.01	1.00	1.06
twolf	4.60	11.10	5.86	42.80
vortex	3.03	7.02	4.44	48.90
vpr	2.53	3.81	6.02	13.51

Table 6.6: Ratio of the cycles elapsed between call site in the program and its trigger point assuming many outstanding executions compared to just one

location of trigger points. All benchmarks see very little effect when branches are excluded, indicating that it may not have significant impact on the trigger points.

In the results so far presented, it is assumed that more than one speculative thread can be outstanding for a given call site, before they can be committed or squashed. Table 6.6 presents the ratio of $Cycle_{Call\ Site} - Cycle_{Trigger\ Point}$ for the case studied so far, which is having several outstanding executions for a given call site, versus, having just one outstanding execution for the call site. In the latter case, the trigger point for a speculative thread can only be after the previous thread's execution has been committed or squashed. For example, assume that a program calls method `M` repeatedly with different parameter values (say, a pointer to a data structure). Also assume that `M` performs some computation with that parameter and each of the executions are independent. Having only one outstanding execution severely limits the opportunities for concurrent speculative executions, especially if the invocations of `M` have no conflicts. In benchmark `mcf` this is noticeable from the high mean of 10x. Similarly, in `crafty`, a chess-playing program, few methods are called to analyze several different moves. Restricting the number of outstanding speculative executions severely limits the parallelism.

6.5.3 Handlers

Table 6.7 presents the number of instructions present in the handlers generated for PD, as the mean, standard deviation, 90-th percentile, and maximum. These numbers are dependent on the handler generation algo-

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Mean	2.91	30.29	33.15	27.50	10.70	19.60	14.66	10.08	21.79	36.96	8.17
Std Dev	1.70	54.02	35.23	20.83	6.19	17.76	18.53	11.63	22.60	24.74	8.38
90%-ile	6.40	66.00	91.40	46.90	20.00	37.60	42.00	27.50	36.80	73.60	16.70
Max	7.00	301.00	125.00	102.00	22.00	50.00	90.00	40.00	94.00	125.00	28.00

Table 6.7: Number of instructions in handlers generated with interprocedural slicing

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Mean	2.70	13.20	5.54	8.53	6.88	10.40	5.28	6.18	9.54	10.26	5.50
Std Dev	1.29	13.06	2.00	5.15	3.03	12.80	4.64	6.53	14.46	2.88	6.19
90%-ile	4.60	31.00	8.00	15.90	11.00	23.00	11.00	13.40	11.00	13.00	12.50
Max	7.00	69.00	11.00	21.00	12.00	33.00	39.00	32.00	78.00	31.00	21.00

Table 6.8: Number of instructions in handlers generated without interprocedural slicing

rithm and the heuristics, the program, and its programming characteristics. The mean for the benchmarks varies from two to forty instructions. The maximum gives a clearer idea on the complexity of the handler for different benchmarks. For example, `bzip2` has a maximum of seven instructions in the handler, whereas `crafty` has 300 instructions. This is due to two reasons: (i) fewer parameters passed in `bzip2` than `crafty` and (ii) `crafty`'s extensive use of the stack to communicate data values in the program. Since the handler generation algorithm includes stack related computation, more instructions are included during the slicing process in `crafty`. For comparison, Table 6.8 presents the number of instructions in the handler without interprocedural slices. The additional constraint limits the number of instructions in the handler. Benchmarks `crafty`, `gap`, and `gcc` have 5 to 10 times smaller handlers than in the previous case.

Table 6.9 presents the fraction of instructions that a handler contributes to a speculative thread. (By default, the handler is assumed to have interprocedural slices in all of the evaluations in this chapter.) Figure 6.6 presents the fraction of a speculative thread's execution cycles spent executing the handler. On average, the handler introduces 8% to 36% instructions in a speculative thread which take 5% to 15% of the execution time. In the worst case, roughly 30% of the thread's execution cycles are spent in the handler for many

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Mean	0.07	0.32	0.38	0.18	0.18	0.40	0.22	0.12	0.13	0.27	0.20
Std Dev	0.05	0.28	0.32	0.20	0.11	0.19	0.23	0.15	0.16	0.17	0.18
90-%ile	0.15	0.76	0.85	0.47	0.31	0.55	0.56	0.31	0.33	0.52	0.38
Max	0.21	0.94	0.92	0.63	0.37	0.59	0.90	0.61	0.57	0.62	0.50

Table 6.9: Ratio of the number of instructions executed by the handler in a speculative thread

benchmarks. For `crafty`, this is over 55%, because the handlers contribute up to 94% of instructions to a speculative thread. This is followed by benchmarks `gap` and `parser` also with over 90% of instructions from the handler.

A handler for a speculative thread is generated by the software infrastructure. Therefore, the software is responsible for determining the optimal number of instructions that must be in the handler to minimize overheads and maximize performance benefits. For example, results presented in this subsection can be incorporated into the handler generation process to determine how the handler can be adjusted, i.e., determining whether to include more instructions in it and possibly increase the separation of the speculative execution from the call site, or make it shorter, to minimize the overheads.

6.5.4 Triggers

In this subsection, I discuss trigger points for call sites chosen for PD, their sensitivity to different input sets, and results related to the evaluation of triggers. First, Figures 6.7 and 6.8 present the average number of trigger points (along with the standard deviation, 90-th percentile, and maximum). The handler used in Figure 6.7 contains interprocedural slices; those in Figure 6.8 do not. For the results presented, a speculative thread whose trigger site is also the call site, i.e., a speculative thread forked when the previous outstanding thread for that call site is committed, is considered to have no trigger point, and accounted as zero.

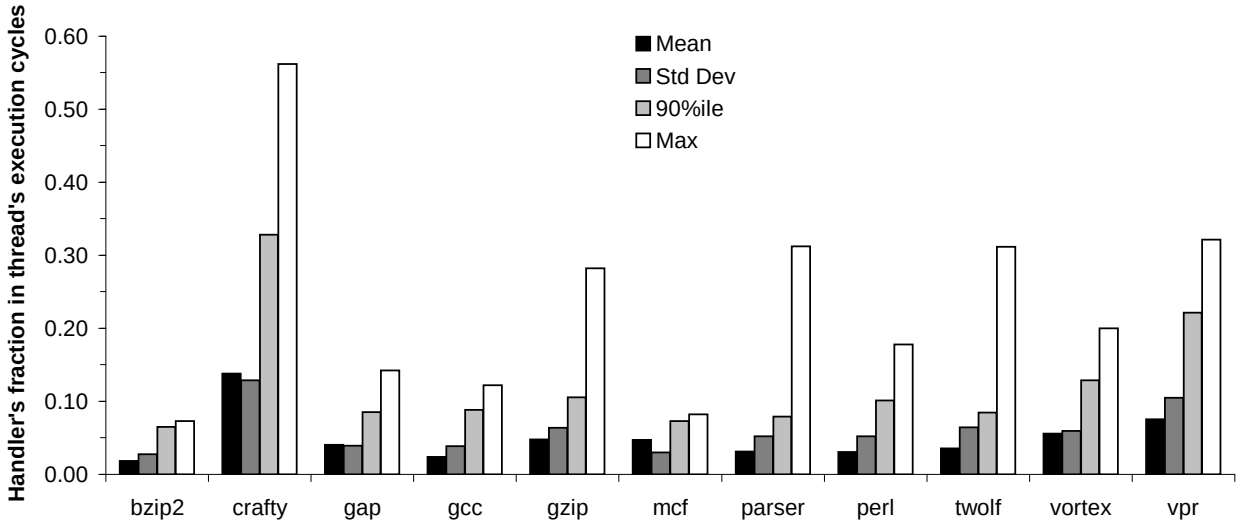


Figure 6.6: Fraction of speculative thread's execution cycles in the handler

The number of trigger points with interprocedural slices is more than the ones without them. Interprocedural slice based handlers extend for more number of instructions, usually require more program state and, therefore, increase the number of trigger points. For the benchmarks evaluated, the average number of trigger points is between one and two. This implies that triggers for a call site, if optimized, may be incorporated statically, instead of using the dynamic evaluation support discussed in Section 5.6.

For `gap`, the average is less than one because many call sites having no trigger points, as many speculative threads are forked when a previous outstanding speculative thread for that call site is committed. Benchmark `crafty` has few hundred call sites for PD, unlike many other benchmarks. The large size of the handlers for the call sites (discussed in Section 6.5.3) introduces many more dependencies with the program. This results in the worst case of six trigger points with interprocedural slices, and five without. Benchmark `vortex`, has over nine trigger points because of one notable programming characteristic. The program code extensively uses macros to create specialized methods from generic implementations (several hundred are created from few generic methods). Macros are processed by the compiler toolchain's preprocessor and, when compiled, the binary only has the generic method calls. Trigger points are generated for the generic versions of methods and not for the specialized methods.

Table 6.10 presents the number of trigger condition code registers needed for these benchmarks. These are set or reset when the program commits an instruction at a program counter that is registered with the trigger evaluation unit. The registers are further used for evaluating the trigger. The number of registers required is dependent on the number of call sites chosen for PD based execution and the number of trigger points for each of the chosen call sites. Based on these, benchmarks `vortex`, `crafty` and `parser` require the maximum number of trigger condition code registers because of the number of call sites selected for PD (refer Table 6.5). While on the other hand, `mcf` requires only seven registers.

The next set of rows in the table presents the hits and number of false positives to the Bloom filter, BF, in the trigger evaluation unit (refer Section 5.6). Program counters used in the predicates of triggers are registered with the Bloom filter, BF. The program counters of committed instructions are passed through the BF to determine if further action, i.e., searching and setting the trigger condition code register, is required.

The second row in the table (titled Hits), lists the fraction of program counters of committed instructions that hit in the BF. A hit indicates that the program counter may be registered; there may be false positives.

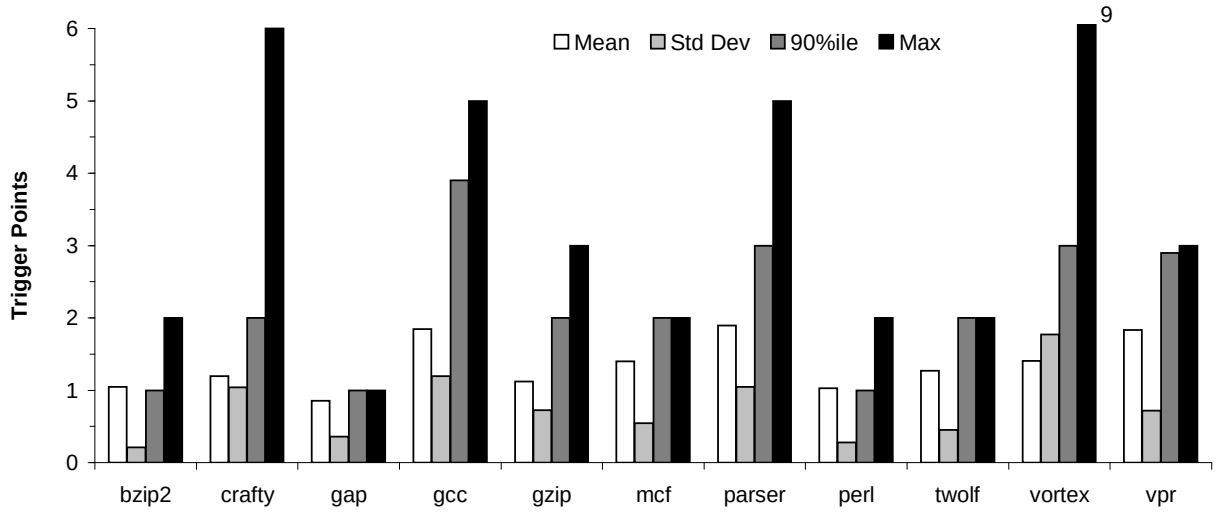


Figure 6.7: Trigger points with interprocedural slices in a handler

It is necessary to minimize the number of false positives so that the BF does not return a hit for a program counter that is not registered. Several different configurations are therefore presented next, to determine the optimal number of hash buckets needed and the hashing scheme that transforms program counter into bucket index.

The third row in the table (titled, 8Kb) presents the number of false positives with eight kilobits of entries in the filter. The hashing used is the XOR of lower 13 bits with bits 16 to 23 of the instruction's program counter.

The fourth row (titled, 16Kb) presents false positives for 16 kilobit entries with the hashing function based on the XOR of lower 14 bits with bits 16 to 23 of the instruction's program counter. The number of false positives is significantly reduced because of lesser collisions due to more number of bits available in the filter.

Benchmark `crafty` has the maximum number of false positives. 75% of the hits with 8Kb filter are false positives, which is significantly reduced with 16 Kb filter (to 39%). Benchmark `parser` has 50% reduction, `twolf` has 75% reduction, and `vortex` has 80% reduction in false positives with the larger filter.

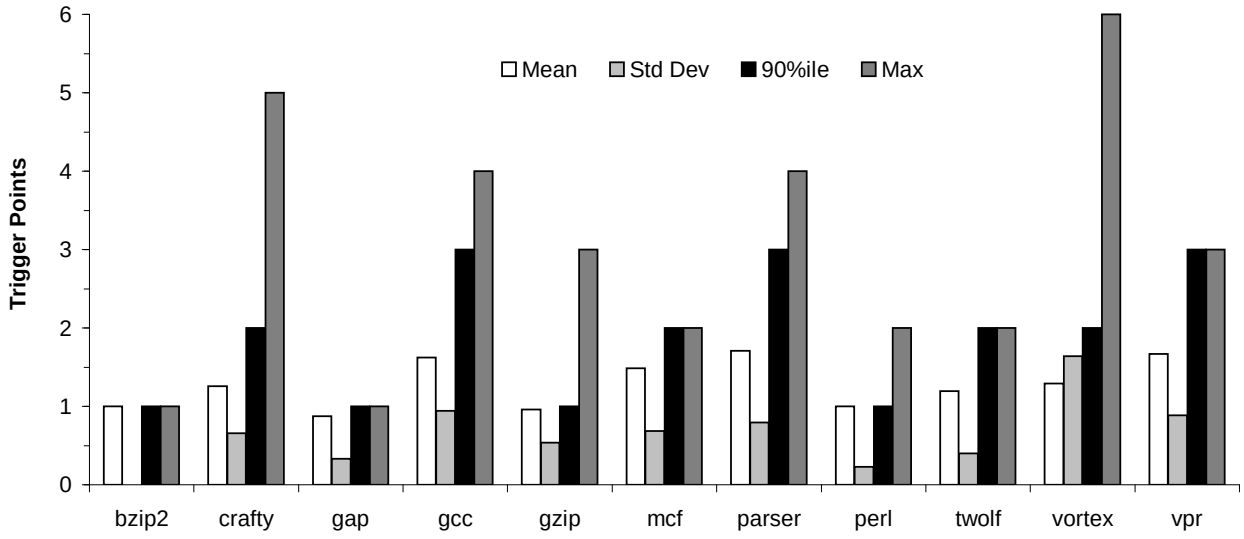


Figure 6.8: Trigger points assuming no interprocedural slices in a handler

	bzip2	**crafty**	**gap**	**gcc**	**gzip**	**mcf**	**parser**	**perl**	**twolf**	**vortex**	**vpr**
Registers Needed	24	132	41	39	28	7	122	41	33	147	22
Fraction Hits	0.02	0.02	0.02	0.01	0.02	0.03	0.02	0.02	0.02	0.02	0.04
8Kb	0.00	0.75	0.01	0.08	0.00	0.00	0.08	0.08	0.12	0.22	0.00
16Kb	0.00	0.39	0.00	0.05	0.00	0.00	0.04	0.03	0.03	0.04	0.00

Table 6.10: Statistics related to the trigger evaluation unit: number of trigger condition code registers, hits and false positives in the Bloom filter

Benchmark	**Input used**
bzip2	Reference input, input.graphic
crafty	Reference input
gap	Reference input
gcc	Reference input, 166.i
gzip	Reference input, input.graphic
mcf	Reference input
parser	Reference input
perl	Reference input, diffmail
twolf	Reference input
vortex	Reference input, lendian1.raw
vpr	Reference input, placement phase

Table 6.11: Input set used for trigger point sensitivity study

bzip2	**crafty**	**gap**	**gcc**	**gzip**	**mcf**	**parser**	**perl**	**twolf**	**vortex**	**vpr**
0.97	0.61	0.66	0.90	0.87	1.00	0.71	0.82	0.85	0.62	0.86

Table 6.12: Fraction of common trigger points with different input sets

6.5.5 Sensitivity of trigger points

Unlike identification of call sites for PD based execution and generation of handlers for these call sites, the identification of triggers relies extensively on profile information and may be sensitive to the behavior of the program. (Even though static information from the compiler, as described in Section 4.3, can help reduce this.) A poor choice of triggers can increase mis-speculations in the system and result in wasted execution resources. In this subsection, I study the sensitivity of trigger points, which are used to identify triggers, to different input sets.

An important requirement for collecting profile data for the generation of triggers is to achieve good coverage of the control flow in the program. This is dependent on the program and its behavior with different inputs. For example, a program may have two different set of inputs each executing different set of methods and/or instructions in control flow paths, such as `vpr` with placement and routing inputs. Some programs may not have different execution paths when input sets are scaled such as `gzip`, whereas some other programs may be dependent on the scale and the semantics of the input, for example `gcc`. Besides this, it is likely that profile information for a program which is collected and processed for triggers on one set of inputs, is used on another set of inputs.

In Section 6.5.4, the trigger points were obtained for the inputs presented in Table 6.3. To determine the sensitivity of these trigger points with different inputs and possibly, completely different phases, trigger points were collected for another set of inputs shown in Table 6.11. It is not known whether the phases that the benchmark programs are in match the phases for the runs with inputs shown in 6.3. Table 6.12 presents the fraction of trigger points that match between the two input sets. On average, 80% of the trigger points match between the two input sets. Benchmarks `crafty`, `gap`, and `vortex` have the lowest percentage of roughly 63, mainly because these programs are large, with complex control flow paths than the other evaluated benchmarks.

Even though it is desirable to have a high correlation of trigger points between different inputs, lower correlation in the benchmarks may not necessarily affect performance. The low fraction of matching trigger points is likely due to the methods executing in different phases of the program; control flow paths that were not exercised when run with train inputs may be used with reference inputs. Detailed program analysis is required to determine the nature of behavior of trigger points with changing inputs.

6.5.6 Speculative execution overhead

Figure 6.9 presents the overheads of separating a method's execution from its call site, and performing it on another processing core. On the Y axis is the ratio of the cycles used for speculative execution of a method in PD, over the cycles used by the method during sequential execution. The speculative execution of a method is measured by the number of cycles taken to execute the speculative thread in its entirety (i.e., including the handler). The following are the effects that a speculative thread in PD may have:

1. Negative effect due to execution of the handler.
2. Negative effect due to compromised locality of data because the method does not execute on the processing core that the program runs on.
3. Positive effect due to increase in the cache capacity. A PD based program uses the cache resources of many processing cores, which may lower capacity and conflict misses. Spatial locality between speculative threads that execute on a processing core may also be a positive effect.
4. Second order negative effects such as poor branch prediction accuracy because of the use of many processing cores, and the inability to train the predictors of all processing cores for an outcome of a branch.

The data in the figure can be divided into three categories: (i) Smaller methods have higher overheads. Therefore, the ratio is significantly greater than 1. (ii) Larger methods have lower overheads because negative effects such as additional cache misses are amortized. Therefore, the ratio is close to 1. (iii) Some large methods finish speculative execution faster than their counterpart in sequential execution. Therefore, the ratio less than 1.

Benchmark `mcf`, has ratio of 7 (7x overhead of speculative execution in PD) because speculative threads in the benchmark are small, and frequently miss in the cache. Any locality that may exist is lost due to threads executing on many processing cores. The worst case overhead for `mcf` is 14x for a speculative thread that executes a method of 10 instructions. All benchmarks have worst cases between 3 to 9x. This is expected since small methods chosen for PD have high overheads. The average numbers for benchmarks are, however, dominated by medium sized (100s of instructions) methods in benchmarks `vpr`, `twolf`, `gcc`,

and `parser` with overheads between 1 to 2x. Benchmark `vortex` has 1.5x overhead because of good cache locality, even though many speculative threads in the benchmark are less than a hundred instructions. `crafty` and `gap` call smaller methods more often than average sized methods. This results in poor locality, and high average overheads of 3.3x and 2.6x respectively. Since poor cache locality is one of the main reasons for the overheads, one way to lower the overheads is to implement data prefetching support in the hardware. The usefulness of a data prefetcher will greatly depend on when the prefetch requests can be issued, and when the data is available. Novel data prefetchers may be needed specifically to deal with short running speculative threads in PD.

Figure 6.10 presents further insights into the overheads of speculative execution with respect to the sizes of methods. The results are separated into methods that have, (i) less than 50 instructions, and (ii) greater than 50 instructions. Because `crafty` and `gap` speculatively execute many small methods, the averages presented in Figure 6.9 are dominated by these numbers (4.4x and 3.4x overhead for less than 50 instructions), rather than by methods with greater than 50 instructions (1.1x overhead). The explanation provided in the previous paragraph for `vortex` is also accurate for these results; the benchmark has very low overheads (1.23x). Noticeably, for all the benchmarks, methods with greater than 50 instructions have overheads between 1x to 1.9x.

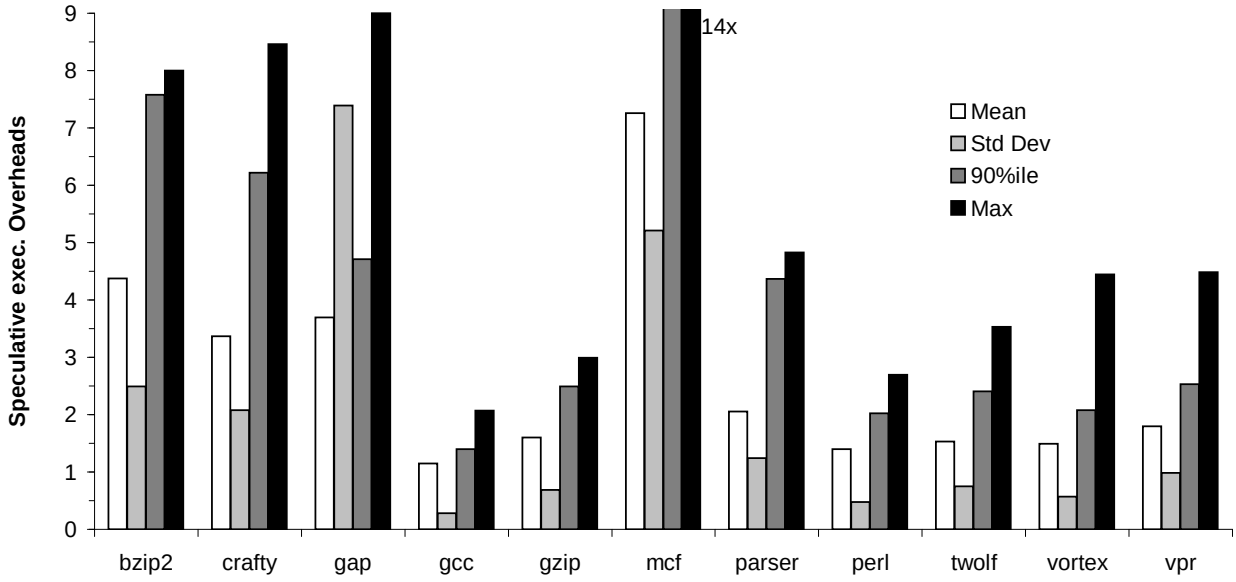


Figure 6.9: Overheads of speculative execution of a method in PD over sequential execution

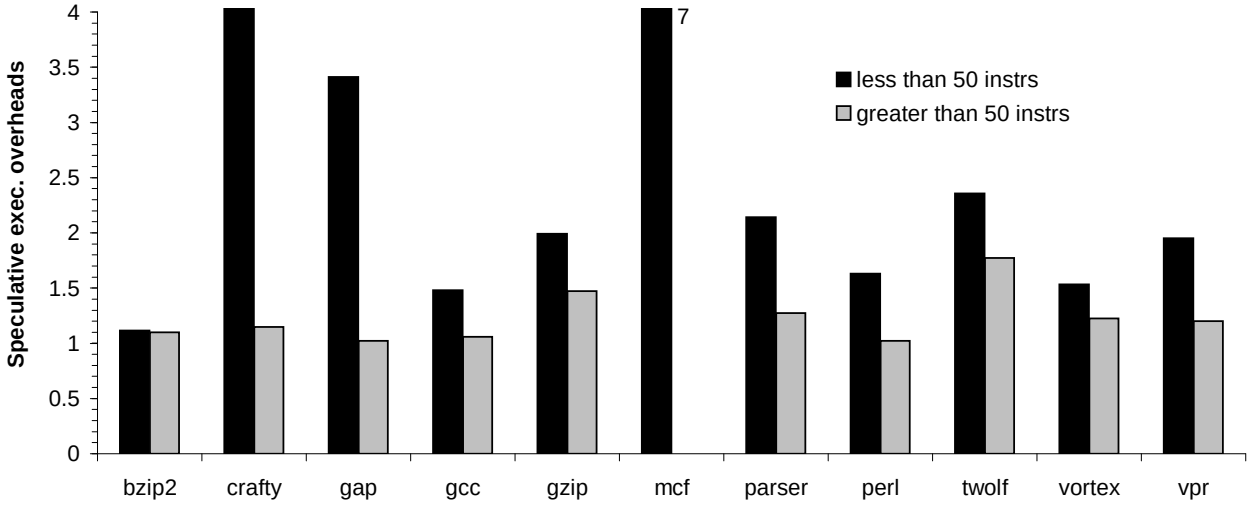


Figure 6.10: Average overheads of speculative execution in PD over sequential execution separated into two bars: methods with less than and greater than 50 instructions

bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
1.46	0.82	0.52	0.84	0.95	5.4	0.54	0.74	0.73	0.71	1.02

Table 6.13: Minimum of the overheads of speculative execution in PD over sequential execution

Table 6.13 presents the minimum overheads of a speculative thread in the evaluated benchmarks. Due to extra cache capacity, for several benchmarks the minimum is less than one, which implies that some speculative threads takes less time to execute than their counterparts in sequential execution.

6.5.7 Cache references

PD based execution, unlike sequential execution, uses many processing cores. The overheads associated with this usage are likely to increase the number of requests to the instruction and data caches of the processing cores, when compared to sequential execution. This increase is because of: (i) the references made by instructions in the handler during its execution and (ii) accesses performed by speculative threads that are aborted or squashed.

Figure 6.11 plots the fraction increase in requests in level one instruction and data caches during PD based execution over sequential execution. In almost all programs, except `mcf` and `gap`, the increase in instruction cache references is higher than that of data cache. On average, the references increase by 20% in the instruction cache and 16% in the data cache.

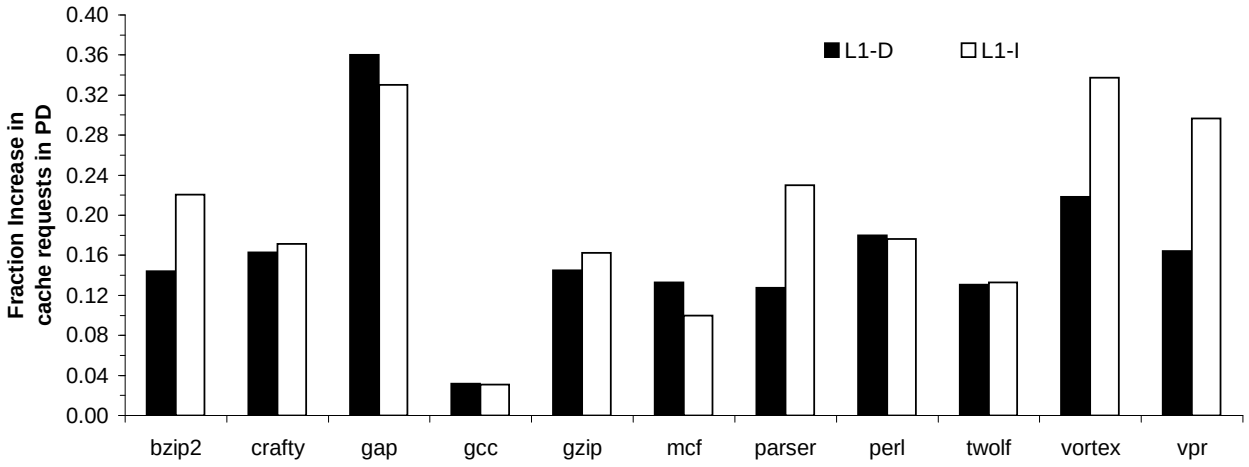


Figure 6.11: Fraction increase in cache requests with PD based execution over sequential execution in level one data and instruction caches.

For level one instruction and data caches, Figure 6.12 and 6.13 presents: (i) the miss rate with sequential execution, (ii) the miss rate on the program's processing core with PD based execution, and (iii) the cumulative miss rate of all speculative processing cores. One observation that stands out is the high miss rate in the data cache of speculative cores, which contributes to the high overheads of speculative execution. Benchmark `mcf` in particular has a 25% miss rate on speculative processing cores, which takes speculative threads seven times more execution cycles than in sequential execution (refer Section 6.5.6). On average, the miss rate on speculative processing cores is 11%. On the other hand, the miss rate on the non-speculative processing core is 3%. This is largely unchanged from the miss rate during sequential execution (average of 4%) because speculative threads are committed through the private cache and, therefore, will have a similar effect as sequential execution.

The observations from studying the instruction cache miss rates are different. The miss rate on the non-speculative processing cores are lower than the miss rate during sequential execution. This is because of the increase in total cache capacity and because the non-speculative processing cores must access and execute only instructions that are not executed by committed speculative threads. The miss rate of the speculative cores in PD is also lower for many benchmarks (except `gap`, `bzip2`, `parser`, and `twolf`) because of temporal locality exhibited when speculative threads of a method are scheduled in the same processing core. In addition, unlike data cache lines, instruction cache lines are not written to, and hence, not invalidated by the program.

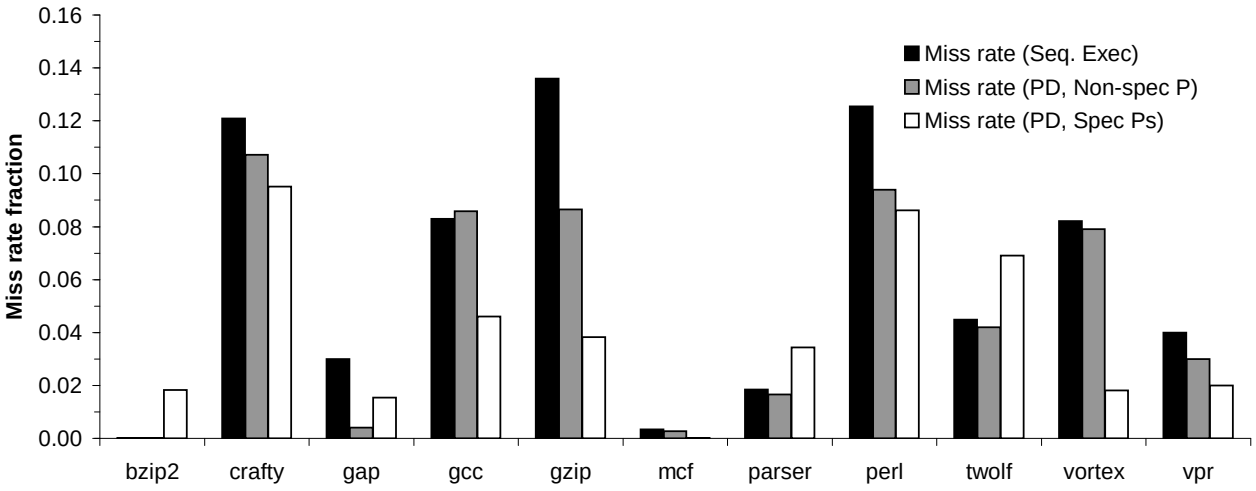


Figure 6.12: Miss rate per instruction for level one instruction cache in sequential execution (label: Miss rate (Seq. Exec)), non-speculative processing core in PD based execution (label: Miss rate (PD, Non-spec P)), and on speculative processing cores in PD based execution (label: Miss rate (PD, Spec Ps))

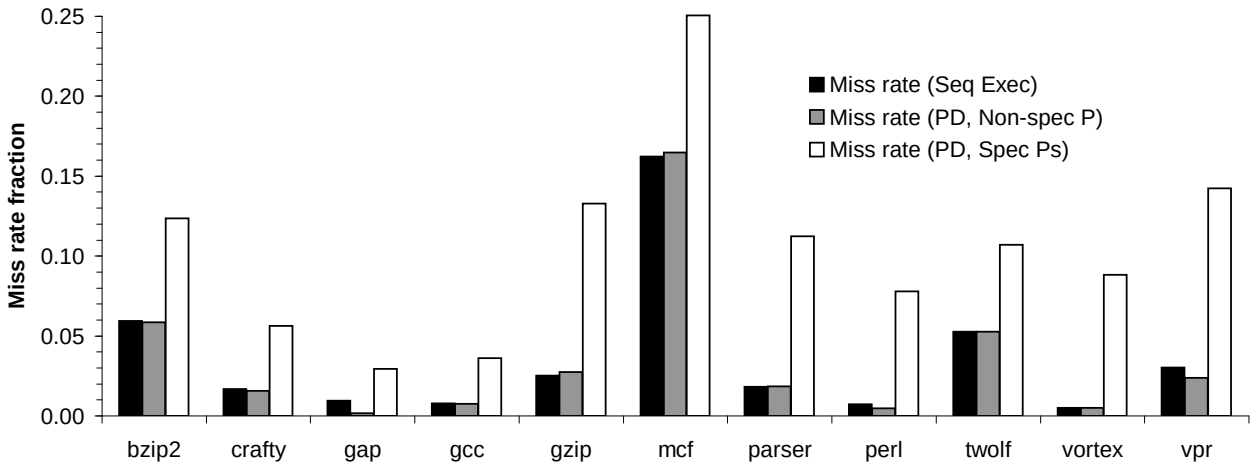


Figure 6.13: Miss rate per instruction for level one data cache in sequential execution (label: Miss rate (Seq. Exec)), non-speculative processing core in PD based execution (label: Miss rate (PD, Non-spec P)), and on speculative processing cores in PD based execution (label: Miss rate (PD, Spec Ps))

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Misprediction fraction in											
Seq. program	0.05	0.07	0.01	0.05	0.08	0.05	0.06	0.04	0.14	0.02	0.08
Cumulative misprediction fraction on processing cores executing speculative threads in											
PD program	0.13	0.09	0.04	0.04	0.13	0.09	0.10	0.06	0.15	0.04	0.10

Table 6.14: Fraction of branches mispredicted in sequential program and cumulative fraction of branches mispredicted on processing cores executing speculative threads during PD based execution

6.5.8 Branch mispredictions

Section 6.5.6 discussed several factors that contribute to the overheads of executing a speculative thread. One of the factors is the overheads introduced by poor branch prediction in an out-of-order superscalar processor pipeline. For example, the decision of a branch in a speculative thread is used to train the predictor table of the processing core executing the thread. Another execution of the same speculative thread, if performed on a different processing core, will not benefit from the trained predictor table. Table 6.14 presents the fraction of branches mispredicted in a sequential program, and the cumulative fraction of branches mispredicted in all processing cores used for execution of speculative threads (which is the ratio of the total number of branches mispredicted over the total number of branches in all speculative threads). On average, the misprediction fraction in sequential program is 6%, which increases to 9% in a PD execution. All benchmarks except `gcc` have poor branch prediction accuracy in speculative threads. The prediction accuracy may be improved with an intelligent scheduling policy that schedules all speculative executions of a given method on one processing core.

6.5.9 Methods table

The average and maximum number of outstanding speculative threads in a system is presented in Figure 6.14, with their average occupancy cycles in Table 6.15. In these results, there is no limit to the number of entries in the methods table, to study what might be an appropriate number to achieve maximum benefits. The data does not include speculative threads that do not complete when the call site in the program is reached as they are used or committed immediately upon completion, and not placed in the methods table. The number of speculative threads outstanding is directly dependent on the number of call sites chosen for PD based execution. It is also proportional to the cycles elapsed between the completion and use of a

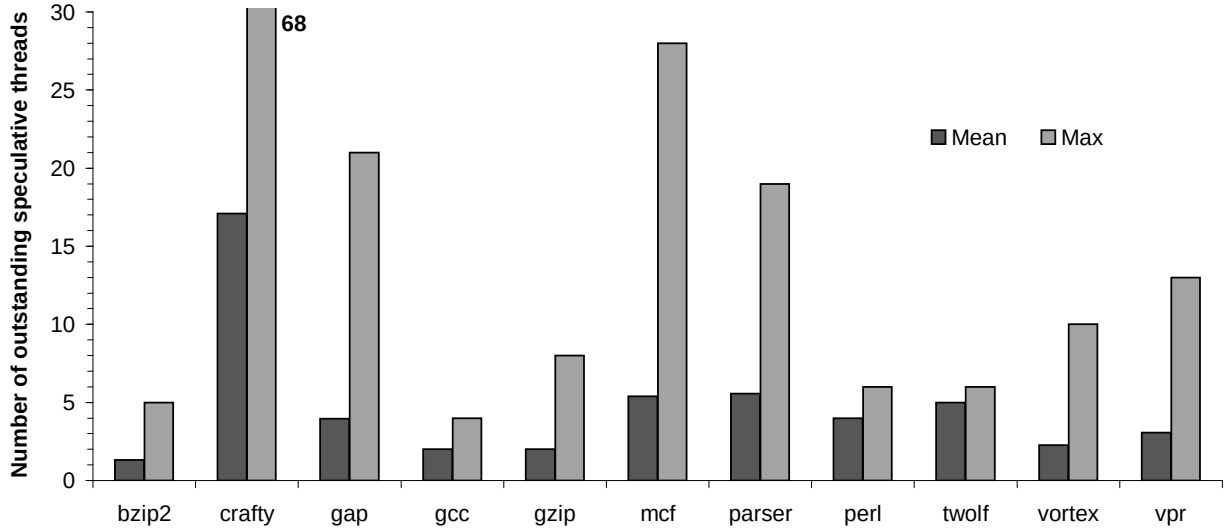


Figure 6.14: Number of oustanding speculative threads (excluding threads that stall the requestor)

bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
39.90	1295.22	281.97	213.13	61.36	709.30	318.06	134.27	331.38	108.85	489.42

Table 6.15: Average number of cycles outstanding speculative threads are held

speculative thread. This is shown in Table 6.15. For example, benchmark `crafty` has over hundred call sites for PD based execution and, therefore, has over 70 threads held for around a thousand cycles. On the other hand, compression programs `bzip2` and `gzip` have few call sites for speculative execution. The speculative threads are used 40 to 60 cycles after completion and, therefore, very few speculative threads are being held by the system. The performance benefits (presented in Section 6.5.13) are dependent not only on the number of outstanding threads presented here, but also the fraction of program's execution that they cover.

6.5.10 Read and write sets

Figures 6.15 and 6.16 presents the average, standard deviation, 90-th percentile, and maximum number of sub-blocks in the read and write sets of a speculative thread, respectively. (Sub-blocks are 16 bytes. For more details, see Table 6.1.) This data accounts for read and write sets of all threads in the system, including those that are in progress when the program reaches the corresponding call site.

The size of read and write sets depend on the method's size and computation performed. Since call sites of large methods were eliminated as candidates for PD, this limits the read and write sets of speculative

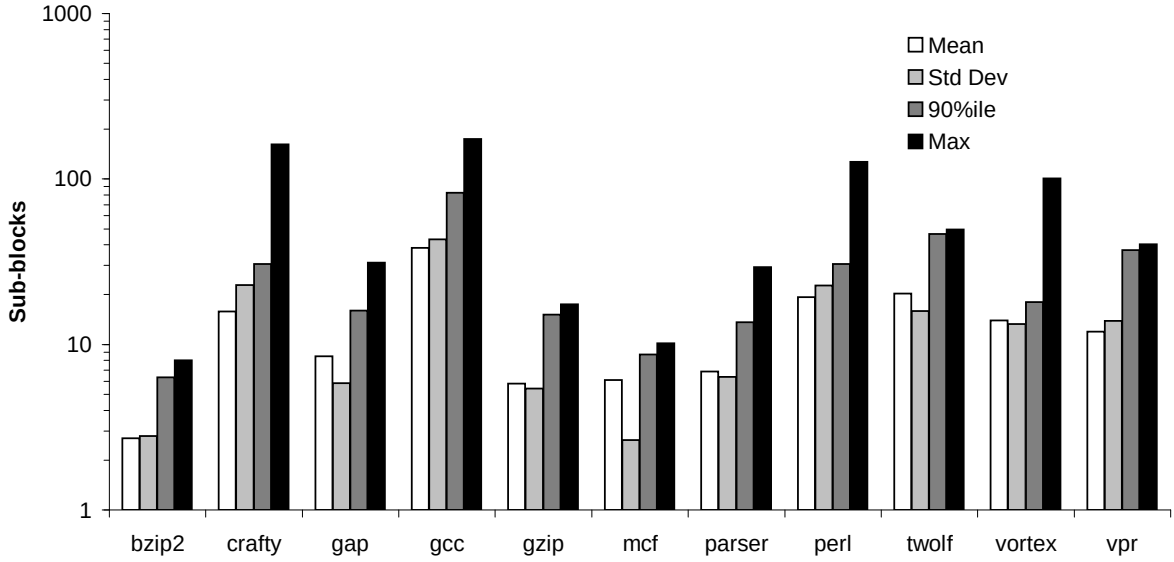


Figure 6.15: Read set size (in sub-blocks) of speculative threads

threads to roughly 20 and 30 sub-blocks, respectively, for most benchmarks. Benchmark `vortex` was earlier described in this section to have several small methods of few hundred instructions. Its mean read and write set sizes are 13 and 10 sub-blocks, respectively, lower than that of other benchmarks. It also has a low standard deviation, and 90-th percentile close to the mean.

Few large methods in some benchmarks contribute to the maximum of 100 sub-blocks in the read and write sets. The performance results (presented in Section 6.5.16) indicate that these methods contribute a sizable fraction to the improvements. Benchmark `crafty` has many small methods contributing to the low average, with some large methods contributing to the over 100 sub-blocks in the read set. Benchmarks `gcc` and `perl` have large read and write sets, making the chances of speculative execution with limited entries in the read and write set tables low. The smallest methods chosen for PD are in benchmark `mcf`. This is reflected in the read and write set sizes of 6 and 1 sub-block(s), respectively.

Studying the sizes of the read and write sets are important not only for deciding the size of several hardware structures but also for determining which methods to speculatively execute. First, the write set, which represents the dirty data of a speculative thread, is held in the private caches of the processing core during its execution. During a method's speculative execution, dirty data cannot be evicted from the cache and, in such a case, results in the termination of the thread. These methods must be eliminated as candidates for speculative execution. The size of the read and write sets also determines the size of the hardware

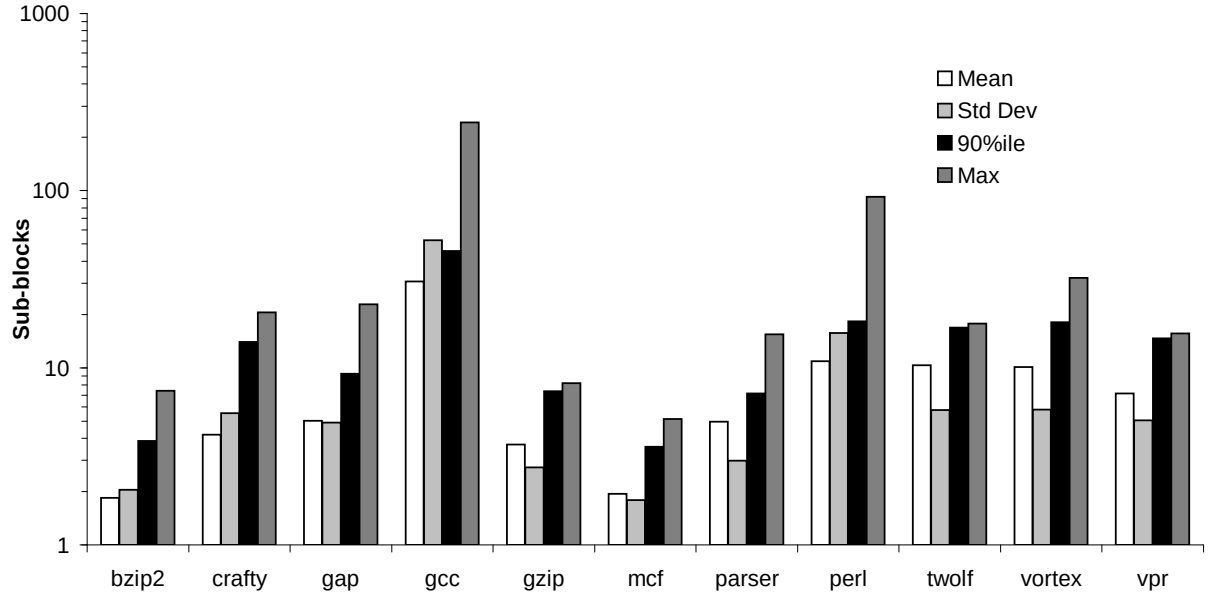


Figure 6.16: Write set size (in sub-blocks) of speculative threads

structures such as the speculative tag unit and execution buffer pool. The speculative tag unit must be sized to hold the tags of read and write sets during the speculative execution. An overflow in the STU aborts the thread. On completion of a speculative thread, its read and write sets are moved to the read and write set tables. If these tables have a fixed number of entries per thread, overflow either aborts the thread, or places the remaining tags (and data) in the overflow table, which may be undesirable for efficiency. The invalidation cache is populated with the read and write set tags and must also be sized to have minimal overflows. The sizing of the structures will be further discussed in Section 6.5.13 on performance improvements with PD.

6.5.11 Invalidation cache

The invalidation cache is introduced in this implementation for the sole purpose of efficiently determining if the address of a store committed by the program is present in the read or write sets of the outstanding speculative threads. Without the invalidation cache, every entry in the read and write set tables for all the speculative threads must be searched, which is likely to be inefficient, even though the operation is not on the program's critical path. Thus, additional storage and logic in the form of invalidation cache is used to minimize the overheads. (Alternatives to the invalidation cache are private caches as discussed in Section 5.7 or signatures proposed in Bulk [32].)

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Fraction sets overflow	0.00	0.00	0.14	0.38	0.01	0.23	0.21	0.00	0.91	0.03	0.04
# overflows	0.0	2.5	8	11	1.8	4	3.8	0.0	11.4	3.2	2.8

Table 6.16: Fraction of sets that overflow and the number of overflows for an invalidation cache of 1024 sets, 8-way

The invalidation cache organization is discussed in Section 5.3. The size and the associativity must be chosen to minimize the number of overflows per set. Table 6.16 presents the fraction of sets in the invalidation cache that overflow, and the average number of overflows in these sets. The invalidation cache has 1024 sets, each 8-way associative. The worst case behavior occurs in benchmark `twolf`; 91% of the sets overflow and requires an additional average of 11.4 entries. 14% to 38% of the sets overflow in benchmarks `gcc`, `parser`, `mcf`, and `gap`. They require an additional 1.8 to 11 entries per set.

6.5.12 Utilization

Figure 6.17 plots the utilization fraction for the seven processing cores that are used for speculative execution. The utilization fraction is the ratio of the cycles spent by a processing core for speculative execution, over the cycles spent by the non-speculative processing core to execute the PD based program. The graph only accounts for the execution of speculative threads that are committed or used by another speculative thread. The processing core spends the rest of the cycles executing speculative threads that are aborted or squashed, and stalling when the write and read sets of a completed speculative thread are transferred from its private caches to the execution buffer pool.

The utilization of processing cores may be used to determine the plausible performance benefits. However, there is no direct correlation as the performance improvements will depend on the overheads of speculative execution and the extent of overlap between the threads in the system. Based on the data from the figure, the highest cumulative utilization (Cumulative utilization is obtained by adding the utilization fraction on all seven processing cores presented in the graph and multiplying by 100. The highest possible cumulative utilization can be 700%.) is from benchmarks `parser` (185%), `gap` (176%), `crafty` (157%), `vortex` (197%), followed by `vpr` (137%), `twolf` (74%), `perl` (63%), and `mcf` (103%), and finally `gcc` (17%) and `bzip2` (52%). These numbers are representative of the discussion presented in this

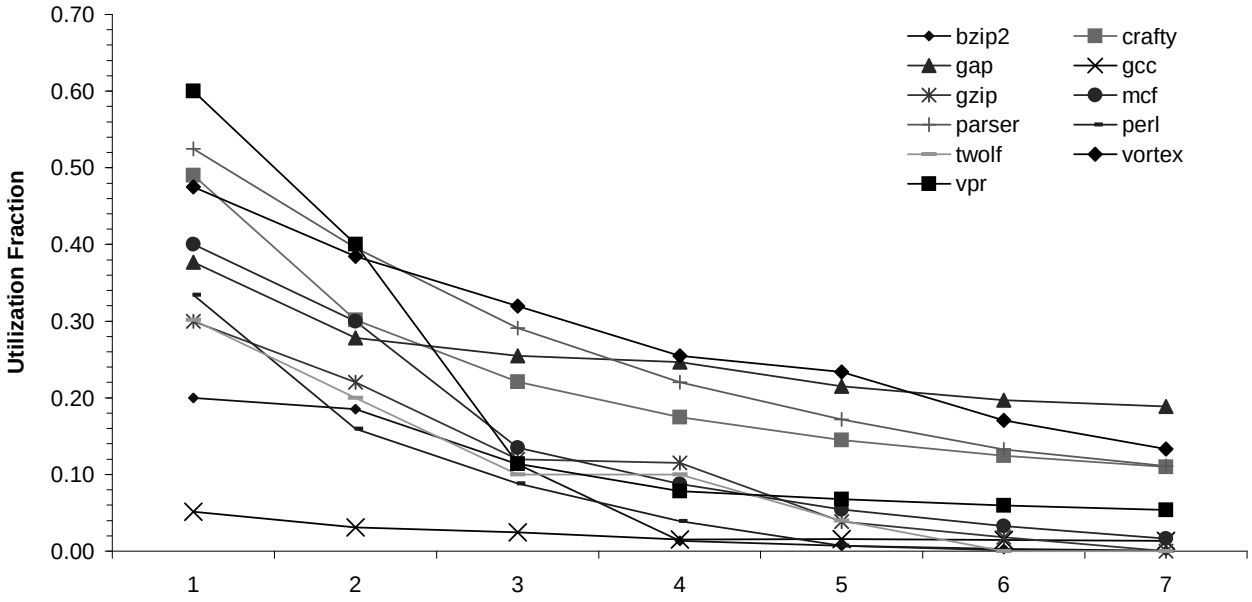


Figure 6.17: Ratio of the cycles spent by a processing core for speculative execution over the cycles spent by the non-speculative processing core in PD based execution. The graph presents the ratio for the seven processing cores used for speculative execution in the system.

section so far. Benchmarks `crafty`, `vortex`, `gap`, and `parser` were observed to have the maximum opportunities (refer Section 6.5.1) and, therefore, have high utilization. On the other hand, methods chosen from benchmarks `mcf`, `bzip2` did not cover significant fraction of program's execution time (refer Table 6.5) and, therefore, utilize only a small fraction of the cycles for speculative execution.

6.5.13 Stall cycles

During PD based execution, when a call site is reached, outstanding speculative threads in the methods table and any ongoing speculative threads in other processing cores are searched for a match. If the call site matches with an ongoing speculative thread, the requestor stalls until the thread finishes execution, and then initiates the operations to commit the thread. For performance reasons, it is best that the requestor wait only if the number of cycles it is going to stall for the speculative thread to complete is less than the cycles the requestor would take to execute the method. Otherwise, the speculative thread slows down the requestor, an undesirable effect. While discussing the software infrastructure in Chapter 4, it was suggested that the call sites chosen for PD be refined based on the feedback from a PD based execution. This is an instance in which if it is observed that PD based execution for a call site hurts program's performance, that call site

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Fraction cycles stalled	0.02	0.03	0.01	0.01	0.16	0.00	0.14	0.20	0.11	0.28	0.18
Fraction threads stall the program	0.28	0.13	0.04	0.74	0.66	0.00	0.25	0.53	0.27	0.32	0.22

Table 6.17: Fraction of cycles a requestor stalls for a speculative thread to complete, and the fraction of threads that stall the requestor

must not be speculated. I performed this step by hand, observing results from simulations, and refining the set of methods that can be used for PD.

Table 6.17 presents fraction of execution cycles that a PD based program stalls waiting for the speculative thread to complete. It ranges for 0% for `mcf` to 28% for `vortex`, with an average of 10% of cycles stalled in all the benchmarks. During this time, the requestor stalls (other speculative executions may be ongoing). The fraction is dependent on two aspects: (i) the size of the speculative threads, which depends on the methods chosen for PD and the program's characteristics, and (ii) the separation between the trigger site and call site for a speculative thread achieved by the corresponding trigger and handler. The numbers presented in the table may be taken into account to determine the appropriate trigger site for a speculative thread.

The second row in the table lists the fraction of used speculative threads whose corresponding call sites are reached by the requestor before their completion. A large fraction indicates that majority of the speculative threads are stalling the requestor, often the case if speculating on large methods. Benchmarks `gcc` and `perl` are dominated by large methods, and 74% and 53% of the threads stall the requestor. Performance improvements with PD are likely to be insignificant for these benchmarks. A program that has small methods for speculative execution, but significant fraction of threads stalling the requestor, implies high overheads of speculative execution and/or speculative threads forked not well before their respective call sites. Benchmark `gzip`, with 66% of the threads stalling the requestor, is an example of such a case.

Finally, among the threads that stall the requestor, Figure 6.18 presents the fraction of the thread's execution time that the requestor waits. Minimizing these numbers may be beneficial to the requestor, as it represents the portion of a speculative thread in its critical path. These numbers, along with the fraction of threads that stall the program presented in Table 6.17, determine the performance benefits of speculative threads. For benchmarks `gcc` and `perl`, 74% and 53% of speculative threads stall the requestor for roughly 65% of their execution time.

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Fraction cycles	0.09	0.22	0.10	0.06	0.22	0.05	0.38	0.07	0.17	0.22	0.13
Fraction threads utilized	0.09	0.20	0.29	0.50	0.15	0.10	0.19	0.30	0.70	0.90	0.33

Table 6.18: Fraction of cycles wasted by speculative threads that are squashed, or aborted by the handler

bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
0	0	0.06	0	0	0	0	0	0.14	0	0

Table 6.19: Fraction of speculative threads aborted because of eviction of a speculative cache line

6.5.14 Wasted execution cycles

This subsection discusses the number of wasted cycles performing speculative execution, which is either due to handler aborting without calling the speculative execution of a method or, a speculative execution that is discarded due to a dependence violation. The cycles spent executing the handler in a speculative thread that is later committed or used is not considered as wasted execution cycles.

The first row in Table 6.18 presents the ratio of the execution cycles on the speculative processing cores that is discarded, over the cycles for PD based execution. On average, 15% of execution cycles are wasted, with benchmarks that achieve higher performance benefits (discussed later in Section 6.5.13) having higher fraction of wasted cycles. These are `crafty` with 22% of cycles, `parser` with 38%, and `vortex` with

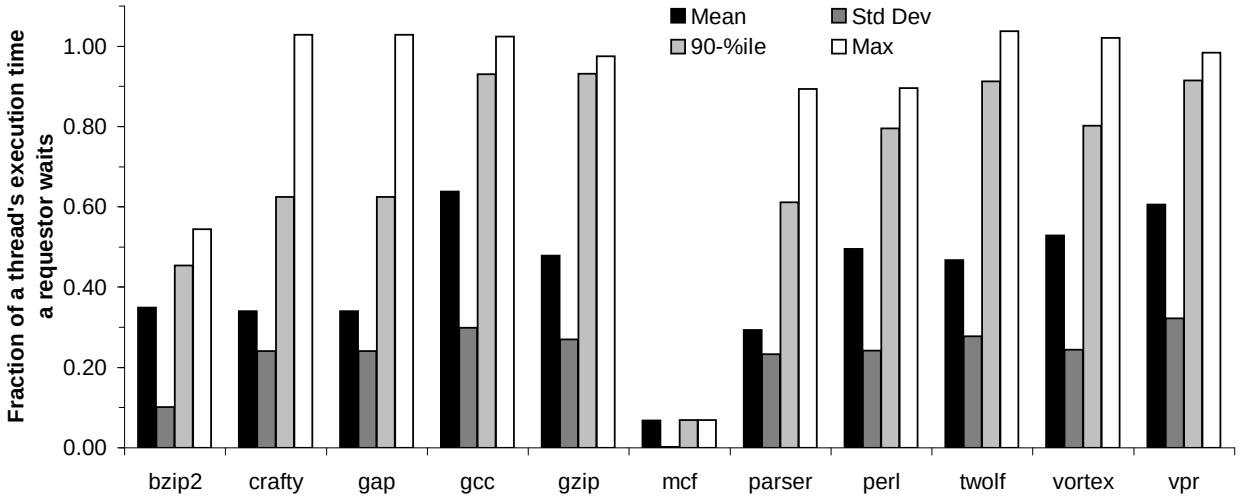


Figure 6.18: Fraction of speculative thread's execution that the requestor stalls (only among threads that stall the requestor)

22%. Benchmarks with low utilization of processing cores for speculative execution (refer Section 6.5.12) such as `bzip2`, `mcf`, and `gcc` have only 9%, 5%, and 6% of cycles wasted, respectively.

Of the cycles wasted, several speculative threads may be forked, but aborted by the handler. The fraction of speculative threads that actually begin speculative execution of a method is presented in the second row of the table. There is a large variance across the benchmarks, ranging from 9% to 90%, with an average of 34%. The fraction of threads aborted strictly depends on the branches included for evaluation in the handler. If the call site is control dependent on one or more branches which when executed rarely takes the path to the call site, including these branches in the handler will therefore result in more frequent aborts. However, the cycles to speculatively execute an handler that aborts is lower than speculatively executing a method that will be squashed.

The fraction of speculative threads that are aborted when a speculative cache line is evicted (refer Section 5.2 for more details) is presented in Table 6.19. 6% and 14% of the threads in `gap` and `twolf` are aborted. The rest of the benchmark programs does not have any speculative cache line evictions.

6.5.15 Performance

This section has so far covered several results that can be broadly divided into three categories: (i) methods chosen for PD and their characteristics, such as size, number of call sites, and execution time, (ii) potential for performance improvements with the chosen call sites, (iii) results on software components of PD namely, handlers and triggers and, (iv) hardware implementation results which consisted of overheads of speculative threads, cache miss rates, mis-speculations, invalidation cache overflows, stall cycles, and processing core utilization.

This subsection discusses the performance benefits of the implementation. I use the hardware implementation listed in Table 6.1 and the “Base” group of parameters in Table 6.20. Speculative threads are scheduled every 50 cycles. Trigger evaluation code is not generated by the software infrastructure, but a 10 cycle execution latency is modeled for evaluating a trigger and determining if it has fired. 5 cycles are used to communicate the handler program counter and stack pointers to the speculative processing core. The system is assumed to have no bandwidth limitations with no overheads for transferring write and read sets to and from the processing cores. The invalidation cache and trigger evaluation unit sizes are listed in

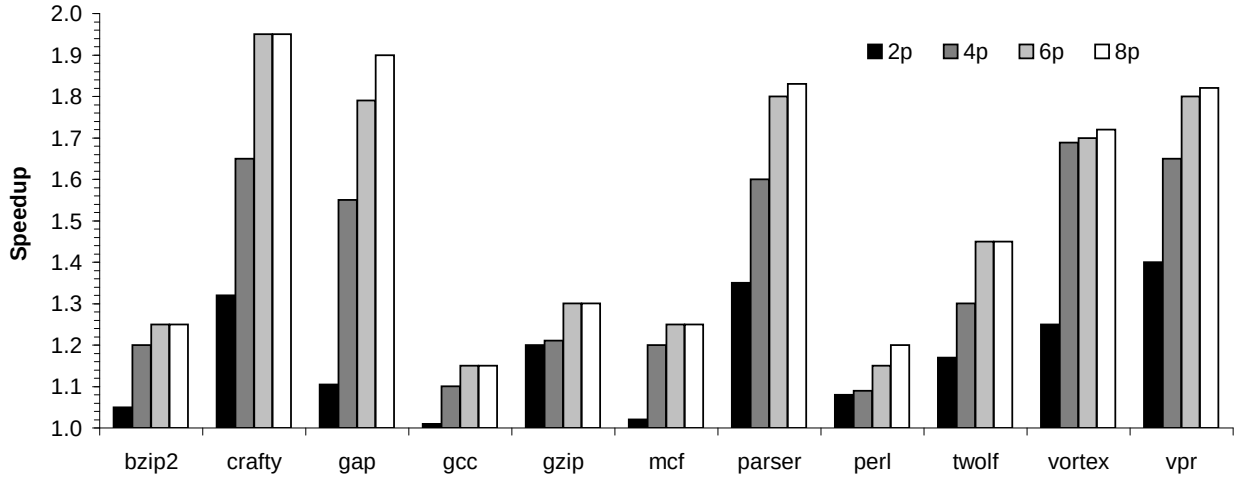


Figure 6.19: Performance benefits from PD with two, four, six, and eight processing cores

the table. The methods table and read/write set tables are assumed to be large enough to accommodate all speculative threads and their respective read and write sets.

Figure 6.19 presents the performance benefits, with respect to sequential execution, from PD on the evaluated benchmarks for two, four, six, and eight processing cores.[8] Speedups range from 1.1.x to 2x, with a harmonic mean of 1.47x. The results follow the trend that has been set up by other results studied so far. Compression programs `gzip` and `bzip2` were expected to have minimal benefits because of very few opportunities and their tightly coupled algorithms; they achieve 1.2x performance improvement in this evaluation. Benchmark `twolf` achieves 1.4x speedup, while `gap`, `crafty`, `vpr`, `parser`, and `vortex` are the best performing benchmarks, averaging 1.8x with eight processors. `crafty` has several speculative threads because of the nature of the program, while with `gap` and `vpr`, the initialization, setup, and lookup of structures used by the programs result in the performance benefits. Benchmark `gcc` and `perl` show lot of potential opportunities, but do not result in any improvements, averaging 1.1x. Many methods in the two programs are speculatively executed with very little overlap with other threads or the program. Benchmark `mcf` is crippled by few opportunities and high speculative execution overheads. In all the benchmarks, eight processing cores are rarely used by speculative threads, as was evident in the utilization study presented in Figure 6.17. Maximum benefits are achieved with four to six processing cores.

[8]The sequential program is compiled with the flags specified in Section 6.3. Unlike PD based execution, the sequential program does not use any profile information. Profile guided optimization was been found to improve performance of the sequential execution of SPEC CPU2000 integer benchmarks compiled with the Intel C compiler, a state-of-the-art optimization compiler, by 7% on an Intel Pentium 4 system. Profile guided optimizations may also be performed on a PD based application.

Base		
	Scheduling	First-come first-serve policy of scheduling speculative threads. Threads are scheduled every 50-cycles.
	Trigger evaluate	10 cycles fixed for all triggers
	Forking	5 cycles to communicate call site and handler program counter, stack and base pointers
	Invalidation cache	1024 sets, 8-way
	Trigger eval. unit	16Kbits Bloom filter. Upto 150 trigger condition code registers.
Execution buffer pool latency and bandwidth		
	Bandwidth & Latency	10 bytes per cycle; two cache lines per 12 cycles to/from read and write set tables; 3 tags per cycle when using a speculative thread.
PD hardware structures		
	Methods table	20 entries
	Spec. tag unit	50 entries
	Read set table	20 entries per thread
	Write set table	30 entries per thread

Table 6.20: Details of the simulated machine: Program Demultiplexing implementation parameters

6.5.16 Inorder forking

A novel aspect of PD that distinguishes it from prior proposals is the unordered forking of speculative threads. This subsection compares this over the program ordered or "inorder" forking models of previous speculative parallelization models. The inorder model of PD, referred to as Inorder PD, uses the PD implementation discussed in this dissertation so far, except for one critical restriction. Speculative threads are allowed to begin execution only after threads that will be committed earlier in the program have begun execution. This restriction ensures that the threads begin speculative execution in program order and, therefore, will be committed in that same order. The Inorder PD implementation however, has two notable aspects that differ from prior speculative parallelization proposals: (i) Mis-speculations in Inorder PD do not propagate to other speculative threads and, therefore, only the thread that violated dependencies is squashed, and (ii) Inorder PD does not allow communication between speculative threads, because the implementation of PD does not support it. To deal with nested method calls which results in nested speculative threads, Inorder PD implements hierarchical tree ordered forking as suggested by Renau et al. [161]: the inner most method begins speculative execution before the outer methods.

Useful utilization of processing cores with Inorder PD is presented in Figure 6.20. The Y axis represents the ratio of the cycles spent on speculative executions by the seven processing cores that are eventually used

	bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
Number of threads	0.40	14.37	2.03	0.53	0.17	3.73	2.17	0.52	1.16	1.73	2.54
Cycles Held	98	1374	520	885	78	626	442	217	458	227	408

Table 6.21: Number of speculative threads outstanding in Inorder PD and the number of cycles they are held before being committed

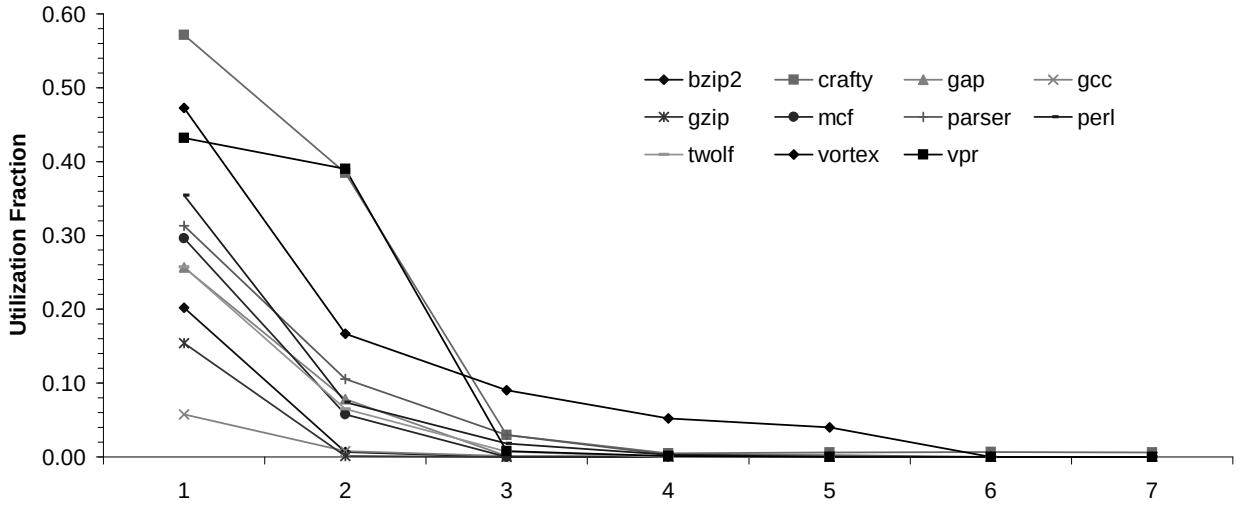


Figure 6.20: Ratio of the cycles utilized for useful speculative executions over the cycles for Inorder PD based execution for the seven processing cores

by the program, over the cycles for Inorder PD based execution. The lack of utilization of processing cores three to seven implies that the forking model is severely restrictive, and is unable to find speculative threads. Compare this with the useful utilization with PD in Figure 6.17. Benchmark `crafty` has the highest cumulative utilization (defined as the sum of utilization fraction for the seven processing cores, multiplied by 100) at 101%, followed by benchmarks `vpr` and `vortex` at 83%. The rest of the benchmarks cover 7% to 45% of execution cycles. Notably, benchmark `gap` and `parser`, which utilizes up to 185% and 176% of execution cycles (the top two among all evaluated benchmarks) in PD, with Inorder PD uses only 34% and 45% respectively (5th and 7th when sorted in increasing order of utilization among benchmarks with Inorder PD). The program ordered forking severely restricts the speculative threads that can be forked.

Performance is presented in Figure 6.21. The implementation parameters are the same as discussed in Section 6.5.13. As expected the lower utilization of the processing cores results in performance that does not match PD. Inorder PD performs well on `vortex` and `vpr` achieving 1.3x and 1.5x speedup. `crafty`'s

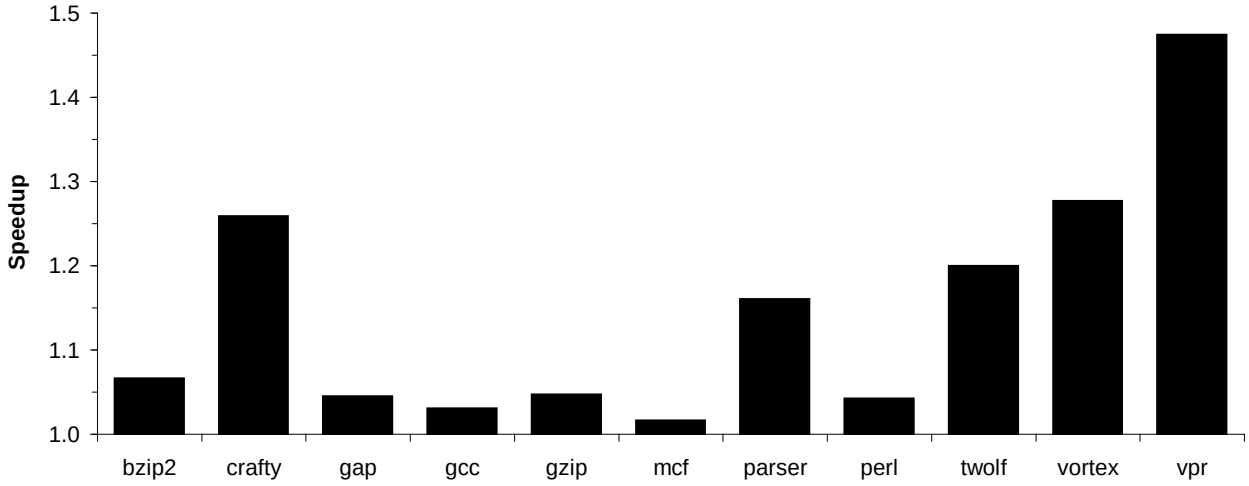


Figure 6.21: Performance benefits with Inorder PD

	bzip2	**crafty**	**gap**	**gcc**	**gzip**	**mcf**	**parser**	**perl**	**twolf**	**vortex**	**vpr**
Fraction Cycles	0.08	0.20	0.11	0.01	0.18	0.05	0.20	0.06	0.16	0.27	0.13

Table 6.22: Fraction of cycles wasted by speculative threads that are squashed or aborted in Inorder PD

high utilization results only in 1.3x speedup. The remaining benchmarks average 8% improvements, with an overall average of 1.15x.

The average number of speculative threads outstanding in the Inorder PD system, and the number of cycles they are held is presented in Table 6.21. The number of outstanding speculative threads is two times lower compared to PD. They are held on average for 56% more cycles than PD (refer Section 6.5.5). This is because of the program performing most of the execution in Inorder PD, which results in a thread being held for more cycles before the program can reach the corresponding call site. Table 6.22 presents the fraction of cycles spent in speculative executions that are discarded. For almost all benchmarks, the mis-speculation fraction is nearly equal or less than that of PD. On average, the Inorder PD implementation has 15% lower mis-speculations as compare to PD.

In summary, the two implementations, PD and Inorder PD, differ only in the ordering in which the speculative threads are forked. Inorder PD forks speculative threads in program order, whereas PD does not. The impact, in terms of utilization of processing cores, and the performance improvements are, however, vastly different. On average, Inorder PD improves performance by 20%, whereas PD achieves 60% improvements. The forking model of Inorder PD affects the ability to fork speculative threads to create

concurrency. Supporting communication of speculative data values between thread would allow a thread to be forked before its data dependencies are satisfied, unlike PD, and obtain them speculatively from other older (i.e., earlier in program order) speculative threads. This in turn, might improve the effectiveness of the forking model.

6.5.17 Performance with latency between execution buffer pool and processing cores

The performance benefits presented in Section 6.5.13 assumed unlimited bandwidth and no overheads to communicate between the execution buffer pool and the processing cores. This subsection presents a realistic implementation of PD. The hardware used for this evaluation additionally models the parameters listed in the latency and bandwidth group of Table 6.20.

In general, increasing the latency of communication to and from the execution buffer pool has the following effects: (i) Increase in the number of cycles a speculative processing core is held after the thread has finished execution. It is assumed that the processing core can be released only after all the data of a speculative thread has been transferred to the execution buffer pool. (ii) Increase in the number of cycles the program or a requesting speculative thread has to wait to obtain the write (and read set if the requestor is a speculative thread) set(s). It is assumed that the requestor stalls until all the tags of the write set (and read set, if the requestor is a speculative thread) are transferred. The write set data is transferred without the requestor stalling (more details in Section 5.5).

Figure 6.22 presents the performance benefits for this implementation. On average, performance improvement drops by 11% from the implementation in Section 6.5.13, with benchmarks `parser`, `vortex`, and `vpr` facing over 25% loss in improvements. On average, the benchmarks see improvements of 40% over the sequential execution.

6.5.18 Performance with limited hardware resources

This subsection evaluates performance with further restrictions to the implementation. The additional parameters used are listed in the PD hardware structures group in the table. For this implementation, the methods table can hold 20 speculative threads, the speculative tag unit has 50 entries, the read set table is fixed with 20 entries per speculative thread, and the write set table is fixed with 30 entries per speculative

thread. These numbers were determined based on the results discussed in previous subsections (Section 6.5.9 and 6.5.10), and chosen to cover the average of the 90-th percentiles presented.

Overflows in the methods table, speculative tag unit, read set table, or write set table, aborts the speculative thread. The fraction of threads that are aborted due to this restriction over the base case is presented in Table 6.23. This has no effect on benchmarks `bzip2`, `gzip` and `mcf` as expected (see results on read and write set sizes in Section 6.5.10). However, other benchmarks, in particular, `perl`, `vortex`, `vpr`, and `twolf` see 20% to 72% of the speculative threads aborted due to the hardware restrictions. Figure 6.23 presents how these restrictions affect performance improvements over the sequential execution. Performance improvements drop by an average of 15% from the previous case. Benchmark `twolf` has the worst effect with 38% drop in performance over PD with limited bandwidth. Performance for `crafty` and `gap` lowers by average of 24%, `vortex` and `vpr` by 10%. Benchmarks `gzip` and `mcf` do not have any impact as expected. Performance improvement for `mcf` lowers by 14% because 20 entries in the methods table is insufficient; upto 29 speculative threads can be outstanding during execution. (Refer Figure 6.14 for number of outstanding speculative threads.) The results clearly indicate that the performance improvements from PD is not only because of the small methods that fit in the execution buffer pool structures used in this implementation but also because of speculatively executing larger methods, even if they are not completely overlapped.

In summary, the choice and the sizes of hardware structures must be carefully deliberated to maximize benefits. However, a real implementation may have to deal with programs that cannot always be

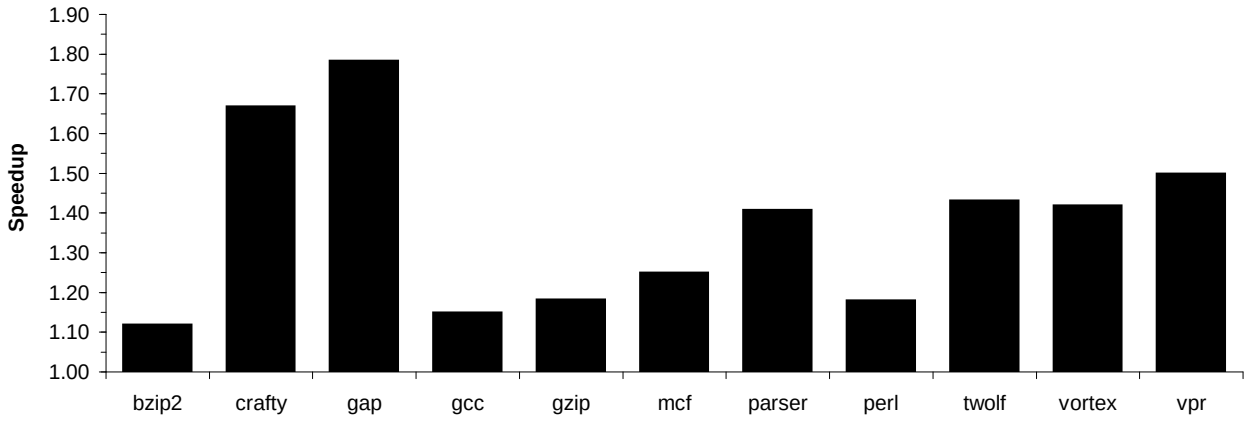


Figure 6.22: Performance benefits from PD with latencies modeled between execution buffer pool and processing cores

bzip2	crafty	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr
0.00	0.07	0.08	0.10	0.00	0.00	0.02	0.72	0.20	0.44	0.35

Table 6.23: Fraction of threads aborted because of limited hardware resources

benchmarked and profiled to optimally size the structures. It is also unrealistic to build a hardware with unlimited resources. Further research is needed to deal with cases that the hardware may not be able to speculatively execute successfully. The simplest option is to revert to sequential execution, a benefit of speculative parallelization. Another alternative is to let software perform the speculative execution at a higher cost.

6.6 Chapter summary

This chapter evaluated an implementation of PD. The experimental infrastructure consisted of a hardware simulator and software toolset based on the Intel x86 instruction set architecture. The evaluation was conducted on integer programs from the SPECCPU2000 suite. Several results were presented. First, opportunities, i.e., methods that form significant fraction of program's execution time were presented. Candidates for PD were chosen from these methods. The potential for PD based execution was presented by introducing the ratio of the number of cycles between the trigger site and the corresponding call site in the sequential program over the number of cycles to execute the method. Several different variants of handlers were generated, their length (number of instructions), impact on the potential, and the fraction of cycles that they contribute to the speculative thread were studied. Results on triggers included a study of the number of trigger points, their sensitivity to changing input sets, and sizing of the hardware structure, the

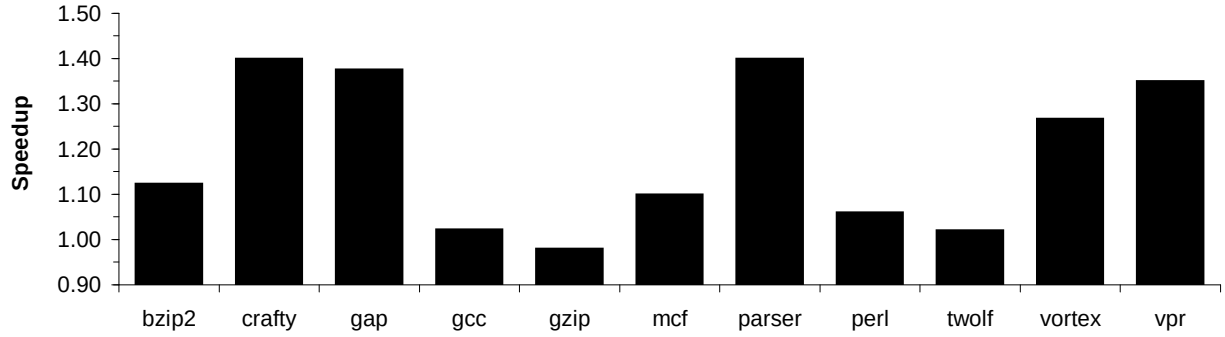


Figure 6.23: Performance benefits from PD with latencies modeled between execution buffer pool and processing cores, and with limited hardware resources

trigger evaluation unit. Then, with a PD based execution system, I discussed the overheads of speculative threads, read and write sets of speculative threads, utilization of processing cores for speculative execution, the number of outstanding speculative threads and their cycles between completion and use, the fraction of cycles the program is stalled for completion of speculative threads, and overflows in the invalidation cache. PD is a speculation based technique bound to increase activity in the system. This was quantified with wasted speculative cycles and increase in cache accesses.

Several results related to performance improvements were presented. Performance benefits of PD with increasing number of processing cores were studied. To compare the benefits of PD"s forking model over the forking model of prior speculative parallelization systems, a new model called Inorder PD was implemented. This implementation had a restrictive program ordered forking of speculative threads. The comparison results indicated lower utilization of processing cores and lower performance improvements. Finally, performance improvements of PD with limited bandwidth between execution buffer pool and processing cores, and hardware resource restrictions were presented.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Industry designers have recently turned to multicore systems to uphold Moore's law. The out-of-order superscalar processors that have been so successful for the past several years have problems with scalability, complex design, significant power consumption, and diminishing returns. The multi-threaded programming model that is commonly used to execute an application on several processing cores also has its shortcomings. Programmers of multi-threaded programs must understand the problem, decompose it into several threads, program in the desired language with the parallel programming support provided, debug any correctness issues, and achieve scalable performance with an increasing number of processing cores. Clearly, this is a demanding set of tasks seldom applicable to large number of programmers. Newer generation of multicore systems that support novel software models seem to be a necessity in the near future. Solutions must be able to use the multicore systems without burdening the programmer, like the multi-threaded programming model now does.

Speculative parallelization has been studied for many years as one plausible solution for creating concurrency from a sequential program. It preserves the sequential execution of a program, but attempts to gain parallelism from the program by dividing it into threads, and executing them speculatively in parallel, with hardware support to ensure that these threads did not violate the sequential program order. Several proposals have studied different aspects of the execution model, in particular, focusing on the hardware support needed, and the composition of speculative threads.

This dissertation introduced a speculative parallelization model called Program Demultiplexing or PD. Speculative threads in PD are composed of methods in a program, which I believe is an apt choice for speculative execution for the following reasons. The large-scale nature of applications has made program code complex and has required adherence to software engineering principles in the development process for achieving modularity of code, flexibility of reusing it by many programmers, and maintainability of applications. Among many principles advocated to achieve these, dividing the program into many methods,

each targeting a specific subtask of the application is one of the fundamental ones. In addition, languages such as Java and C# are being widely adopted because of its object-oriented programming style and managed execution environment. These languages make modular programming a fundamental aspect of the program design, and provide a managed system to handle routinely performed tasks such as dynamic allocation and freeing of memory, ensuring safety of data references, and managing addresses of objects used in the program. Even though this dissertation does not evaluate a model for these languages, methods have always been, and will remain, a natural choice for programmers to encapsulate dependent computation.

The second aspect of PD is the novel execution model. Methods, even though called in sequential order, may be executed in parallel because of the parallelism that may exist between them. This implies that a call site for a method in the program is specified for convenience in conveying the execution to the hardware and may not represent its earliest possible execution. For a chosen call site of a method, PD speculatively executes the method at the trigger site, which often occurs earlier than the call site. At the call site, if the speculative execution is still valid, the thread is used or committed, and the program continues with its execution. The trigger site, therefore, specifies the point in the program when the speculative executions of a method for a particular call site can begin, without usually violating any of the dependencies that they may have with the program.

The trigger site is determined by observing several executions of a call site chosen for PD. A trigger point represents the point in the program when the dependencies for a given method's execution and its corresponding handler are satisfied. Trigger points collected for several executions are analyzed, and the suitable triggers are determined. Since the trigger site separates the method's execution from the sequential program, on firing of a trigger, the corresponding handler is speculatively executed to launch the method. The handler performs the tasks of providing parameters and predicting the reachability of the program to the call site, when the program is at the trigger site. This is achieved by speculative precomputation, i.e., speculatively executing a backward slice of instructions composed of instructions that compute the parameters and branches that these instructions and the call site may be control dependent on. Wasted execution in PD can, therefore, be due to evaluation of handler that aborts and does not invoke the method, and speculative threads that are executed but are later, squashed.

The hardware support for PD in this implementation is heavily based on prior speculative parallelization

proposals. Speculative execution of threads is performed on available processing cores in a multicore system. Private caches are used to hold the changes and accesses made by a speculative thread, which are tracked by a speculative tag unit. After the thread's execution, one means suggested in the dissertation for storing and validating the data of a speculative thread until its use, is the auxiliary set of storage structures referred to as the execution buffer pool. They consist of: a methods table which holds the list of outstanding speculative threads, a read set table that holds the read set consisting of locations referenced, a write set table that holds the write set consisting of locations modified and the data, and an invalidation cache, to efficiently determine if an address is the read or write sets of a speculative thread. The invalidation cache is accessed on every store location committed by the program to determine if there is any outstanding speculative thread that has referenced that location, and if so, indicates a dependence violation and the thread is squashed.

The implementation results were studied on a simulation based multicore system consisting of Intel x86 processors. The software support included generation of handlers and triggers from application binaries. Several integer programs from the SPEC CPU2000 suite were used for evaluation. The evaluation focused on frequently called methods in these benchmarks, since they usually represent a large fraction of program's execution time, and have read and write sets of size that can be held by hardware with modest storage requirements. Several results of the implementation were studied, which concluded with the performance improvements of PD and the benefits of PD's unordered forking model over a PD implementation with a restrictive program ordered forking model used in prior speculative parallelization proposals. Applying PD based execution model to larger scale applications is likely to create more opportunities for speculative execution with higher scalability and greater performance improvements.

The PD model is a generic framework that can extended in many different directions. I discuss some promising avenues for future work next.

Software implementation. The scope of handlers and triggers used in this implementation of PD can be expanded by altering the heuristics. Handlers and triggers may be optimized to minimize overheads and maximize potential for parallel execution. Alternate variants of handler and even alternate implementations of the execution model may be studied.

Hardware implementation. The implementation can be extended to support communication of data values between speculative threads, and allowing speculative threads to be forked not only by the program, but by other speculative threads. Similarly, tighter integration of processing cores, such as communication of register values via a low-latency inter-operand network, would allow handlers to obtain values from registers of other processing cores with low latency, instead of accessing the memory. Extensions to the hardware proposed in this implementation are required to support many of these ideas.

Runtime system implementation. I also see an implementation of PD in a managed system environment to be very promising due to a number of reasons. The software community, in large, has been migrating to managed execution environments and object-oriented languages due to its benefits with better software engineering, manageability of code, and productivity of programmers (for example, Microsoft Windows Vista running on the .NET managed environment). For PD, I believe that this means higher chances of identifying parallelism at the granularity of methods and, therefore, more opportunities. A managed environment implies dynamic optimizations and online profiling. With online profiles, the PD based program can be dynamically optimized with insertion and deletion of triggers depending on changing phases, application behavior, performance benefits, and wastage of execution resources. Runtime system also provides the appealing choice of performing software based speculative execution and evaluating triggers without any hardware support.

PD must be extended to support this kind of an environment. First, the system must be implemented on a different compilation model in which programs are compiled to an intermediate representation, and converted to native machine instructions only at execution time.

Second, handlers must be able to handle features of object oriented languages such as dynamic polymorphism, i.e., run time method binding or dynamic dispatch, a feature that is used when multiple classes contain different implementations of the same method. The target of a call site is resolved during program's run time, for example, by looking up a virtual table or vtable. To support this feature, the task of the handler, besides providing parameters, must also determine the target of the call site. The speculatively chosen target is speculatively executed and forms part of the speculative thread's data. Before a speculative thread can be committed or used, the target of the call site must also be compared with the parameters, to determine if the handler called the right target method and used the correct parameter values.

Finally, triggers that rely on a memory profile must be aware of garbage collectors, and could obtain information on when heap locations are live and dead, and consider them to identify opportunities in an application. In addition, garbage collectors can move a data value from one location to a different location during program's execution. Correctness is an issue if a speculative thread performs some computation on a location (i.e, the location belongs to the read or write set), which is later (non-speculatively) moved by the garbage collector to a different location. To deal with this problem, the garbage collector may perform a fake write to the source location of a data value which it is going to move. This generates an invalidate message to the execution buffer pool and squashes speculative threads that have accessed or modified the source location.

PD on future applications. In the future, we are likely to see many programming languages that provide easy means of expressing problems in a particular domain. These domain specific programming languages may also have support to automatically obtain concurrency from an application. One interesting possibility is unifying these parallelization models with PD to create higher level of concurrency. For example, a speculative thread in PD can be further optimized by other (speculative or non-speculative) parallelization models. Similarly, PD can be used to speculatively parallelize regions of code that have not already been parallelized by other means. Unifying many parallel models will be the key in using hundreds of processing cores that are likely to be available in the near future.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A low-level virtual instruction set architecture. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.

[2] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting postdominance for speculative parallelization. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[3] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 226–236, 1998.

[4] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the 14th International Symposium on Principles of Programming Languages*, pages 63–76, 1987.

[5] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 241–249, 1988.

[6] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[7] S. C. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[8] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proceedings of the 15th Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[9] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of the 17th Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 111–129, 2002.

[10] K. Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions Computer*, 39(3):300–318, 1990.

[11] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 149–159, 1996.

[12] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, 1992.

[13] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computer Survey*, 26(4):345–420, 1994.

[14] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[15] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 41–53, 1989.

[16] U. Banerjee. *Speedup of ordinary programs*. PhD thesis, University of Illinois, Urbana-Champaign, 1979.

[17] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[18] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[19] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 415–440, 2002.

[20] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in cw. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 287–311, 2005.

[21] A. J. C. Bik. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[22] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[23] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM*, 13(7):422–426, 1970.

[24] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[25] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic Ideas for an object parallel language. *Scientific Programming*, 2(3), 93.

[26] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[27] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques*, pages 777–786, 2004.

[28] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the Symposium on Compiler Construction*, pages 162–175, 1986.

[29] M. Burke, R. Cytron, J. Ferrante, W. Hsieh, V. Sarkar, and D. Shields. Automatic discovery of parallelism: a tool and an experiment (extended abstract). In *Proceedings of the International Conference on Parallel Programming*, pages 77–84, 1988.

[30] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the Conference on Java Grande*, pages 129–141, 1999.

[31] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32:444–458, 1989.

[32] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238, 2006.

[33] K. M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 124–144, 1993.

[34] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W.-m. W. Hwu. IMPACT: an architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275, 1991.

[35] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 186–195, 1999.

[36] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. Difficult-path branch prediction using subordinate microthreads. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 307–317, 2002.

[37] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[38] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: parallel programming on a network of multiprocessors. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 147–158, 1989.

[39] D. K. Chen, J. Torrellas, and P. C. Yew. An efficient algorithm for the run-time parallelization of DOACROSS loops. In *Proceedings of the Conference on Supercomputing*, pages 518–527, 1994.

[40] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 176, 1998.

[41] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, 2003.

[42] A. A. Chien and W. J. Dally. Concurrent aggregates. In *Proceedings of the Second Symposium on Principles & Practice of Parallel Programming*, pages 187–196, 1990.

[43] T. W. Christopher. Early experience with object-oriented message driven computing. *Proceedings of the 3rd Symposium Frontiers of Massively Parallel Computation*, 8(10):503–506, 1990.

[44] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 13–26, 2000.

[45] M. Cintra, J. Martinez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, 2000.

[46] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson. Design and implementation of the Power5 microprocessor. In *Proceedings of the 41st Annual Conference on Design Automation*, pages 670–672, 2004.

[47] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 306–317, 2001.

[48] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, 2001.

[49] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Tolerating dependences between large speculative threads via sub-threads. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 216–226, 2006.

[50] R. Colwell and R. Steck. A 0.6 um BiCMOS processor with dynamic execution. In *Proceedings of the 42nd International Conference on Solid-State Circuits*, pages 176–177, 1995.

[51] R. P. Colwell, W. E. Hall, C. S. Joshi, D. B. Papworth, P. K. Rodman, and J. E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Proceedings of the International Conference on Supercomputing*, pages 910–919, 1990.

[52] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions Computer*, 37(8):967–979, 1988.

[53] R. Cytron, M. Hind, and W. Hsieh. Automatic generation of DAG parallelism. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 54–68, 1989.

[54] W. J. Dally and A. A. Chien. Object-oriented concurrent programming in CST. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 434–439, 1988.

[55] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, 2006.

[56] A. L. Davis. The architecture and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, pages 210–215, 1978.

[57] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the International Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[58] J. B. Dennis. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd International Symposium on Computer Architecture*, pages 125–131, 1975.

[59] J. B. Dennis. First version of data flow procedure language. Technical Report MAC TM61, MIT Laboratory for Computer Science, 1991.

[60] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[61] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 71–81, 2004.

[62] P. K. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, 1995.

[63] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the alpha architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 281–292, 2002.

[64] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 59–68, 1998.

[65] J. A. Feldman, C.-C. Lim, and T. Rauber. The shared-memory language pSather on a distributed-memory multiprocessor. *SIGPLAN Notices*, 28(1):17–20, 1993.

[66] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.

[67] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[68] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin at Madison, 1993.

[69] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, 1992.

[70] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[71] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions Parallel Distributed Systems*, 3(2):166–178, 1992.

[72] M. Girkar and C. D. Polychronopoulos. Extracting task-level parallelism. *ACM Transactions Programming Language Systems*, 17(4):600–634, 1995.

[73] M. B. Girkar. *Functional parallelism: theoretical foundations and implementation*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[74] B. Goldberg and P. Hudak. Implementing functional programs on a hypercube multiprocessor. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues*, pages 489–504, 1988.

[75] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[76] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon dataflow processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 36–45, 1989.

[77] A. S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *Computer*, 26(5):39–51, 1993.

[78] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communication of the ACM*, 28(1):34–52, 1985.

[79] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in SUIF. *ACM Transactions Programming Languages and Systems*, 27(4):662–731, 2005.

[80] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of the Conference on Supercomputing*, pages 424–434, 1991.

[81] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, 2004.

[82] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, 1998.

[83] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102, 2004.

[84] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the Annual International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[85] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions Parallel Distributed Systems*, 1(1):35–47, 1990.

[86] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[87] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[88] C. Hewitt. Viewing control structures as patterns on passing messages. *Journal of Artificial Intelligence*, 8:323–364, 1977.

[89] G. Hinton, D. Sager, M. Upton, D. Boggs, and et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 1, 2001.

[90] C. Houck and G. Agha. Hal: a high-level actor language and its distributed implementation. In *Proceedings of the International Conference on Parallel Processing*, pages 158–165, 1992.

[91] R. A. Iannucci. Toward a dataflow/von neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, 1988.

[92] Intel. The IA-32 Intel architecture software developer's manual. *Intel Corporation*, 2, 2001.

[93] C.-H. Jo, K. M. George, and K. A. Teague. Parallelizing translator for an object-oriented parallel programming language. In *Proceedings of 10th Annual Conference on Computers and Communications*, 1991.

[94] M. S. Johnson. Some requirements for architectural support of software debugging. In *Proceedings of the 1st International Symposium on Architectural Support for Programming Languages and Operating System*, pages 140–148, 1982.

[95] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 59–70, 2004.

[96] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the Principles and Practice of Parallel Programming*, 2007.

[97] L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object-oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 91–96, 1993.

[98] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[99] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of the Conference on Supercomputing*, pages 407–416, 1990.

[100] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19:24–36, 1999.

[101] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–170, 2002.

[102] H.-S. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 71–81, 2002.

[103] M. Kishi, H. Yasuhara, and Y. Kawamura. Dddp-a distributed data driven processor. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 236–242, 1983.

[104] T. Knight. An architecture for mostly functional languages. In *Proceedings of the Annual Conference on LISP and Functional Programming*, pages 105–112, 1986.

[105] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[106] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2), 2005.

[107] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220, 2006.

[108] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, 1992.

[109] L. Lamport. The parallel execution of DO loops. *Communication of the ACM*, 17(2):83–93, 1974.

[110] J. R. Larus. C**: A large-grain, object-oriented, data-parallel programming language. In *Languages and Compilers for Parallel Computing*, pages 326–341, 1992.

[111] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[112] W.-C. Lau and V. Singh. An object-oriented class library for scalable parallel heuristic search. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 252–267, 1992.

[113] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of the 6th International Conference on Supercomputing*, pages 313–322, 1992.

[114] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *Proceedings of the Conference on Parallel Programming: Experience with Applications, Languages and Systems*, pages 85–99, 1988.

[115] D. J. Lilja. Exploiting the parallelism available in loops. *Computer*, 27(2):13–26, 1994.

[116] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[117] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 158–167, 2006.

[118] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, 2001.

[119] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 31st Annual International Conference on Programming Language Design and Implementation*, 2005.

[120] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 25(2):50–58, 2002.

[121] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In *Proceedings of the 8th International Parallel Processing Symposium*, 94.

[122] P. Marcuello and A. Gonzalez. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing*, pages 365–372, 1999.

[123] P. Marcuello and A. Gonzalez. A quantitative assessment of thread-level speculation techniques. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 595, 2000.

[124] P. Marcuello and A. Gonzalez. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 55. IEEE Computer Society, 2002.

[125] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing*, pages 77–84, 1998.

[126] I. Martel, D. Ortega, E. Ayguade, and M. Valero. Increasing effective IPC by exploiting distant parallelism. In *Proceedings of the 13th International Conference on Supercomputing*, pages 348–355, 1999.

[127] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th International Symposium on Principles of Programming Languages*, pages 2–15, 1993.

[128] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[129] J. R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, 1982.

[130] K. S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *Proceedings of the 8th International Conference on Supercomputing*, pages 54–63, 1994.

[131] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[132] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, 2003.

[133] M. Moir. Transparent support for wait-free transactions. In *11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

[134] K. E. Moore, J. Bobba, M. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 254–265, 2006.

[135] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *Proceedings of the 15th International Conference on Supercomputing*, pages 321–334, 2001.

[136] R. S. Nikhil. The parallel programming language Id and its compilation for parallel machines. In *Workshop of Massive Parallelism: Hardware, Programming, and Applications*, 1990.

[137] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[138] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*, pages 81–92, 2005.

[139] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, 1999.

[140] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*, pages 105–118, 2005.

[141] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communication of the ACM*, 29(12):1184–1201, 1986.

[142] P. Palatin, Y. Lhuillier, and O. Temam. CAPSULE: Hardware-assisted parallel execution of component-based programs. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 247–258, 2006.

[143] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, 1990.

[144] I. Park, B. Falsafi, and T. N. Vijaykumar. Implicitly-multithreaded processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 39–51, 2003.

[145] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Proceedings of the International Conference on Automated and Algorithmic Debugging*, pages 119–132, 1995.

[146] Y. N. Patt, W. M. Hwu, and M. Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, pages 103–108, 1985.

[147] D. A. Patterson and J. L. Hennessy. *Computer architecture: a quantitative approach, 4th Edition*. Morgan Kaufmann Publishers Inc., 2007.

[148] A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[149] C. D. Polychronopoulos. On program restructuring, scheduling, and communication for parallel processor systems. Technical report, University of Illinois, Urbana-Champaign, 1986.

[150] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the 9th International Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2003.

[151] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of 10th Annual International Symposium on Principles and Practice of Programming Languages*, pages 142–152, 2005.

[152] C. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzalez, and D. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 31st Annual International Conference on Programming Language Design and Implementation*, pages 269–275, 2005.

[153] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 294–305, 2001.

[154] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, 2002.

[155] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[156] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of the Conference on Supercomputing*, pages 637–646, 1989.

[157] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[158] B. R. Rau. Dynamically scheduled VLIW processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 80–92, 1993.

[159] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *Journal of Supercomputing*, 7(1-2):9–50, 1993.

[160] L. Rauchwerger, N. M. Amato, and D. A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th International Conference on Supercomputing*, pages 137–146, 1995.

[161] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 179–188, 2005.

[162] J. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[163] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, page 84, 1999.

[164] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, 1998.

[165] A. Roth, A. Moshovos, and G. S. Sohi. Improving virtual function call target prediction via dependence-based pre-computation. In *Proceedings of the 13th International Conference on Supercomputing*, pages 356–364, 1999.

[166] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 37, 2001.

[167] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, pages 185–196, 2006.

[168] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, 1989.

[169] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, 2003.

[170] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the Conference on LISP and Functional Programming*, pages 202–211, 1986.

[171] K. E. Schauser, D. E. Culler, and T. v. Eicken. Compiler-controlled multithreading for lenient parallel languages. In *Proceedings of the 5th Conference on Functional Programming Languages and Computer Architecture*, pages 50–72, 1991.

[172] M. S. Schlansker and B. R. Rau. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[173] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, 2000.

[174] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[175] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi. Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 226–234, Tokyo, Japan, 1986.

[176] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions Computer*, 37(5):562–573, 1988.

[177] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[178] G. S. Sohi and A. Roth. Speculative multithreaded processors. *Computer*, 34(4):66–73, 2001.

[179] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 27–34, 1987.

[180] SPARC. *The SPARC Architecture Manual*. Prentice Hall, 1992.

[181] G. Steffan. *Hardware support for thread-level speculation*. PhD thesis, Carnegie Mellon University, 2003.

[182] J. Steffan and T. 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 1–12, 1998.

[183] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, 2000.

[184] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 65–75, 2002.

[185] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, 2000.

[186] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[187] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 291–302, 2003.

[188] T. Takayanagi, J. L. Shin, B. Petrick, J. Su, and A. S. Leon. A dual-core 64b ultraSPARC microprocessor for dense server applications. In *Proceedings of the 41st Annual Conference on Design automation*, pages 673–677, 2004.

[189] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[190] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[191] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[192] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal*, pages 25–33, Jan. 1967.

[193] M. Tremblay. MAJC: An architecture for the new millennium. In *Hot Chips Symposium*, 1999.

[194] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliam, and S. Tse. The MAJC architecture: a synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[195] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the Symposium on Compiler construction*, pages 176–185, 1986.

[196] J.-Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.-C. Yew. The Superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[197] T. H. Tzen and L. M. Ni. Dependence uniformization: A loop parallelization technique. *IEEE Transactions Parallel Distributed Systems*, 4(5):547–558, 1993.

[198] L. Van Put, B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere. LANCET: A nifty code editing tool. In *Proceedings of the Sixth Workshop on Program Analysis for Software Tools and Engineering*, pages 75–81, 2005.

[199] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of the Principles and Practice of Parallel Programming*, 2007.

[200] F. Warg and P. Stenstrom. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *Proceedings of the 12th Annual International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, 2001.

[201] F. Warg and P. Stenstrom. Reducing misspeculation overhead for module-level speculative execution. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 289–298, 2005.

[202] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[203] M. Weiser. Programmers use slices when debugging. *Communication of ACM*, 25(7):446–452, 1982.

[204] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proceedings of the 20th Annual International Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 439–453, 2005.

[205] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 519–542, 2004.

[206] M. J. Wolfe. Optimizing supercompilers for supercomputers. Technical report, University of Illinois, Urbana-Champaign, 1982.

[207] A. Yonezawa, editor. *ABCL: an object-oriented concurrent system*. MIT Press, Cambridge, MA, USA, 1990.

[208] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.

[209] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[210] C. B. Zilles. *Master/slave speculative parallelization and approximate code*. PhD thesis, University of Wisconsin-Madison, 2002.

[211] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, 2000.

[212] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, 2001.

[213] C. B. Zilles and G. S. Sohi. Time-shifted modules: Exploiting code modularity for fine grain parallelism. Technical Report TR1430, University of Wisconsin-Madison, Computer Sciences Dept., 2001.

[214] C. B. Zilles and G. S. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, 2002.