PRE-EXECUTION VIA SPECULATIVE DATA-DRIVEN MULTITHREADING

BY

AMIR ROTH

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the University of Wisconsin—Madison

2001

Abstract

This dissertation introduces *pre-execution*, a novel technique for accelerating sequential programs. Pre-execution directly attacks the instructions that cause performance problems—mis-predicted branches and cache missing loads. In pre-execution, future branch outcomes and load addresses are computed on the side and the results are fed to the main program. In doing so, the main program is spared from having to incur the full computation latencies of these instructions. Pre-execution exploits *out-of-order fetch* and *decoupling*. Fetching and executing only critical load and branch computations while skipping over all unrelated instructions allows pre-execution to compute values faster than the main program. Decoupling, doing so in a separate thread, isolates stalls that occur in these computations so that they do not directly impact the main program thread.

This dissertation describes *speculative data-driven multithreading (DDMT)*, an implementation of pre-execution. DDMT implements the runtime component of pre-execution—responsible for pre-executing computations and communicating the results to the main program—as an extension to a superscalar processor. In addition to using the single cache hierarchy to allow pre-executing computations to prefetch for the main program, DDMT stores individual pre-executed instruction results in the shared physical register and then passes them one-by-one to the main program via a novel modification to register renaming called *register integration*.

For DDMT's setup component—responsible for finding load and branch computations and conveying them to the runtime component—this dissertation introduces an algorithm for automatically extracting performance-enhancing computations from program traces. The algorithm evaluates a benefit-cost function over all candidate computations in a trace and chooses those that maximize benefit (latency tolerance) while minimizing cost (execution overhead). The algorithm is formulated to permit software, hardware, and hybrid implementations.

The dissertation includes a simulation-driven performance evaluation of DDMT. Our results show that DDMT achieves 10% to 15% performance improvements for general-purpose integer programs running on an aggressive baseline processor with large caches, with the potential for greater improvements on likely future processor designs. We conclude that pre-execution and DDMT are promising technologies that merit consideration for inclusion in future machines.

Acknowledgements

Many people deserve thanks for helping me through the often frustrating, sometimes wonderful world of graduate school. First among these is my wife, Marci. Marci and I came to Wisconsin together in 1995. In the six years since, she has continuously supported me, repeatedly comforted me, occasionally dissuaded me from dropping out, and once even agreed to marry me. She is my rock. She is my best friend. She is my far better half and certainly far better than I deserve. I can imagine neither the last six years nor the next sixty without her.

My parents, Itzhak and Zipora, also deserve thanks, not only for the love and support they have given me—over the phone, through the mail, and occasionally in person—for the last six years, but for everything they have done for me during the previous twenty-three. I can only hope that I will be as good a parent to my future children as they have been to me. My sister Nurit and my brother Eatai have always been there when I needed them, and I will always be there for them.

I may not like to admit it, but I owe much of my academic and professional development to my advisor, Guri Sohi. Guri took me on in 1996 after, in his own words, "no one else would work with me." His handling of me in the five years since is best summed by the Rolling Stones: I may not have always gotten what I wanted, but I always got exactly what I needed. Guri has given me continuous support and provided me with countless opportunities. His technical insights have been inspiring and his confidence in my abilities and my work has been uplifting. He has taught me how to write, how to ask insightful questions and how to stand up for my ideas. Oh yes, it was his 552 class that turned me on to computer architecture in the first place. I respect any person who can put up with me for one week, much less five years, but my respect for him goes far beyond that. I hope he knows how much I appreciate everything he has done for me and everything he has made me do for myself.

I want to thank the other members of my thesis committee. Charlie Fischer, Jim Goodman, Mark Hill and Jim Smith not only agreed to read (or at least skim) this monstrosity, but also to sign my defense warrant. More importantly, they have provided me with valuable feedback and advice, whenever I asked for it, about any topic whatsoever. I also want to thank Charlie for assisting Milo Martin and myself with our first research venture and for letting me drive his BMW, Jim Smith for his friendly barbs and for embarrassing me cardiovascularly despite being 20 years my senior, and Mark for being the perfect counterpoint to Guri. In his two years in the department, Ras Bodik has also been a good friend while showing me by example how an assistant professor is supposed to act.

Most of my technical growth has come from interactions with other students rather than with faculty. Andreas Moshovos helped me organize my writing and my thinking and generally showed me the ropes during my early rounds of research. Stephanos Kaxiras, Subbarao Palacharla and Avinash Sodani were also helpful in those early years. More recently, I have needed less hand-holding and more sound-boarding. Fortunately, the University of Wisconsin has some of the best architecture sound-boards in the world. I want to thank Adam Butts, Brian Fields, Milo Martin, Ravi Rajwar, Dan Sorin and Craig Zilles for reading my paper drafts, attending my practice talks, helping me think through my ideas, listening to my complaints, sharing in my academic successes and failures, and allowing me to share in theirs. They have been as good an intellectual support system as one can ask for, and they have contributed much more than technical input to my graduate school experience. I want to thank Milo, in particular, for essentially handing me my first conference publication, yet still speaking to me after that experience.

I have made many friends in the past six years. Glenn Ammons, Paul Bradley, Bill Donaldson, Elton Glaser, Ben Teitelbaum, Todd Turnidge and Victor Zandy are a few, but there are more than I can name. Two in particular, Chris Lukas and David Melski, have been with me since my first days in graduate school. Our Friday lunches at Mickey's will undoubtedly be my most fond and lasting memories of Madison. I want to thank Chris and his wife, Erika, for providing Marci and me with countless nights of old-married-couple entertainment. Of the aforementioned, Craig and his wife Julie, Milo and his wife Denise, and Adam Butts and his fiancee Sue, have also been good friends to Marci and me. I will miss our cabin getaway weekends and our trips to the APT. Craig and Adam have also accompanied me on multiple snowboarding trips and have never laughed at me in my presence. Finally, I want to thank Dan Sorin, my fellow departmental sports fanatic, for allowing me to make merciless fun of the Cavs in exchange for his doing the same to the Phillies.

Finally, want to thank all those people with whom I have had less than regular interaction, but who supported me and contributed greatly to my academic experience nonetheless: the computer science faculty, the administrative staff which has dealt so pleasantly so many times with my disregard for deadlines and my propensity for losing materials, and the incredibly competent lab staff which has dealt with my ignorance of system administration even more frequently. I also want to thank the Condor team for their tool, without which the experimental portion of this work would have taken at least ten times as long.

My graduate work has been financially supported by a Wisconsin Alumni Research Foundation (WARF) fellowship, an IBM graduate fellowship and an Intel graduate fellowship, National Science Foundation grants MIP-9505853 and CCR-9900584, U.S. Army Intelligence Center and Fort Huachuca contract DABT63-95-C-0127, and by equipment donations from Intel and Sun Microsystems.

Table of Contents

List of Figures

Figure 2.10: Induction unrolling 53

List of T ables

Chapter 1

Introduction

This dissertation introduces *pre-execution*, a new paradigm for improving the performance of sequential programs. It also introduces *speculative data-driven multithreading (DDMT)*, an implementation of pre-execution.

Performance degradation in sequential programs is primarily due to *value latency*, the late availability of a value that directly or indirectly causes program execution to stall. In the natural course of execution, all needed values are ultimately computed. However, certain important values like branch outcomes and the addresses of soon-to-be referenced but uncached data blocks are often needed earlier than execution can supply them. Techniques for providing these values quickly by predicting them from previous values are extremely useful [7, 11, 13, 17, 16, 27, 28, 33, 47, 51, 57, 61, 62, 64, 77]. However, their accuracies and coverages are only about 95% and are unlikely to get much closer to 100% [34, 47]. The approximately 5% of dynamic branches and loads whose outcomes and addresses defy prediction are called *performance degrading instances (PDIs)*. PDIs account for up to 50% of execution times of many sequential programs [104, 106]. The few static instructions that are responsible for the majority of PDIs [1] are called *problem instructions (PIs)* [106].

Pre-execution is a way of using actual program execution to obtain PDI values in a timely manner. Pre-execution is motivated by the observation that only a fraction of the program is required to generate any particular value. If this fraction is small enough and if enough execution bandwidth is available, then the latency of computing this value can be effectively hidden from the complete program by executing—actually pre-executing—a parallel copy of this computation. Because it executes fewer instructions, the copy initiates the high latency component of the computation earlier than the complete program is able to. Because the parallel copy and the main program are decoupled from one another, long latencies in one do not induce stalls in the other. A pre-executing copy acts as a "performance troubleshooter," absorbing latencies so that they do not stall the complete program whose performance is externally visible.

In addition to introducing and defining pre-execution, this dissertation also describes an implementation of pre-execution called *speculative data-driven multithreading (DDMT)*. DDMT implements pre-execution as a set of small extensions to a dynamically scheduled superscalar processor.

This dissertation is organized into seven chapters. The goal of this chapter is to motivate, introduce, and define both pre-execution and DDMT. We begin with a brief argument for the need for the kind of parallelism pre-execution enhances: instruction-level parallelism (ILP). We continue with a description of the incumbent model for extracting ILP, the superscalar model, and its limitations. We then introduce pre-execution as a technique for overcoming the limitations of the superscalar model and its satellite technologies in fundamental ways. Finally, we provide a high level overview of DDMT, our proposed implementation of pre-execution, pointing out its novel aspects. The last section of this chapter summarizes the contributions of the dissertation and outlines the remaining chapters.

1.1 The Need for Implicit Instruction Level Parallelism (ILP)

The central role played by computers in every aspect of society has been driven by the rapidly increasing performance of general purpose microprocessors. Successive order-of-magnitude increases in processor performance—which recently have occurred every six years or so—have enabled leaps in software functionality and made the microprocessor into a ubiquitous commodity. The charter of computer engineers is to ensure that performance continues to grow at an adequate pace.

Microprocessor performance is a product of two factors: clock frequency and parallelism—the execution of multiple useful operations per clock cycle. Semiconductor technology—the continued miniaturization of CMOS devices—has driven both. Device scaling produces faster individual transistors and more of them, enabling higher clock frequencies and providing computer architects with more raw materials with which to extract parallelism. Continued growth necessitates an increased contribution from parallelism. Even at today's frequencies, processors spend a significant fraction of their time waiting for data from memory.

Increasing processor frequency will increase performance, but at a sublinear and shrinking rate of return as the memory waiting time is not reduced. Parallelism is needed both to make up for this decreased efficiency, by providing useful work that may be overlapped with memory latency, and to further boost the performance in its own right.

Parallelism can take many forms. An important form is the *instruction level parallelism (ILP)* available in varying degrees in every sequential—not explicitly parallel—piece of code. Enhanced ILP complements other granularities of parallelism and is the primary way to improve the parallelism of sequential code. It is the function of the microprocessor—potentially with help from software—to discover and exploit ILP, while giving the external appearance of sequential execution.

Parallelism and performance are not the only goals. Tightening constraints on power consumption, area, design and verification complexity, and reliability have shifted the research emphasis from ILP techniques that provide diminishing returns or use special purpose hardware, to innovative solutions that enhance ILP in fundamentally new ways while leveraging existing components. It is my belief that pre-execution is such a technique.

1.2 Obstacles to Instruction Level Parallelism

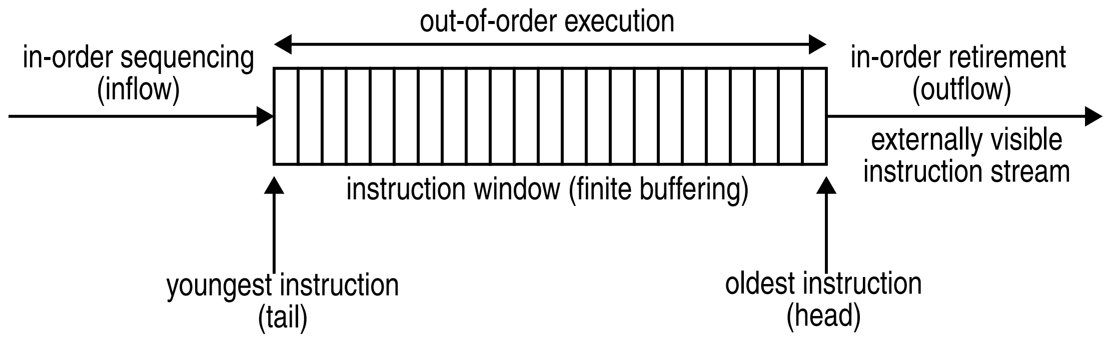
Limit studies of sequential programs have shown that, theoretically, most have sufficient inherent concurrency to sustain the parallel execution of over 20 instructions per cycle, even accounting for 100 cycle execution latencies that result from the use of realistic memory systems [90, 99]. However, for almost all of these programs, most modern microprocessors cannot sustain useful levels of parallelism that approach even one tenth of that amount.

This discrepancy is ultimately due to the von Neumann sequential programming model. The structure of computations in a program imposes a partial order on its instructions called the *data-flow order*. It is this partial order that defines maximum concurrency. The sequential programming model imposes a total order, called the *control-flow order*, on top of this partial order. The total-order of control-flow defines program state unambiguously at every instruction boundary, giving the programmer an interface that supports repeatable executions and allows him/her to reason about program behavior. Moreover, total-order (control-flow) is often the only way to specify the partial order, which for non-functional languages (most languages) is statically ambiguous. While the total order makes for a convenient programming interface, its implementation restricts concurrency.

1.2.1 The Incumbent: The Dynamically-Scheduled Superscalar Model

Dynamic scheduling or *out-of-order execution* is the standard technique for maximizing concurrency while implementing a sequential interface. An abstract dynamically scheduled superscalar processor is shown in Figure 1.1. The processor slides a fixed-size window over a program's dynamic instruction stream. Instructions are sequenced (flow into the window) and retired (flow out of the window) in program order. In-order retirement provides the external appearance

Figure 1.1 Abstract dynamically-scheduled superscalar processor.



of sequential execution. In-order sequencing establishes the correct data dependences between instructions required to implement the meaning of the program. Result buffering within the window allows instructions to execute out of program order. Precise knowledge of data-dependence information facilitates the exploitation of concurrency. Instructions that are known to be data-independent may execute in parallel. The earliest possible execution time of any instruction may be determined precisely by simply monitoring the availability status of its input dependence values. This same buffering even allows for *renaming*, a form of versioning that removes false dependences that are artifacts of the architectural name spaces, exposing even more parallelism.

1.2.2 Limitations of the Superscalar Model

Processor performance is measured by retirement (outflow) throughput. A processor can achieve peak performance if it can execute *useful instructions*—i.e., instructions that will ultimately be retired—at peak bandwidth every cycle. Performance degrades when not enough ready-to-execute useful instructions are available in a given cycle. The number of ready-to-execute useful instructions can drop to sub-peak levels due to interruptions in the inflow of useful instructions.

There are two kinds of inflow interruptions and both are due to the sequential programming model. *Direct interruptions* are caused by events that occur at the sequencing end of the processor. The primary causes of direct interruptions are *instruction cache misses* and *branch mis-predictions*. Instruction cache misses stall inflow completely, while branch mis-predictions stall the inflow of useful instructions. Branch mis-predictions are an artifact of the sequential pipeline which implies that instructions be sequenced *speculatively* (i.e., before all prior branches have executed and determined the actual dynamic path). The sequential data-dependence model means that any instruction sequenced after a mispredicted branch is useless.

Indirect interruptions are caused by events that occur at the retirement end of the processor. Indirect interruptions directly stall outflow. The finite size of the window propagates the outflow stall backwards via back-pressure to create an inflow stall. The primary cause of indirect interruptions are instructions with long execution latencies, most commonly *loads that miss in the data cache*. In this dissertation, we deal only with branch mis-predictions and data cache misses. We ignore instruction cache misses as they are both infrequent in the programs we study and not easily handled by our technique.

In-order sequencing poses another difficult problem. Ideally, execution should take place in global data-driven order. However, execution order is constrained by sequencing order, and sequencing order and global data-driven order rarely match. Since both sequencing and execution can only be performed with certain bandwidth, the result is a window-restricted execution schedule which under-utilizes execution bandwidth. In other words, in-order sequencing is not conducive to high processor utilization. It delays some instructions that—for data-independence reasons—could have been executed right away, while inserting others into the window that—for data-dependence reasons—will not be executed for a long time.

There are two apparent solutions. One is to create a machine with a very large window. Such a machine would smooth temporary inflow stalls and dampen outflow stalls by removing back-pressure. There are several problems with this approach. First, superscalar execution requires the implementation of several highly-parallel search algorithms whose purpose is to find independent instructions to execute every cycle. The complexity and performance of these algorithms degrades when they must examine many instructions in parallel. Second, and more fundamentally, there is no point in filling a large window with useless instructions. Larger windows necessitate speculating across more unexecuted branches in order for sequencing to proceed. The sequential nature of control flow speculation creates a geometrically decreasing probability that the instructions sequenced using accumulated guesses are useful.

With the physical size of a superscalar processor apparently limited, a second approach is to statically schedule instructions in a program to take full advantage of whatever buffering and latency tolerance the window does provide. This approach is useful and extremely important, but static instruction scheduling is limited by dependence analysis and no amount of scheduling can prevent stalls if latencies are sufficiently long.

To summarize, processor performance degrades whenever the number of unexecuted, ready-to-execute useful instructions drops below peak execution bandwidth. Instances of this scenario are due to a combination of five factors. We take the first three as given: (1) the von Neumann in-order retirement constraint, (2) engineering constraints and diminishing control-speculation utility that restrict the size of a practical window, and (3) fundamental limitations on compiler scheduling. We turn our attention to the last two: (4) inflow interruptions due either directly to control mis-speculations or indirectly to long latencies of realistic memory systems, and (5) low execution utilization due to the ordering mismatch between data-driven execution and control-driven (in-order) sequencing.

1.2.3 Value Analytical Predictors and Problem Instructions

Inflow interruptions are due to value latency. Execution is the primary, golden method for generating values. However, execution cannot generate certain values—branch outcomes and the addresses of needed data blocks that are not in the cache—quickly enough to prevent the need for those values from causing stalls.

Control-speculation and proactive management of the memory hierarchy (prefetching) are popular techniques for hiding the two most common sources of value latency—the pipeline and the memory system. The workhorses of control-speculation and prefetching are *branch* and *address predictors*, whose fundamental functions are to reduce the latency of branch outcomes and data addresses, providing them faster than conventional execution is able to. Branch and address prediction have been active subjects of research for a long time [7,11, 13, 16, 17, 27, 28, 33, 47, 51, 57, 61, 62, 64, 77]. Both have advanced to the point where values can be correctly predicted (and no performance penalty is incurred) for upwards of 95% of dynamic instances [33, 47]. However, the possibility for taking either technique to a 100% coverage and accuracy limit appears slim [34].

Before we proceed, we should mention that another possibility for tolerating load latency is to use *load value prediction* [39, 52, 53, 76] to generate load values that can be speculatively used in downstream computation. The predicted value—and the speculative computation—is then verified when the actual load completes. We do not consider load value prediction in this dissertation for two reasons. First, load value prediction accuracies are low—60% to 80%—relative to address prediction accuracies. Second, even if perfectly accurate, load value speculation may not improve performance significantly due to long verification latencies.

Providing accurate branch outcomes and prefetch addresses in a timely manner requires a mechanism that is faster than execution, but that can mimic execution results with high accuracy. The basis for such a mechanism lies in the regularity of addresses and branch outcomes produced by programs. Branch and address predictors study streams of addresses and branch outcomes and learn patterns that repeat in those streams. Patterns are associated with the program contexts in which they occur. When a context repeats, the associated pattern is used to generate a guessed value for a branch or address—usually with high accuracy. We call this approach *value analytical*. The value analytical approach works because programs are repetitive and because common programming idioms—i.e., program structures—produce common and repetitive branch outcome and address patterns. The problem is that some program idioms produce patterns which externally do not appear regular or repetitive (e.g., a branch that tests a data element obtained from external input). Other idioms produce repetitive patterns, but do so over such a large and incompressible context that simply storing this context becomes impractical (e.g., a traversal of a very large pointer-based data structure [11]). Still other idioms produce value streams with small and finite contexts that simply do not repeat, so that learning the requisite patterns on the first pass over the stream does not accelerate the handling of future passes over the stream, as no future passes will be made [11]. These are fundamental problems with value analytical approaches that do not appear surmountable from within.

The instructions that, for program-structural reasons, expose the limitations of value-analytical mechanisms have been called *problem instructions (PIs)* [104, 106]. Informally, a PI is any static instruction whose dynamic instances account for a disproportionate amount of a program's execution time, both on a per-static and a per-dynamic instruction basis. In other words, a problem instruction has many long latency instances. In practice, instructions that have a large fraction of long latency instances are pathologic in some way and, in particular, they are loads and branches whose behavior cannot be reproduced by value-analytical predictors. Problem instructions cause problems for value-analytical predictors and, as a result, for performance.

1.3 Pre-Execution

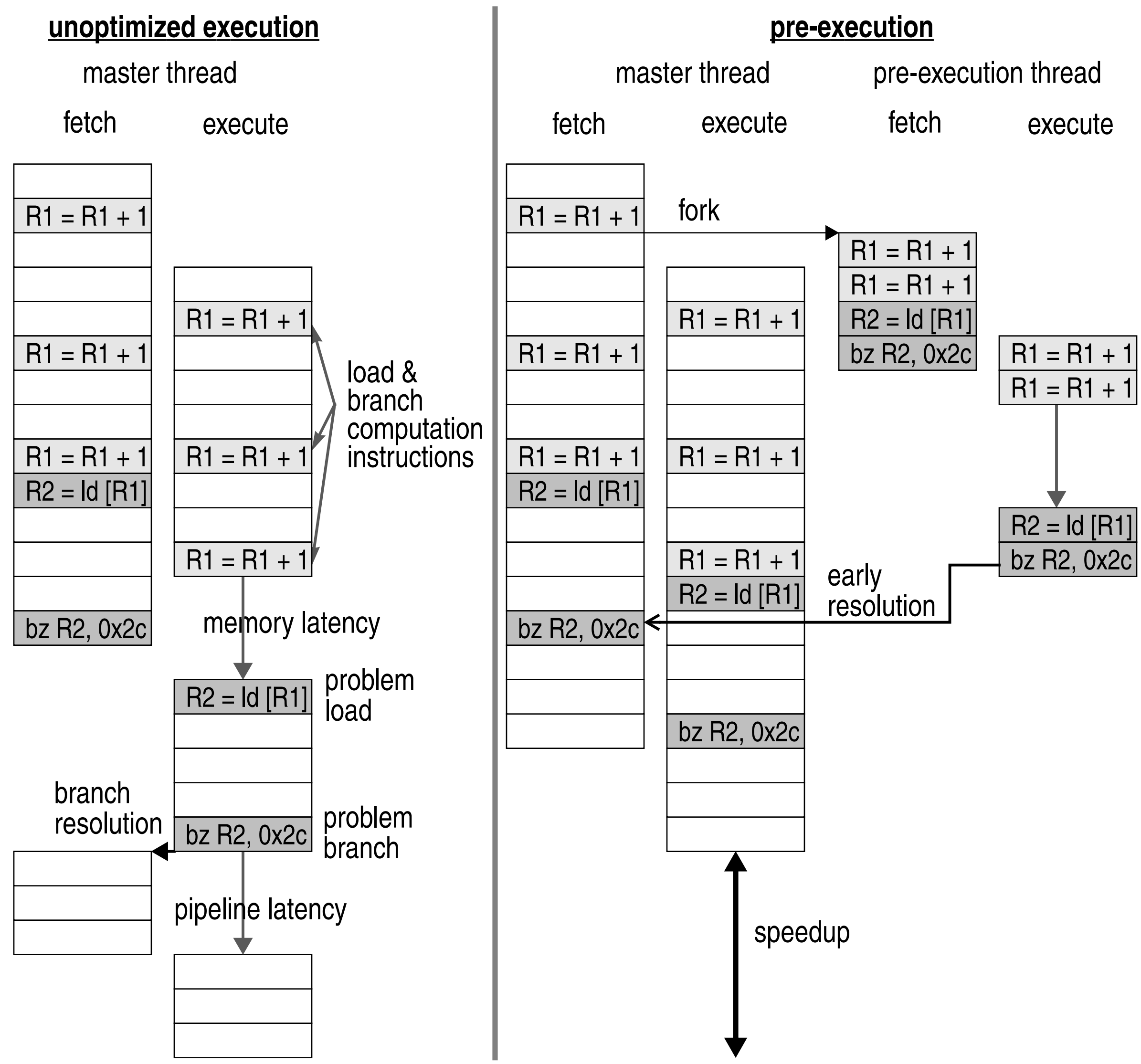
The limitations of conventional branch and address prediction lead to the following hypothesis: the values associated with problem instructions can only be obtained reliably via execution. However, this leaves us with a dilemma. On the one hand, we must obtain these values in a way that is faster than execution. On the other, the only reliable tool we have is execution. In essence, we must use execution in a way that is somehow faster and more streamlined than normal execution. The observation that motivates pre-execution is that, while the program is the source of all values, only a small subset of the program is needed to compute any particular value. Provided its computation is a small enough subset of the program and provided resources exist to execute this computation, any value can be made available in a timely fashion to the main program simply by expediting its computation. We call this kind of execution *pre-execution* because it executes pieces of the original program before a control-driven processor would be able to execute them via the conventional route.

This dissertation explores the use of pre-execution in the role of providing timely values for problem branches and loads. We propose to create copies of problem computations—computations of problem loads and branches—and to pre-execute these computations in parallel with, and decoupled from, the main program. The pre-executing copies compute problem instruction values faster than the complete program by virtue of their smaller sizes. Pre-executed computations act as "branch oracles" for the complete program, supplying it with pre-computed branch outcomes. In addition, the demand-driven cache management performed by pre-executing computations looks like proactive oracle cache management to the complete program. In essence, pre-executed computations "absorb" value latencies and their associated stalls on behalf of the complete program.

1.3.1 Example

Figure 1.2 shows an abstract example of pre-execution. An unoptimized execution is on the left, showing both fetch and execution schedules. Each box represents a dynamic instruction. Boxes in dark gray are instances of problem instructions (PIs). Boxes in lighter gray are instructions that participate in a PI computation. Empty white boxes are instructions that do not contribute to a PI computation. We will follow this convention in the rest of the dissertation. The abstract pipeline depth is three cycles from fetch to execute. Two sources of delay are shown. Memory latency delays the execution of a problem load that misses in the cache. Branch resolution latency exposes the depth of the pipeline on a mispredicted problem branch. Our convention is to use arrows to represent delays. A branch delay arrow appears after the corresponding branch signifying that delay "starts" once the branch completes. A load delay arrows is shown before the corresponding load, signifying that load latency "ends" when the load

Figure 1.2 Abstract example of pre-execution.



completes.

The right side of the figure shows how pre-execution hides these two latencies. The original (master) thread is again on the left. On the right is a pre-execution thread that fetches and executes the problem load and branch (in dark gray) and their computation (in lighter gray). The pre-execution thread is forked by the main thread after the first instruction in the problem computation is encountered. Because it executes fewer instruction, the pre-executing computation initiates the memory request for the problem load earlier than the master thread is able to on its own. By the time the master thread arrives at the problem load, the response from memory has already arrived and its latency has been effectively absorbed by the pre-execution thread. The pre-execution thread also hides the branch resolution latency from the master thread by sending the branch result to the fetch stage of the master thread. Pre-execution can hide pipeline latencies by sending pre-computed results to pipeline stages that occur before execution.

1.3.2 Aspects of Pre-Execution

Pre-execution is a simple idea. However, it has several aspects that may not be initially obvious. We make those explicit here.

The process of reducing master thread value latency is composed of two tasks. First, the desired value must be *computed* before the master thread needs it. Second, the value must be *communicated* to the master thread. The communication mechanism is specific to the implementation of pre-execution. It may be something as simple as the data cache which communicates pre-executed load values to the master thread implicitly via the prefetching effect, a branch predictor interface for communicating pre-executed branch outcomes, or something else entirely. The conceptual center of pre-execution lies in the much harder task of computing a given value faster than the master thread is able to.

Producing a value faster than the master thread requires that the value's entire computation is executed ahead of its master thread schedule. However, because the execution schedule is constrained by the sequencing schedule, simply "compressing" a computation's execution schedule will not significantly hoist (move up) its execution latency, unless its sequencing schedule is similarly compressed. We call the compressed sequencing of a computation—i.e., a sequencing that skips over unrelated master thread instructions—*data-driven sequencing*. Data-driven sequencing is a key component of pre-execution.

While data-driven sequencing allows long latencies to be initiated earlier during the execution of a program, decoupling is needed to allow these latencies to be tolerated. Latencies are "exposed" when the processor runs out of useful instructions to execute in parallel with them. A long latency instruction stalls the thread in which it executes, limiting the number of useful instructions whose execution can be overlapped with its own to the size of the window. By pre-executing instructions in a separate, decoupled context, we effectively move their latencies to a different thread. Now, when the latency is incurred, only the pre-execution thread stalls. The main thread can keep fetching, executing, and retiring arbitrary numbers of instructions. Performance does not visibly degrade until the main thread itself experiences a stall.

Data-driven sequencing powers pre-execution, but it also casts it in a *complementary* role. In-order sequencing of the complete program unambiguously establishes the data dependences that produce the program's intended outcome. There is only one currently known way of determining whether a data-driven sequencing of a computation produces data dependences that precisely correspond to those established by in-order sequencing of the complete program: executing the two sequences and comparing the results. This fact has two implications. First, any data-driven sequencing activity—and consequently pre-execution—must be performed *redundantly* with the in-order sequencing of the complete master thread. Second, a pre-executed result can only be used *speculatively* by the master thread—i.e., it must verified via re-execution. Now, redundant execution can be avoided if a pre-executed result can be proven to be correct via some sequencing-based invariant. In fact, we later introduce a mechanism for doing just that. However, redundant sequencing is unavoidable. Since a processor does not contain nearly enough bandwidth to redundantly sequence the computation of every branch and every load (even if this bandwidth did exist, it would very likely be better used to sequence other programs), pre-execution must be staged as a rarely-used complement to a second, lower cost technique that can perform the bulk of the heavy lifting with satisfactory results. Fortunately, value-analytical prediction is such a technique.

Despite its relatively high cost, complementary-technique status and "merely speculative" nature, pre-execution is a powerful general mechanism. Pre-execution fundamentally overcomes the basic limitations of superscalar processors to provide timely values for problem instructions when these cannot be supplied early enough and accurately enough by any other means. When used judiciously, pre-execution can provide a significant performance benefit.

1.3.3 A Definition of Pre-Execution

To help understand the important aspects of pre-execution as well as its context in the realm of high-performance sequential processing, we provide a formal definition. Pre-execution is characterized by three properties:

- *Proactive, out-of-order sequencing of problem computations*. If master thread stalls are to be *avoided*, then problem instruction (PI) value latency must be overlapped with instructions that are older than the PI instance itself. Doing so requires both foreknowledge of the downstream existence of the PI instance (proactive) and the ability to get to it faster than the master thread (out-of-order sequencing). Proactive, out-of-order sequencing removes the in-order sequencing constraint from PI computations, allowing them to be hoisted arbitrary distances with respect to the master thread.
- *Decoupling of problem computations from the master thread.* Decoupling enables arbitrary degrees of overlapping and latency hiding. Decoupling a problem computation from the master thread—e.g.., executing it in a different thread—effectively "moves" the latencies and stalls associated with that computation to that thread. Since stalls incurred within one thread do not effect the other thread, long latencies moved to a pre-execution thread may be overlapped with an arbitrary number of useful master thread instructions. In parallel with a stall in a pre-execution thread, the master thread can keep retiring instructions out of the window, sequencing new instructions into the window, and generally doing useful work for an arbitrary number of cycles.

- *Applicability to ordinary, unmodified, sequential programs*. There are systems that naturally support proactive out-of-order sequencing and decoupling. However, these do not also support arbitrary sequential programs. To be applied to sequential programs, pre-execution is cast in a way that relieves it from architectural correctness obligations, leaving it only with statistical correctness obligations—i.e., pre-execution must do the right thing often enough so that overall performance improves.

There have been several proposed pre-execution implementations [14, 20, 36, 68, 69, 72, 71, 73, 85, 106]. Speculative data-driven multithreading (DDMT) [73] is the subject of this dissertation, all the rest will be discussed in Chapter 6. All of these implementations share the above three properties which do *not* specify (a) whether pre-executed computations are copied from the program or created from scratch to mimic computations within the program, (b) whether the computation selection process is manual or automatic, or (c) how pre-executed results are communicated to the master thread. It is the implementation details of these aspects that differentiate pre-execution systems from one another.

These properties also serve to distinguish pre-execution implementations from systems that use performance enhancing execution modes other than pre-execution. Systems that redundantly sequence pieces of the program in a control-driven manner (i.e., sequence more instructions than are necessary to execute specific computations) [4, 8, 54] are not pre-execution systems. Neither are systems that divide a sequential program along dynamic control boundaries [2, 29, 38, 83]. Systems that redundantly sequence computations but do not decouple them from the main program—i.e., duplicate computations in line with the main program—are not considered to perform pre-execution. Finally, systems that require changes to the basic representation and externally visible execution model of the program [5, 24, 42, 46, 63, 74, 75, 78, 89, 91, 98] are also excluded by this definition. Chapter 6 will discuss several of these systems as well.

1.3.4 Implementation: Speculative Data-Driven Multithreading (DDMT)

A pre-execution implementation has two groups of components. The first group contains *pre-execution setup tasks*—tasks that are not fundamentally coupled to the runtime mechanics of pre-execution. These tasks are certainly crucial, but their completion sets up pre-execution rather than actively participating in every pre-execution instance. These tasks may be performed once, potentially statically, and their cost may be amortized over many dynamic executions. These tasks include identification of static problem instructions, selection of computations for problem instructions, and communication of problem computations to the processor.

The second group of components implement *pre-execution runtime tasks*—tasks that are performed every time a pre-execution instance takes place. Runtime tasks are *initiation and initialization* of problem computations, *pre-execution* of problem computations, and *communication* of pre-executed results to the master thread. The cost of these tasks is dynamic and any inefficiency in their implementation directly detracts from the performance impact of pre-execution.

Speculative data-driven multithreading (DDMT) is an implementation of pre-execution as it is defined above. DDMT's runtime component is built as a microarchitectural extension to a dynamically-scheduled superscalar processor that uses a physical-register-style out-of-order execution core. The proposed implementation can also leverage some of the microarchitectural components that would be present in a processor that supports simultaneous multithreading (SMT) [22, 26, 94, 95, 102]. The choice of processor substrate is motivated by the advantages that a centralized pipeline organization and a shared memory system provide for pre-execution.

Like most other pre-execution proposals, DDMT uses the shared memory hierarchy to allows a pre-executing computation to perform de facto cache-prefetching for a master thread. However, unlike all other proposals, DDMT implements a second, very aggressive channel for communicating pre-executed results to the master thread. Specifically, it stores individual pre-executed instruction results in the shared physical register file and then passes them one-by-one to the master thread via a modification to register renaming called *register integration*.

To complement the proposed implementation of the runtime component, this dissertation presents a framework for automated selection of pre-execution computations which may be used as the basis for a future implementation. The actual implementation of the setup component of DDMT is left open, although several implementations are considered and discussed at the end of Chapter 3.

This dissertation includes a simulation-driven performance evaluation of DDMT. The baseline for this evaluation is an aggressive 8-wide superscalar processor with large caches and a large branch predictor. Despite high baseline performance, we find that DDMT achieves speedups of 10% to 15% on general purpose integer programs. Performance improves as DDMT successfully reduces both the average load latency and the average branch mis-prediction resolution latency in these programs.

1.4 Dissertation Contributions and Outline

This dissertation's thesis comprises the following two assertions:

- ***Pre-execution is a good method of extracting additional ILP from sequential programs.***
- ***Speculative data-driven multithreading (DDMT) is a good instantiantion of pre-execution, supporting an evolutionary implementation of its runtime component and automation of its setup component.***

In this dissertation, I make the following contributions:

- I introduce and define pre-execution.
- I characterize data-driven threads (DDTs) and describe an algorithm for automatically extracting and selecting useful DDTs from program traces.
- I introduce and describe speculative data-driven multithreading (DDMT), an implementation of pre-execution, as a set of extensions to a dynamically scheduled superscalar processor.
- I present an empirical evaluation of DDMT in support of my thesis.

The remainder of the dissertation is organized into six chapters. Chapter 2 presents the DDMT pre-execution model—i.e., the way in which DDMT implements pre-execution. The use of register integration has many implications which make DDMT's interpretation of pre-execution rather distinctive. Chapter 3 describes the metrics and algorithms used in the process of automatic extraction and selection of candidate pre-execution computations. Chapter 4 presents the DDMT microarchitecture, focusing on the extensions and modifications required to implement DDMT on top of a conventional superscalar design. A large fraction of the chapter deals with the implementation of register integration, a new technique with several interesting aspects. Chapter 5 presents an empirical, simulation-driven evaluation of DDMT. The chapter contains both limit studies and characterizations that are of general interest to pre-execution, as well as a focused design-space exploration of the effects and interactions of the various DDMT components, including the DDT selection algorithm. Chapter 6 provides context for pre-execution and DDMT by discussing prior and contemporaneous related work by both myself and others. Chapter 7 summarizes the dissertation and concludes.

Chapter 2

Pre-Execution via Speculative Data Driven Multithreading

Speculative data-driven multithreading (DDMT) is our proposed implementations of the runtime component of pre-execution. DDMT consists of a small set of modifications to a dynamically scheduled superscalar processor.

What separates DDMT from other proposed implementations is its mechanism for communicating pre-executed results to the master thread. A description of this new mechanism, which we call *register integration,* occupies a large fraction of this dissertation. Register integration implements direct, instantaneous communication via a modification to register renaming that effectively allows a master thread and pre-executing threads to share physical registers. In DDMT, pre-executed operations are often *not re-executed by the master thread*. Register integration enables DDMT to implement pre-execution in the truest sense. Pre-executed instructions are actually instructions from the original complete program whose execution has been time-shifted ahead of (i.e., pre) its “natural” place in the program.

DDMT's unit of pre-execution is the *data-driven thread (DDT)*. A DDT is a sequence of instructions that encodes the computations of one or more problem load or branch instances. Each static DDT is associated with a static *trigger instruction*. Whenever the master thread encounters an instance of a trigger instruction, the processor forks a copy of the corresponding DDT. At this point, the two threads—the master thread and the DDT—begin executing in parallel. A DDT implements pre-execution—rather than post-execution or redundant parallel execution—because it is even with the master thread at the point of the fork and from that point forward, sequences and executes a small subset of the instructions processed by the master thread.

To sequence a DDT, the processor steals some sequencing bandwidth away from the master thread. Once in the execution core, DDT instructions are dynamically and transparently interleaved with instructions from the master thread. When a DDT completes execution, its hardware context is reclaimed but the results it computed are still "alive" in physical registers. The master thread uses register integration to locate these pre-computed values, associate them with the proper corresponding instructions in its own sequential stream and claim them as its own.

The use of register integration conveys many benefits—both performance and otherwise—but also imposes many constraints on DDMT. The most stringent constraint prevents DDTs from interpreting control-transfers of any kind—both implicit and explicit—and hence from using control-flow for sequencing. The inability to use control-flow restricts the kinds of DDTs that can be pre-executed.

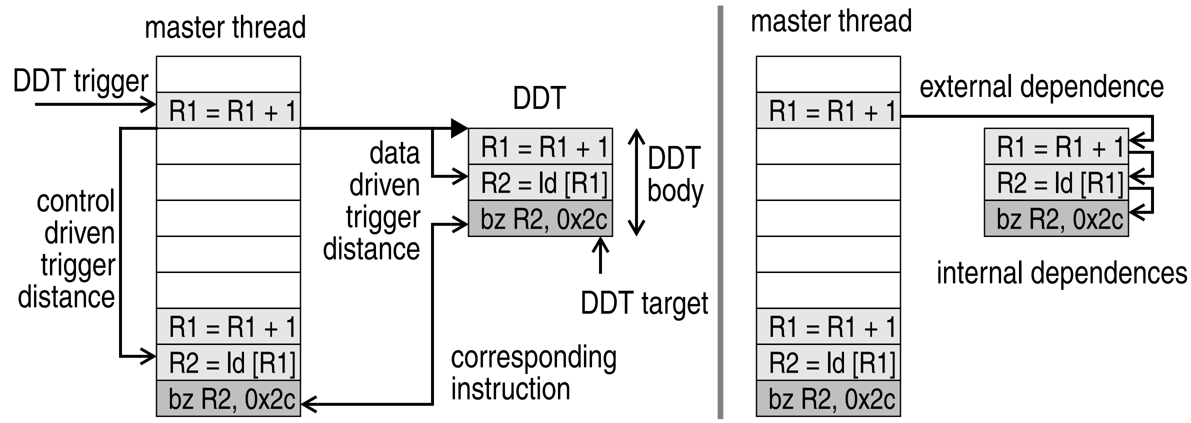
However, this restriction is not fatal as DDMT can use various techniques to "simulate" control-flow. A second constraint imposed by register integration is that DDTs must match dynamic program dataflow graphs precisely, instruction for instruction, and may not be optimized beyond those graphs. The inability to optimize DDTs hurts performance. Ironically enough, however, it also restricts DDT structure sufficiently to allow the DDT selection process to be automated.

Automatic DDT selection and the details of the DDMT microarchitecture are the subjects of chapters 3 and 4, respectively. In this chapter we describe DDMT's implementation of pre-execution. We briefly introduce the microarchitecture and walk through a "day in the life of a DDT." We also introduce some key concepts of register integration. These are needed to motivate the control-flow and optimization restrictions. The bulk of the chapter describes the DDMT pre-execution model the collection of unique constructs employed by DDMT to circumvent, but sometimes to actively exploit, the constraints imposed on it by register integration.

2.1 DDMT Primer

A description of the DDT execution model requires a basic understanding of the mechanics of DDMT. This section presents a brief introduction to the DDMT microarchitecture, describes its basic operation, and introduces the function of register integration.

Figure 2.1   DDMT terminology.



2.1.1 T erminology

To help with the rest of the chapter , we begin by establishing some terminology. We ma y ha ve used some of this terminology loosely already . At this point we would like to formalize it using F igure 2.1 as an aid.

The sequential, control-driven program thread is referred to as the *master thread* . A pre-executing computation is called a *data-driven thread* or a *DDT*. We use the term DDT in both a static and a dynamic sense . A DDT is both the static description of a problem computation and a pre-executing dynamic instance of that computation. We most commonly use the term in its dynamic sense . When speaking of the static representation of a DDT , we will use the term *static DDT* .

A DDT has several components . The *body* of a DDT is the sequence of instructions that DDT pre-executes . The *DDT target(s)* are the problem instruction instances the DDT is ultimately attac king . An important component of a DDT and a term we will use frequently is the *DDT trigger*. The DDT trigger is an instruction in the master thread that "triggers" the forking of the DDT. The DDT trigger is not a part of the DDT body as it is not pre-executed. Like a DDT, the DDT trigger is also both a static and a dynamic term. A static DDT trigger is associated with a static DDT. A dynamic instance of a static DDT trigger triggers the forking of a dynamic instance of the corresponding static DDT.

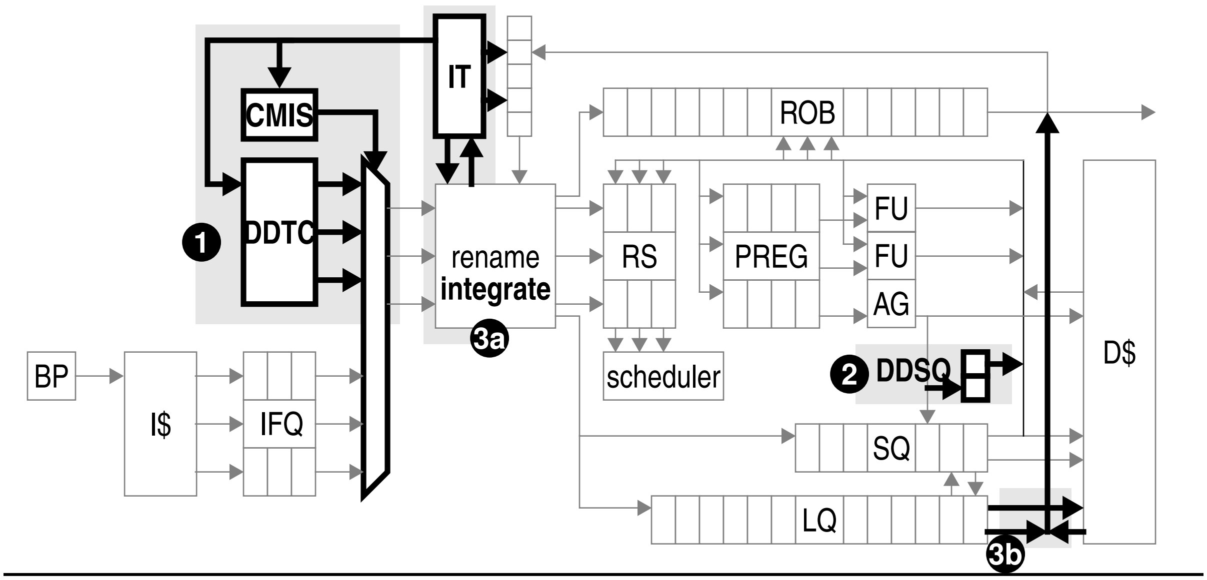
Each dynamic DDT instruction has (hopefully) a *corresponding instruction* within the master thread. Collectively, a dynamic DDT has (again, hopefully) a *corresponding computation* within the master thread.

When speaking of a DDT instruction—especially in the context of calculating how much faster a DDT can sequence to a given instruction than the master thread can sequence to its corresponding instruction—we will refer to the instruction's *trigger distance*. An instruction's trigger distance is the number of dynamic instructions between itself and its DDT's trigger instruction. A DDT instruction actually has two trigger distances. The *data-driven trigger distance* is its trigger distance from within the DDT. Its *control-driven trigger distance* is the trigger distance of its corresponding instruction within the master thread. The data-driven trigger distances of the instructions in a DDT are consecutive and start from zero. The control-driven trigger distances of the same instructions are not consecutive reflecting the fact that the corresponding master thread instructions are separated by other unrelated instructions. An instruction's data-driven trigger distance is always smaller than or equal to its control-driven trigger distance.

When discussing DDT pre-execution, we will speak of values and data-dependences. In doing so, we distinguish between two kinds. A value or a data-dependence is *DDT-internal* if both producer and consumer of the value are DDT instructions. A value or a data-dependence is *DDT-external* if the consumer is a DDT instruction and the producer is a master thread instruction outside the DDT. The producer of a DDT-external value may be the trigger instruction or any instruction older than the trigger. However, it cannot be any instruction younger than the trigger. Note, there is *no* explicit flow of values from a DDT to the master thread. In other words, master thread instructions do not read values written by DDT instructions. The only way for DDT values to flow back to the master thread is via register integration and once integrated a DDT value effectively “becomes” a master thread value.

Finally, we also need some terminology in order to effectively discuss problem instructions. A *problem instruction (PI)* is a static instruction that meets some performance criteria. The precise criteria are unimportant right now, only the term is important. The dynamic instances of a problem instruction are collectively called *problem instruction instances (PIIs).* We distinguish between two kinds of PIIs. The PIIs that dynamically exhibit problems—i.e., the particular dynamic instances of a load which miss in the cache and the particular dynamic instances of a branch which are mispredicted—are called *performance-degrading instances (PDIs).* The PIIs that do not exhibit problems are called *non-performance-degrading instances (NPDIs).*

Figure 2.2 DDMT microarchitecture.



2.1.2 DDMT Microarchitecture Overview

DDMT is implemented as a set of localized extensions and modifications to a dynamically scheduled superscalar processor with a physical register file style execution core microarchitecture. Figure 2.3 shows a block diagram of the DDMT microarchitecture. The DDMT-specific components are emphasized.

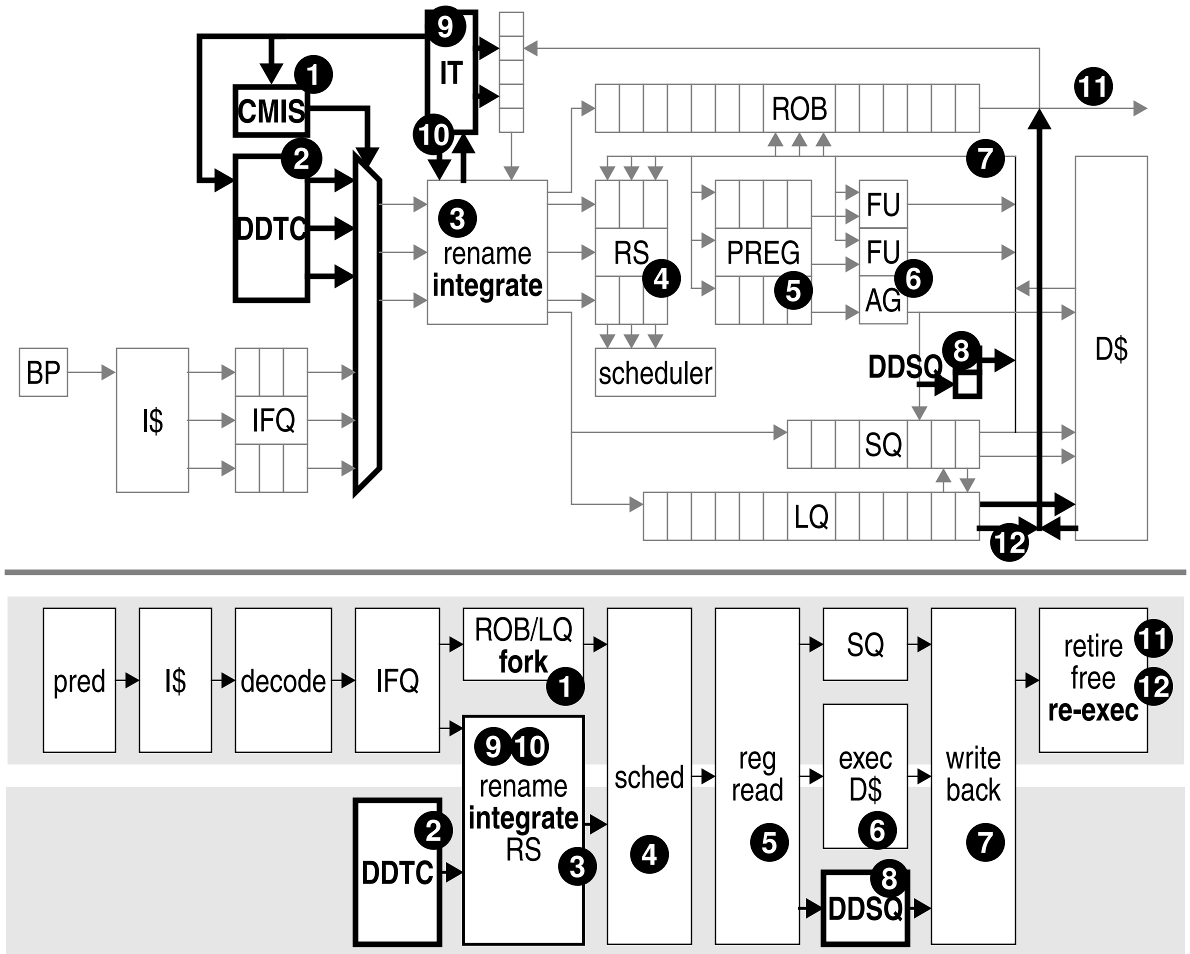
DDMT extensions come in three groups. The first group (marker #1) is the DDT-specific front end. The major components in this group are the *data-driven thread cache (DDTC)* and the combined *context manager/injection scheduler (CMIS)*. The DDTC stores static DDTs. The CMIS is responsible for forking DDTs, allocating and deallocating hardware register contexts, and scheduling active DDTs for "injection" into the execution core. The second group (marker #2) comprises a single extension to the out-of-order execution core, the *data-driven* *store queue (DDSQ)*. The DDSQ is the DDT analog of the conventional store queue (SQ). It is required because DDT stores must be handled differently than conventional stores. The final group of extensions is the implementation of register integration. The major components in this group are the *integration table (IT)* and integration logic which act during the register renaming stage (marker #3a) and a load re-execution mechanism (marker #3b) which acts at the retirement stage.

2.1.3 The Life of a DDT

With a basic understanding of the DDMT microarchitecture, we turn to its mechanics. Figure 2.3 reprises the microarchitecture and augments it with a dual pipeline diagram. The top pipeline shows the processing stages of control-driven—i.e., master thread—instructions. The bottom pipeline shows the processing stages of data-driven DDT instructions. Stages that straddle both pipelines are common to both types of instructions. The matching numbered markers on the figure represent the important events in the life of a DDT.

A DDT is born when the master thread renames its trigger instruction. The CMIS observes the master thread's renaming stream and forks the corresponding DDT if any instruction matches a trigger (marker #1). Forking a DDT consists of allocating a free hardware register context (map table) for it, initializing this context with a copy of the master thread's context and scheduling the DDT for injection into the renaming stage. Since DDTs are purely speculative, the CMIS may ignore a trigger sighting if, for instance, no free register context is available.

Figure 2.3 Events in the life of a DDT.



In an injection cycle, the next uninjected DDT segment is extracted from the DDTC (marker #2) and the corresponding instructions are injected into the register renaming stage (marker #3). If the renaming of a DDT is complete, the CMIS reclaims its hardware register context. If a DDT cannot be fully injected in one cycle—and most cannot be—the CMIS schedules its next uninjected portion for injection in some later cycle. There are several possible policies for scheduling the injection of a DDT vis-a-vis the master thread and other active DDTs. DDT instructions are renamed using the register map table that belongs to their allocated context. A DDT instruction's result is allocated a free physical register and a corresponding entry entered into the IT.

All instructions injected into the DDMT execution core are marked with three additional bits. The *data-driven bit* distinguishes DDT instructions from conventional instructions. The *internal communication bit* is set only for DDT stores and some DDT loads. The *integrating bit* distinguishes instructions which have integrated a physical register allocated by another instruction from those that have not. The data-driven and internal communication bits are statically generated and tag each instruction in the DDTC. The integrating bit is dynamically generated by the integration circuit. On entry to the execution core, DDT instructions (recognized by their set data-driven bits) are allocated reservation station (RS) entries but are not allocated entries in the ROB, LQ or SQ. The ROB, LQ and SQ are structures whose function is the implementation of externally visible sequential semantics. Since DDT instructions are not externally visible and do not obey sequential semantics per-se, they have no use for entries in these structures.

DDT instructions execute exactly like control-driven instructions. The map copy operation and subsequent renaming make sure that data-driven instructions are woken when corresponding inputs from either the master thread or from older instructions within the same DDT become available. DDT operations read ordinary physical registers (marker #5) issue to ordinary functional units (marker #6) and write their results into the ordinary physical register file upon completion (marker #7). The lone execution difference between DDT and control-driven instructions manifests for all stores and for loads marked with the internal-communication bit. These are routed to the DDSQ (marker #8). DDT loads whose internal communication bit is clear are treated as conventional loads.

On completion and writeback, the first stage in the life of a DDT instruction ends rather uneventfully. DDT instructions do not retire or free any physical registers. They have no ROB, LQ or SQ entries that need to be freed. DDT store values are not written into the data cache. When a DDT instruction completes, the only record that remains of its existence is the cache block it potentially prefetched, the physical register into which it computed its result and the IT entry that allows the master thread to locate this physical register (marker #9).

The second phase in a DDT's life begins when the physical registers it allocated are integrated by instructions from the master thread (marker #10). A data-driven result may be integrated in any state: after completion, mid-flight (i.e., during execution), or while sitting in a reservation station waiting for input operands to become available. If the integrating instruction is a mis-predicted branch or jump and the integrated result is complete, the DDMT processor may initiate recovery immediately, during the rename stage. This effect is called *instantaneous branch resolution* and is the effect we strive for when pre-executing problem branches. Since it operates during register renaming, register integration cannot help the master thread avoid a branch mis-prediction entirely. To avoid branch mis-predictions completely, a mechanism that sends pre-executed branch outcomes to the fetch stage is required. Several such mechanisms have been explored [14, 36, 69, 106].

Regardless of result status, the integrating instruction performs the complementary steps to those performed by the corresponding DDT instruction. Recall, DDT instructions are allocated RS entries but not ROB, LQ or SQ entries. Integrating instructions are allocated ROB, LQ and SQ entries but not RS entries. An integrating instruction does not need a new RS entry because the integrated result is either already sitting in some appropriately-tagged RS entry or has already been issued (and perhaps completed), in which case there is no need to schedule it for a second execution. The one exception to this rule is if the integrated instruction is a completed store. In this case, the store must be re-issued to the store queue to allow subsequent loads to retrieve values from it. Once integrated, a DDT result is identical to a control-driven physical register for all intents and purposes. An integrating instruction completes only the unfinished phases of its processing. If the integrated result is already computed, the integrating instruction may retire immediately (marker #11). Integrated loads are re-executed just before retirement to verify that their integration was correct (marker #12).

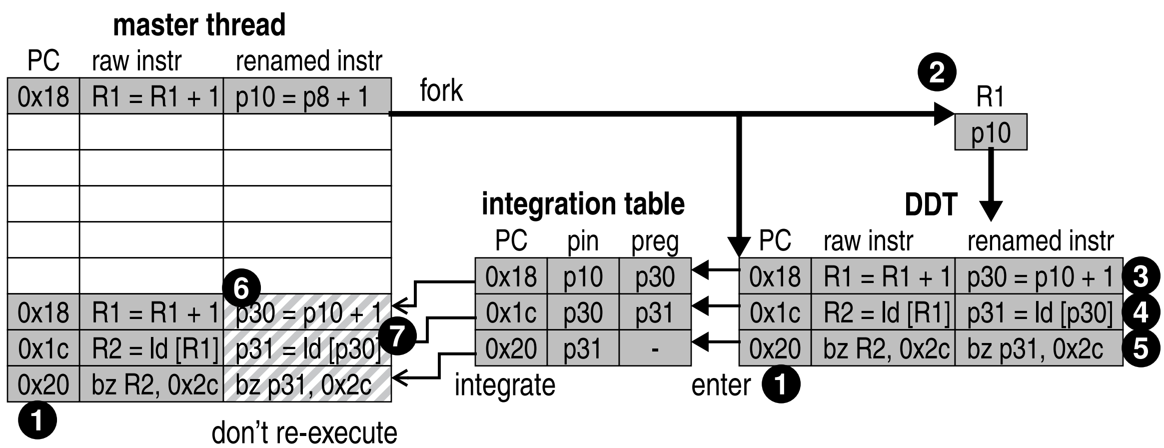
2.1.4 Pre-Execution Reuse via Register Integration

One of the most important, interesting and internally influential microarchitectural components of DDMT is *register integration*. Register integration exploits a unified physical register file to implement true result sharing between instructions from different speculation contexts of the same architectural thread. Result sharing is implemented as a modification to register renaming. An instruction shares a result already computed by another instruction by mapping its output to the appropriate already-written physical register rather than by computing a new value and writing it to a newly allocated physical register.

The interesting component of register integration is the facility that allows instructions to find physical registers that contain the values they themselves will compute and do so at register rename time. Register integration accomplishes this by maintaining a second index—called the *integration table (IT)*—over the physical register file. The IT indexes each physical register *(preg)* using the PC and input physical register numbers *(pin#)* of the instruction instance that initially created the value. The reason for this choice of indexing information is that it both completely specifies the computation *and* is available at register rename time. We use the PC rather than the opcode—which really specifies the computation—because we would like to distinguish between different static instructions that have the same opcode (we will explain why shortly). While renaming an instruction, the processor uses this index to locate the value that was generated using exactly the same computation the current instruction is about to perform—i.e., an older instance of the same instruction with the same physical register inputs. The argument for reuse is that the same computation will yield the same result. Notice, register integration only manipulates register mappings, *reuse is implemented without reading or writing the physical registers themselves*.

To use register integration for *pre-execution reuse*—master thread reuse of pre-executed results—we must deliberately manufacture a scenario in which DDT instructions have the same PCs and the same physical register inputs as

Figure 2.4 Pre-execution reuse using register integration.



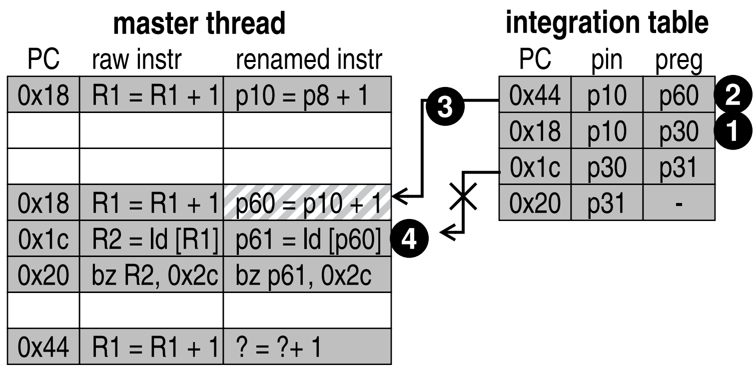
their corresponding master thread instructions. Figure 2.4 shows a register integration implementation of pre-execution reuse. The matching PC requirement is satisfied in a straightforward manner (markers #1). This is not to say that this executing DDTs with PCs that match those in the master thread is simple. In fact, we will see shortly that it is not. However, the basics of the solution are clear—DDT instructions have to be exact copies (down to the PC) of instructions from the master thread. A PC match is required for two reasons. One reason is that it is a good way of signalling that the instruction is an exact copy. We hold off on explaining the second reason until we have a little more background.

The matching physical register input requirement is satisfied recursively. To form the base of the recursion, we initialize a forked DDT with a copy of the master thread's register map (marker #2). This ensures that the initial DDT instructions—those that depend only on inputs from the master thread—will have the same input physical registers as their master thread counterparts. The inductive step is provided by integration itself. By integrating a particular instruction, we set its master physical register output to equal that of its DDT counterpart. In doing so, we are also setting the *inputs of its dependent instructions* to match those of *their* DDT counterparts.

We walk through Figure 2.4 as a working example. The DDT in the figure contains only one external input, R1 of instruction 0x18. The register context copy operation (marker #2) initializes the DDT's physical register mapping of R1 to p10, its post-fork value from the master thread. The DDT renames instructions within its own context in the usual manner, allocating physical registers for new results. The DDT executes a second copy of instruction 0x18 using p10 as its input and allocates p30 as its output (marker #3). It then executes the dependent instruction 0x1c with p30 as its input and a newly allocated p31 as its output (marker #4). Finally, it executes a dependent instruction 0x20 with p31 as its input (marker #5).

The master thread is able to integrate all three DDT instructions. We ignore the PC matches that facilitated integration and concentrate on the requisite physical register matches. The physical register match, p10, that enabled the initial integration of register p30 by instruction 0x18 was enabled by the context initialization procedure (marker #2). After integration, p30 is set as the master thread mapping of R1 (marker #6). This in turn makes it the master thread's input physical register for instruction 0x1c (marker #7) creating the input match that allows register p31 to be integrated. Repeating the process for p31 allows 0x20 to integrate the pre-computed branch outcome.

Figure 2.5 Problem with use of opcode matching in register integration.



We now have the necessary background to quickly explain the second reason why integration uses PC matches to distinguish between different but operationally identical static instructions—i.e., instructions with the same opcode and input logical registers. Figure 2.5 shows an integration scenario similar to the one in Figure 2.4, the difference is that the IT contains entries for two physical registers—p30 and p60—created by different but operationally identical instructions 0x18 (marker #1) and 0x44 (marker #2). For the purposes of integrating the current instruction 0x18 (marker #3), it does not matter which one we integrate. However, if we integrate the physical register created by the “wrong” static instruction as we are in this example, we forfeit the ability to integrate the instructions that depend on it (marker #4).

As shown in Figure 2.4, register integration relieves the master thread from having to re-execute instructions that have already been executed on its behalf in DDTs. Of course, since it takes place during register renaming, register integration cannot relieve the master thread from *re-sequencing* DDT instructions (we already knew that control-driven sequencing could not be avoided). When introducing pre-execution, we stated that pre-executed results must be treated as speculative, that in order to be used they must somehow be "checked" by the master thread. In addition to the mechanism for locating pre-executed results, register integration can be thought of as a mechanism that uses control-driven data-dependence information to "check" pre-executed results.

Register integration implements pre-execution by design. As it turns out, there is another reuse scenario that it implements naturally: *squash reuse*, the reuse of instructions that have been squashed due to a sequential control mis-speculation and subsequently re-executed. Squash and pre-execution reuse are each effective in its own right, and the two forms exhibit interesting synergy as well. A more detailed description of squash reuse and its interactions with pre-execution reuse is saved for Chapter 4.

2.2 Control Flow in DDMT

Data-driven sequencing—sequencing *only* those instructions that contribute to a problem computation and skipping over unrelated instructions—is one of the pillars of pre-execution and the mechanism that allows DDTs to get to and execute problem instructions faster than the complete program. Register integration's requirement that DDT instructions match their master thread counterparts PC for PC is significant. A DDT is data-driven not only in the sense that its instructions are all related via data-dependences. A DDT is also data-driven in the sense that it is *not control-driven*—i.e., its instructions' PCs are not sequential or related via explicit or sequentially implicit control transfers.

A discontinuous PC structure makes control transfers—both explicit transfers (branches and jumps) and the implicit transfers that take place from an instruction to the next sequential instruction—meaningless in DDTs and control useless as a sequencing mechanism. It is impossible to generate the instructions in a DDT and *only those instructions* using a combination of explicit and implicit control transfers from its first instruction.

The inability to use control flow leaves us with two problems. First, we must find some mechanism other than control flow for sequencing even the simplest DDTs. Second, we must find a way of achieving the effects of control flow within DDTs. There are two kinds of control structures whose effects we would like to simulate. *Conditional structures*—i.e., if statements—are used in DDTs to ensure that the pre-executed computation contains instructions from the path that matches the one the complete program will actually take. *Loop structures* enhance a DDT's ability to run ahead of the master thread, improving its latency tolerance capabilities. Many of the idiosyncrasies of DDMT and its unique pre-execution idioms are a product of dealing with this constraint.

2.2.1 Implicit Data-Driven Sequencing

In lieu of control, the model and mechanism used by DDMT to sequence DDTs is called *implicit data-driven sequencing*. In DDMT, the static representation of a DDT is a *list* of all of its instructions. To sequence a DDT, the CMIS injects this list into the pipeline in order and in its entirety. The execution core then executes all the instructions in the list, again in its entirety. A DDT may certainly contain conditional branches—a good thing since an important use of DDMT is the pre-execution and mis-prediction latency reduction of problem branches. However, the processor does not "interpret" branches within DDTs. The sole purpose of a branch in a DDT is so that its pre-executed outcome can be integrated by the master thread. A DDMT processor will continue executing all instructions within a DDT, regardless of the outcome of any branch.

In addition to providing the basis for a sequencing mechanism, implicit sequencing also supplies the vehicle for achieving the effects of control flow within DDTs. Specifically, we can exploit the fact that a DDMT processor will blindly execute any instruction sequence we give it, to "simulate" any control flow we wish, even control flow that is dynamically impossible. We can even "trick" the processor into executing control flow that is dynamically impossible, by including instructions in a single DDT that dynamically lie along disjoint paths. The next sections—Section 2.2.2 and Section 2.2.3—discuss some of the tricks used to simulate conditional control and loop control, respectively, within DDTs.

An alternative to the list representation would be a *static data-flow representation* in which each instruction explicitly names its data-flow successors. This is the kind of representation used in dataflow architectures [24, 42, 5, 63] and used in an earlier version of DDMT called speculative dataflow [72]. As far as *sequencing* goes, a static dataflow representation has some advantages including the ability to simulate loop control. In fact, in our concluding remarks in Chapter 7, we propose to adapt this particular feature into our list representation using a technique called *injector-based loop control*. However, in order to take advantage of register integration we need to rename a DDT in the conventional way and to do so requires that we impose a total order on its instructions. Once a total order is imposed on a static dataflow representation, it becomes equivalent to—but no simpler than—our list representation.

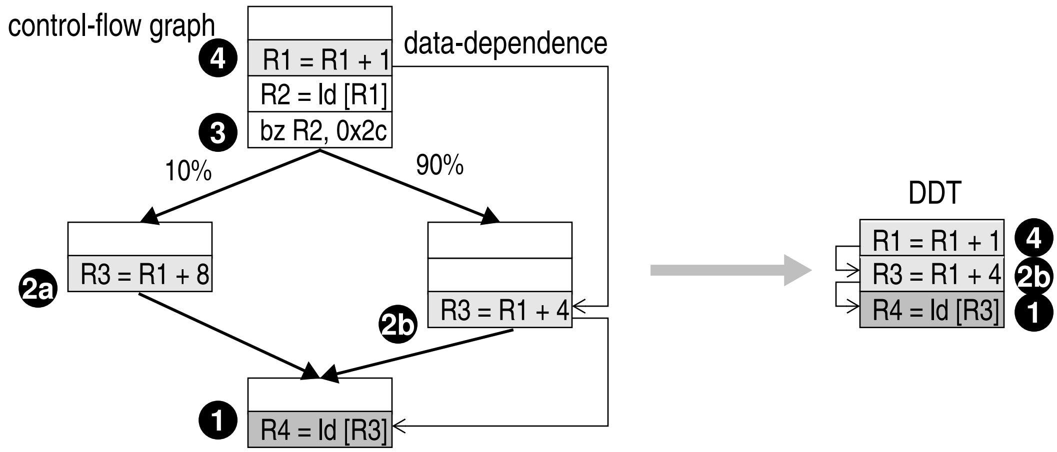
2.2.2 Simulating Conditional Control in DDMT

The answer to coping with the lack of conditional control in DDTs has three parts. First, in modern architectures many conditional control structures are replaced by predication which—being a data-flow representation control-flow—is something DDTs can deal with easily. Second, for unpredicated cases that are statistically biased, conditional control flow is accounted for implicitly within DDTs. Finally, when statistical bias cannot be exploited, a control-less greedy approach suffices. Including predicated instructions in DDTs is straightforward and does not require further explanation. We explain the other two.

2.2.2.1 Implicit Conditional Control

By virtue of containing a given list of instructions, a DDT implicitly encodes the path that list of instructions lies along. There is no need to execute the control transfers that in a control-driven world unrolls this particular path. The purpose of unrolling the path is to expose the instructions that contribute to the computation. If those instructions are exposed in some other way—e.g., via implicit data-driven sequencing—then explicit control is unnecessary. An example of implicit

Figure 2.6 Implicit conditional control in a DDT.



control is shown in Figure 2.6. On the left is a static control flow graph that contains a problem load (marker #1). Again, unimportant instructions are represented by empty boxes. The identity of a particular instruction in the computation of the load (marker #2a or marker #2b) depends on the outcome of a branch (marker #3). We use implicit control if the path is highly biased. Specifically, we construct the DDT to contain instructions only from the biased path (marker #4 and marker #2b). Register integration ensures that the DDT will not be integrated should the program actually take the less frequent path. The data-dependence arcs super-imposed on the control-flow graph are those represented in the DDT.

We should mention that a *pre-execution bias* is not the same as a conventional branch bias. Pre-execution is biased toward the path along which most of the dynamic problem instances occur. A branch may be completely unbiased in the conventional sense, but if problem instances occur only along one conditional arm then the same branch is completely biased from a pre-execution sense. The 90% bias in the figure does not imply that the branch is 90% biased to the right. Rather, it means that 90% of the problem load (marker #1) instances that resulted in a cache miss took place when the path on the right was followed.

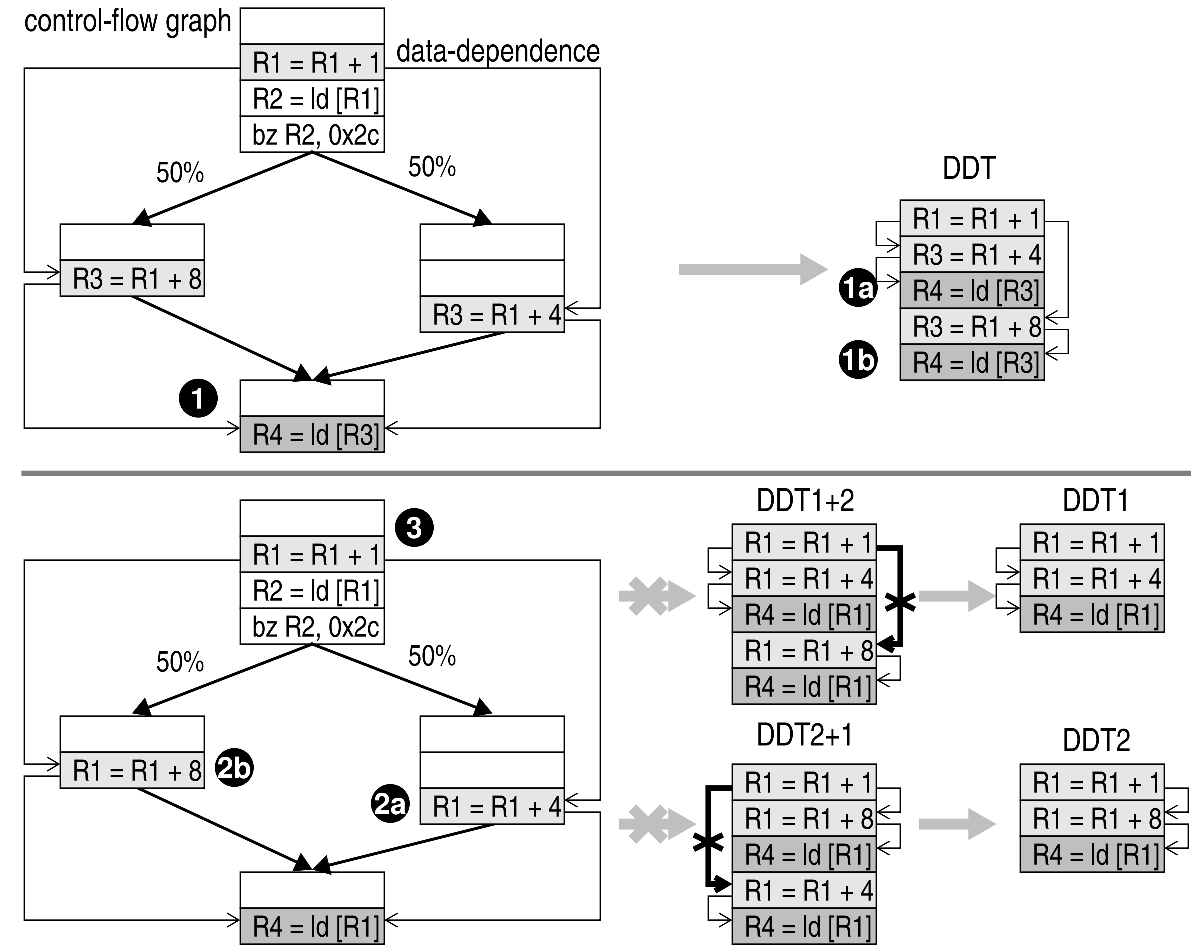
2.2.2.2 Greedy Conditional Control

A strong pre-execution bias may not always exist. In these cases, choosing one particular path via implicit control means sacrificing problem coverage. Fortunately, we don't have to choose. We can execute both paths in the same DDT! This pre-execution idiom is called *greedy control* and is an example of a control flow that is possible in DDMT, but impossible in conventional execution.

The top of Figure 2.7 shows a use of *greedy control*. Of course, once instructions from divergent paths are included within the same DDT, instructions along downstream reconvergent paths must be duplicated. As long as the dataflow graphs along each represented path are intact, integration will naturally reuse the instructions along the pre-executed path that matches the one eventually taken by the master thread. Instructions pre-executed along the other path will not be integrated and will eventually be recycled. In the figure, the problem load (marker #1) must be duplicated along each path encoded within the DDT (markers #1a and #1b). Note, although multiple paths are encoded within the DDT, each DDT instruction actually "observes" the dataflow along one path—its own.

As we foreshadowed in the previous paragraph, it is not always possible to use greedy control. Due to logical register name conflicts in both paths, it may be

Figure 2.7 Greedy conditional control in a DDT.



impossible to create a DDT in which data-dependences for all included paths are faithfully conveyed. An instance of such a scenario is shown at the bottom of the figure. Instructions along separate paths both depend on a pre-branch instruction (marker #3). However, since their output register names also match this dependence, if both are included within the same DDT, the first one would mask the true dependence (marker #3) for the second one. Both greedy DDTs shown on the right (DDT1+2 and DDT2+1) are incorrect. When greedy control is impossible, we are simply forced to choose one path over the other—i.e., revert to using implicit control.

Implicit and greedy conditional control are used in DDMT because there is no other choice. Ironically, however, their use has also been advocated in pre-execution systems that allow control-flow in pre-executing threads [104]. The argument is based on the assumption that most of the cost of a pre-executing thread is in its sequencing bandwidth consumption. Empirically [104], a typical control computation comprises more instructions than the computation it is gating. It is more profitable to sequence a computation and occasionally throw it away than to sequence the computation that decides whether the targeted computation should be executed. Similarly, it is more profitable to sequence two versions of a computation and throw one away, rather than sequence an auxiliary computation that decides which of the two should be executed.

2.2.3 Simulating Loop Control in DDTs

The loop is (by far) the most important pre-execution control idiom. Looping enables a DDT to advance several iterations ahead of the master thread and thereby tolerate longer latencies. An inability to execute loops not only restricts latency tolerance, but also precludes the amortization of DDT startup. Admittedly this last concern is less significant if the microarchitecture supports fast DDT creation, which our proposed implementation does. However, while startup overhead may not be a big issue, limited latency tolerance certainly is.

Before we proceed, we should mention that an inability to execute loops is not a completely bad thing. Along with that limitation comes the inability to execute infinite loops, or at least very long useless loops. DDTs of finite size make for natural and extremely effective overhead control.

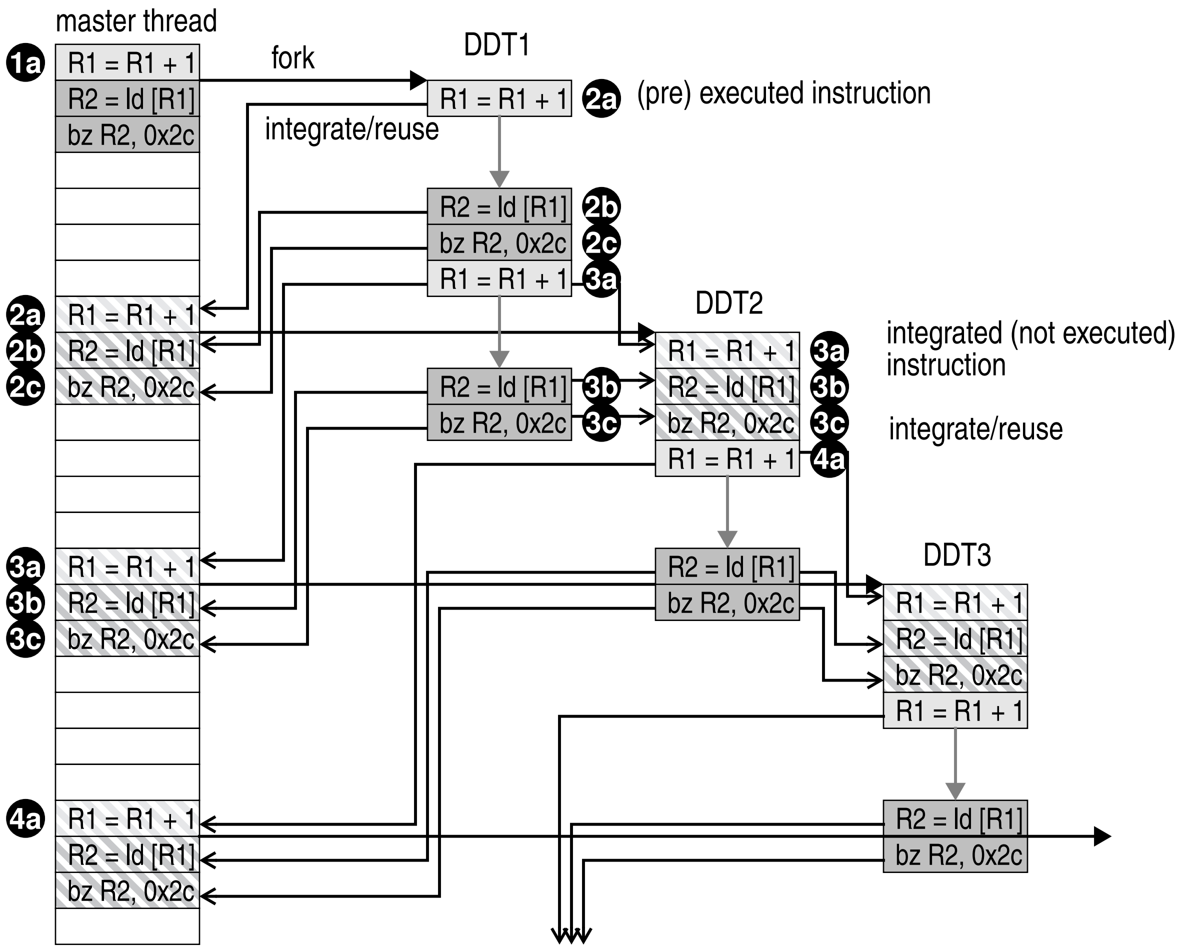
2.2.3.1 Overlapped Full Unrolling

A DDMT processor can be tricked into pre-executing a loop by unrolling the loop within a DDT. DDMT can exploit three kinds of unrolling, although only two are used in practice. We begin by discussing the unused idiom, *overlapped full unrolling*, since its shortcomings motivate the other two.

Full unrolling is DDMT's analog of the loop unrolling performed by compilers. In full unrolling, a DDT contains full problem computations from two or more successive loop iterations. Shown in Figure 2.8, *overlapped full unrolling* is a variant in which a fully unrolled DDT is triggered from *within* a loop by each iteration's induction instruction (markers #1a, #2a, #3a and #4a). In the figure, each DDT contains computations from two successive iterations. DDT1, triggered by the induction instruction of iteration 1 (marker #1a), contains computations from iterations 2 (markers #2a, #2b and #2c) and 3 (markers #3a, #3b and #3c). DDT2, triggered by the induction instruction of iteration 2 (marker #2a), contains computations from iterations 3 and 4. We define the *unrolling degree* of a DDT as the number of times a copy of the trigger instruction appears in the DDT body. The importance of this definition will become clear in Section 2.2.3.2 and in the next chapter, which describes DDT selection. The unrolling degree of the DDTs in the figure is two.

DDMT as described in this dissertation does not support *trigger chaining*—the

Figure 2.8 Overlapped full unrolling.



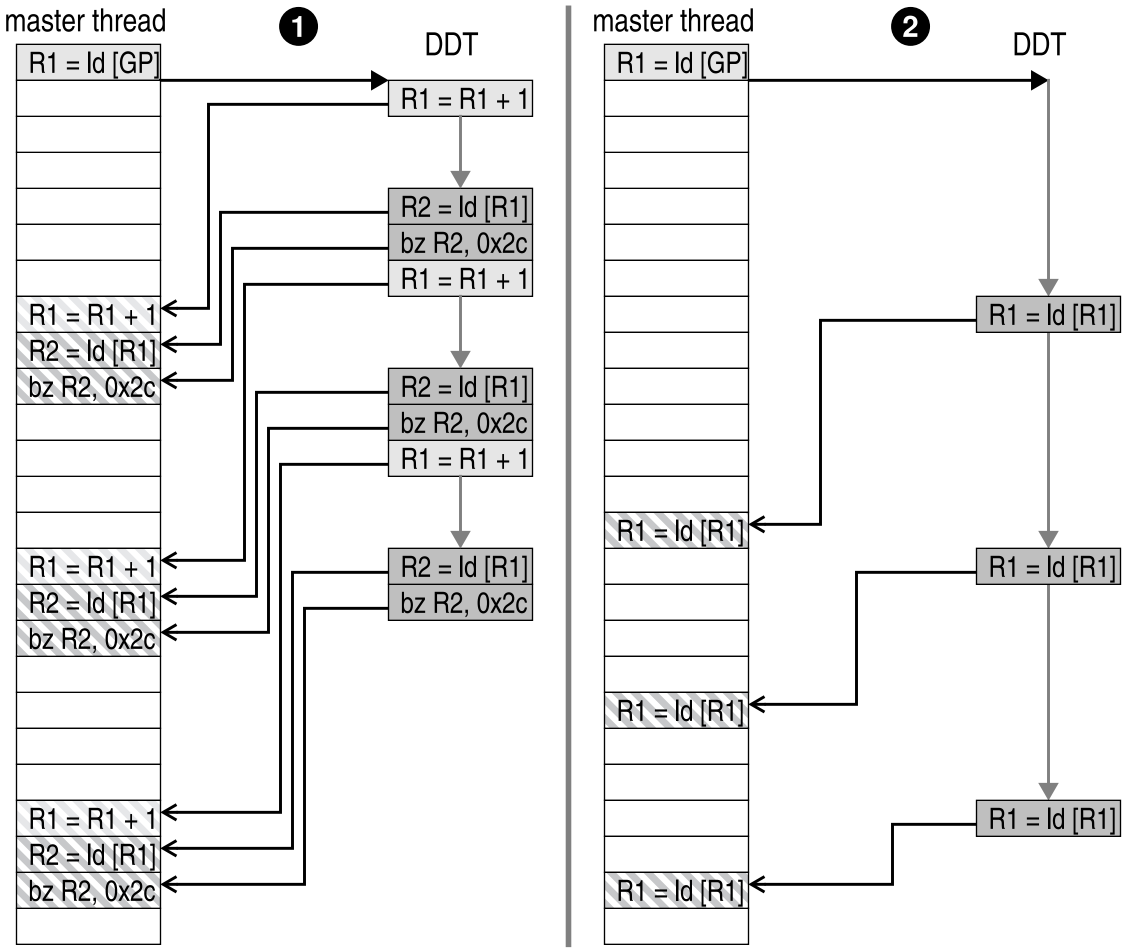
forking of DDTs by other DDTs. In DDMT, only the master thread can trigger DDT forks. The reason behind this design choice will become evident in Section 2.2.3.4. Still, if a fully unrolled DDT is forked by every master thread iteration then every pre-executed master thread instruction will be *sequenced* by a number of DDTs equal to the degree of unrolling. In the figure, the computation from iteration 3 (markers #3a, #3b and #3c) is included in both DDT1 and DDT2. Similarly, the computation from iteration 4 is included in both DDT2 and DDT3. Overlapped full unrolling can exploit DDT-to-DDT register integration to limit each pre-executed master thread instruction to being *executed* by only one DDT. In the figure, the computation from iteration 3 (markers #3a, #3b and #3c) is only executed by DDT1. These instructions are integrated and not re-executed by DDT2. DDT-to-DDT integration of the induction sequence also increases the latency tolerance of each DDT, since by integrating the induction sequences computed by previous DDTs, a DDT effectively executes those induction steps in zero time catapulting itself forward.

The problem with overlapped full unrolling is that its *sequencing overhead* — the number of instructions per DDT that are sequenced but not executed—is $(U-1) * (I + C)$, where U is the degree of unrolling, I is the number of instructions in each inductive step and C is the number of instruction in each single-iteration problem computation. This represents an unacceptably high level of overhead if C is large.

2.2.3.2 Unoverlapped Full Unrolling

A variant of full unrolling that *is* used in DDMT is *unoverlapped full unrolling*, in which a fully unrolled DDT is forked *once* by an instruction *outside* the loop. An example of unoverlapped unrolling is shown in Figure 2.9. Unlike its overlapped cousin, the sequencing overhead of unoverlapped unrolling is always the minimum—zero. Notice, according to our definition of unrolling degree, the unrolling degree of an unoverlapped fully unrolled DDT is always *zero* since the instruction that is replicated within the DDT is not the trigger instruction. It is this definition that allows us to distinguish the two kinds of fully unrolled DDTs

Figure 2.9 Two examples of unoverlapped full unrolling.



from one another .

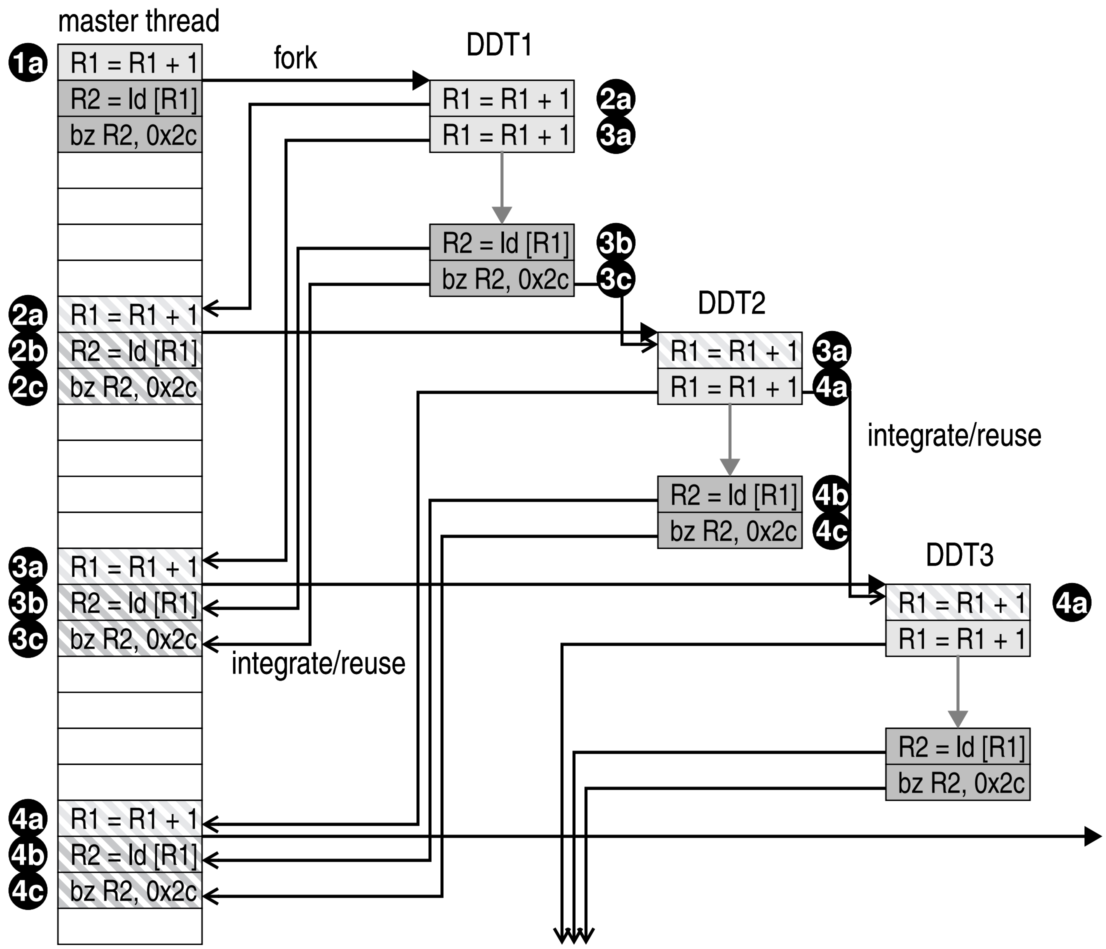
Unoverlapped full unrolling is a powerful pre-execution idiom because it has the potential to tolerate *dependent* latencies that are too long to be fully overlapped by the work of a single iteration. Although they are shown as sequential, the latencies of the loads on the left side of the figure (marker #1) and on all prior figures have, in fact, been parallel. Although inefficient, overlapped full unrolling can tolerate these latencies fully regardless of the amount of work in each iteration, simply by unrolling a sufficient number of iterations ahead. The loads on the right side of the figure (marker #2) are *dependent* —i.e., they are pointer-chases— and their latencies must be taken sequentially, even by a DDT. If the work within a single iteration does not suffice to overlap one of these latencies, then neither will any amount of overlapped unrolling. Essentially, a DDTs latency tolerance over one iteration is negative and any degree of unrolling will just make it more so. However, with unoverlapped full unrolling, pre-loop work can be harnessed to tolerate this latency provided the DDT can be triggered sufficiently early.

Due to the finite size of DDTs, unoverlapped full unrolling is effective only on the first few iterations of a loop. As we will see in the next chapter, our DDT selection framework cannot even recognize an instance of unoverlapped full unrolling unless the loop typically executes only a few iterations. Consequently, the occasion to use unoverlapped full unrolling is limited in DDMT to idioms like hash table searches. The restricted applicability of unoverlapped full unrolling is perhaps DDMT's greatest performance shortcoming.

2.2.3.3 Induction Unrolling

Overlapped full unrolling can be used to hide long latencies in loops as long as these latencies are parallel. The problem with overlapped full unrolling is sequencing overhead as the problem computation of each iteration is sequenced multiple times. *Induction unrolling* attacks this problem directly. In induction unrolling, only the induction sequence itself is unrolled and only one non-induction copy of the problem computation instruction exists in the DDT. Consequently, the sequencing overhead of induction unrolling is $(U\text{-}1) * I$, certainly

Figure 2.10 Induction unrolling.



acceptable as the induction step is typically a simple instruction sequence—a variable increment or pointer dereference , with a possible sa ve/restore pair and stac k pointer manipulation. There is no conventional compiler -based counterpart of induction unrolling , whic h is shown in Figure 2.10. Notice , only the induction instructions (markers #1a, #2b , #3a and #4a) appear in more than one DDT. The computation from eac h iteration (markers #Xb and #Xc) is pre-executed by a only one DDT .

Like overlapped full unrolling, induction unrolling can exploit the latency-tolerance enhancing "catapult" effect that comes with the integration by one DDT of previous induction steps completed by older DDTs. However, induction unrolling has another advantage. Because only the induction sequence is unrolled, an induction-unrolled DDT can sequence multiple iterations ahead of the master thread very quickly as bandwidth is not wasted sequencing the problem computation at every iteration (as it is in overlapped full unrolling). It should now be clear why overlapped full unrolling is never used. If the trigger can be placed outside the loop, unoverlapped unrolling is used, otherwise induction unrolling is used.

Induction unrolling and unoverlapped full unrolling can be used together to pre-execute computations within nested loops. Induction unrolling is used to advance multiple outer-loop iterations ahead. The inner-loop iterations are then unoverlapped fully unrolled.

2.2.3.4 Alternative: Trigger Chaining

One possibility for implementing unoverlapped full unrolling using finite sized DDTs is via *trigger chaining*—i.e., allowing DDTs to trigger the fork of other DDTs. If DDT startup overhead is indeed low, chaining should be roughly equivalent to true unoverlapped unrolling.

The theoretical problem with chaining is overhead in the form of redundant forking. In general, it is difficult to detect and hence prevent the situation of both the master thread and a DDT—or two different DDTs—from triggering what is essentially the same DDT instance. This problem is illustrated on the left side of

Figure 2.11 Trigger chaining.

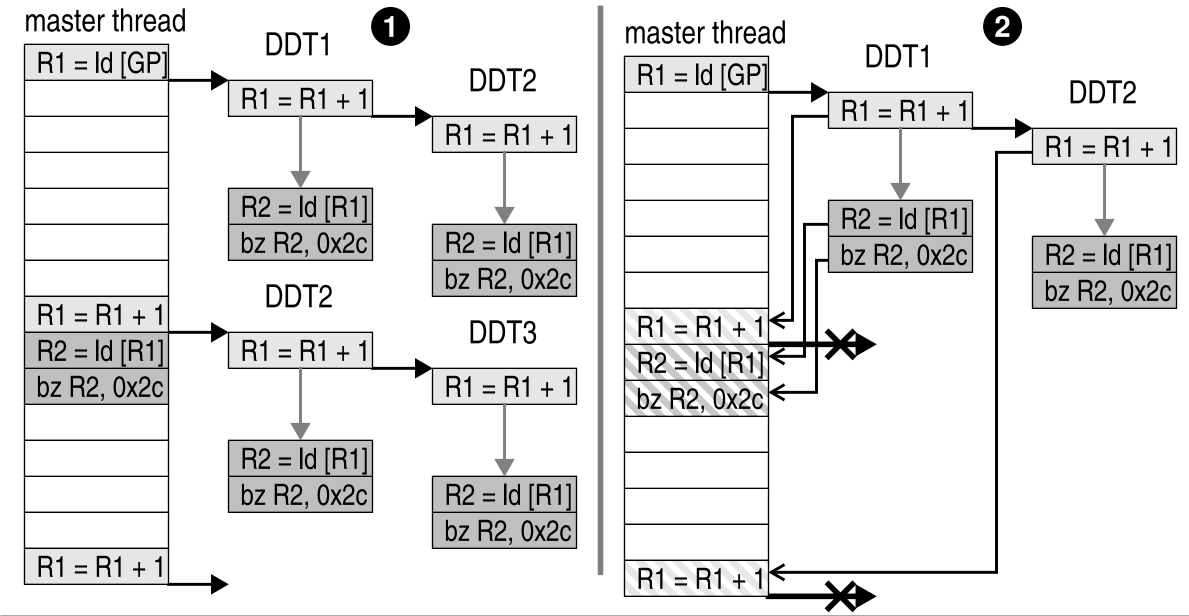


Figure 2.11 (marker #1) which shows both the master thread and DDT1 forking an instance of DDT2. Redundant forking is not a problem if we restrict forking to the master thread only. This is the default DDMT behavior.

That said, DDMT contains a natural redundancy detector—register integration. Redundant forking can be reliably and absolutely eliminated by following one simple rule, namely not forking DDTs from integrating instructions. The obvious reason for this rule is that the instruction that initially computed the integrated result already triggered the fork. In fact, this policy is already used in DDMT to avoid redundant forking by instructions that are squashed and subsequently integrated. The combination of trigger chaining and register integration as a redundancy suppression mechanism is shown on the right of the figure (marker #2).

Recent follow-on work [20] has shown chaining to be beneficial even with crude redundancy control mechanisms. However, chaining is not implemented in DDMT as described in this dissertation. The reason for this is twofold. Chaining can be used to simulate both induction unrolling and unoverlapped full unrolling. Induction unrolling actually performs *better* than chaining, as we will show in Chapter 5. At the same time, chaining's only advantage over DDMT's unoverlapped full unrolling is its ability to handle loops with many iterations. However, our automated DDT selection process cannot capture that case anyway. A static framework that can actually analyze looping behavior explicitly is probably needed to recognize this behavior. Both chaining and static frameworks for DDT construction are important directions for future research.

2.3 DDT Dataflow Structure

The requirements of pre-execution reuse are stronger than simply that DDT instructions must have the same PCs as their corresponding master thread instructions. Pre-execution reuse requires that a DDT match its corresponding computation master thread precisely, instruction for instruction. In other words, for integration to succeed not only must individual instructions of the computation's dataflow graph match precisely, but the entire dataflow connectivity of the graph must precisely match piece-wise as well. This is the *dataflow correspondence requirement*.

The dataflow correspondence requirement implies that DDTs cannot be optimized in any way, even in trivial ways that will not change the outcome of the computation. Even a simple register move instruction, which contributes nothing other than bandwidth consumption to the pre-executed result, may not be removed. Think of register integration as an inductive proof of the correctness of the pre-executed computation with precise dataflow edge matches as the inductive engines. As soon as a single dataflow edge is not a precise match—and any optimization will disturb at least one edge—the induction fails and the ability to prove the correctness of all downstream edges is lost.

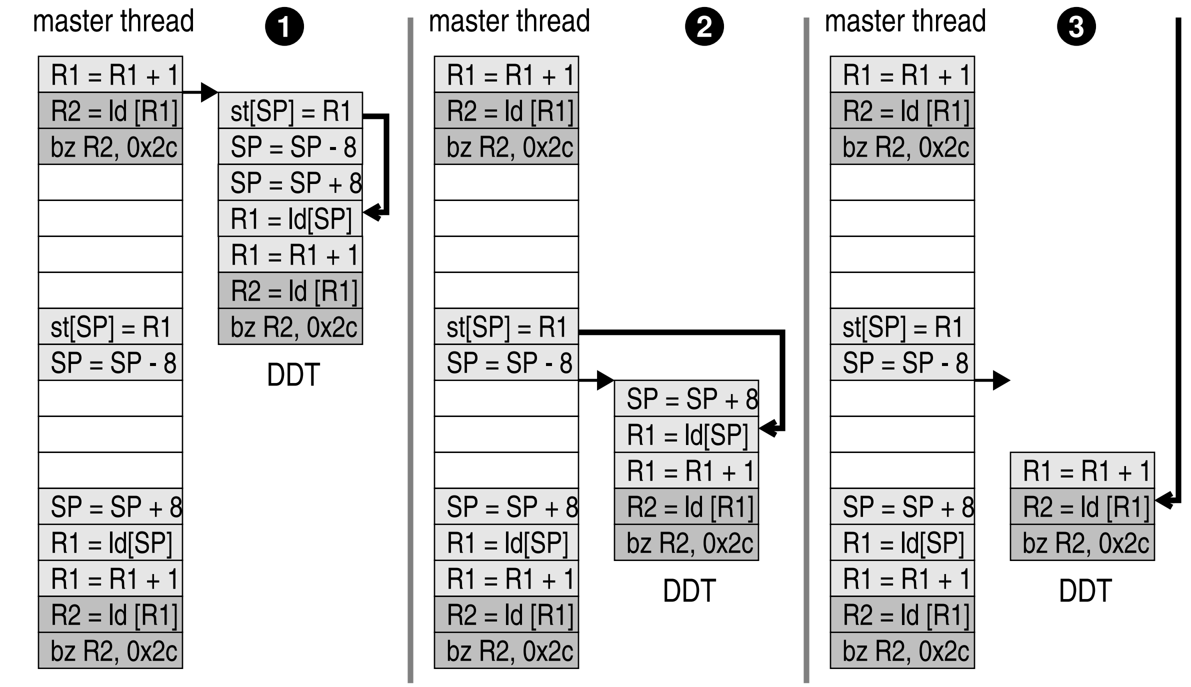
An inability to optimize DDTs detracts from DDMT's performance impact, to be sure. However, it has other negative effects as well. Specifically, the inability to "register allocate" memory communication pairs—i.e., save/restore pairs—within DDTs forces the DDMT microarchitecture to support stores and memory communication within DDTs. Although there are straightforward, localized ways of accomplishing this—as Chapter 4 will demonstrate—it is an added complexity that without register integration could be avoided.

This section will address the role of stores and memory communication within DDTs. A discussion of the *potential* for marrying DDT optimization with register integration is left for the future directions section of the concluding chapter.

2.3.1 Stores and Memory Communication in DDTs

There is only one reason for a DDT to contain a store. Namely, that store was found to supply a value—via memory—to a younger load whose presence in the DDT has already been determined. Stores only exist in DDTs as part of *store-load communication pairs*. A store-load pair within a DDT performs *DDT-internal*

Figure 2.12 DDT-internal and DDT-external memory communication.



memory communication. The left side of Figure 2.12 (marker #1) shows an internal memory communication.

Like all other DDT pre-execution constructs—implicit conditional control, for instance—DDT-internal memory communication is *statistical*. The particular communication need not happen with certainty all of the time. All that is required to justify its insertion into the DDT is that it happens most of the time, or frequently enough so that the resulting pre-execution improves performance.

The two other forms of memory communication shown in Figure 2.12 are instances of *DDT-external memory communication*—in which a DDT load reads a value from either a pre-trigger DDT store (marker #2) or the data-cache (marker #3). DDT-external memory communication is not problematic per se. A DDT-external communication cannot be optimized by “register allocation”. At the same time, no special microarchitectural support is required to implement DDT-external communication. An externally-communicating DDT load reads its value from either the data cache or the store queue. In other words, it behaves just like a conventional load from the master thread.

Special hardware support *is* needed, however, to implement DDT-internal memory communication correctly. DDT-internal memory communication cannot take place through the data cache. DDT instructions are speculative and the data cache (as currently defined) holds only non-speculative state. DDT-internal memory communication also cannot take place via the conventional store queue, since the backing store for that structure is, again, the data cache. DDT-internal memory communication, therefore, takes place using a separate structure—the *data-driven store queue (DDSQ)*. Notice, whether or not a given DDT load participates in internal or external communication is statically known. This static knowledge can be conveyed to the processor to allow DDT loads to be properly steered to either the conventional memory system or to the DDSQ. That is the function of the *internal-communication bit* we spoke of in a previous section.

2.4 Chapter Summary

This chapter described DDMT’s pre-execution model. DDMT is unique in the realm of proposed pre-execution implementations, because its choice of a result communication mechanism—register integration—precludes the use of control flow to sequence DDTs. This, in turn, prevents DDTs from using conditional control to ensure that path sensitive computations are pre-executed correctly and more importantly from using loops to run further ahead of the master thread and tolerate longer latencies . DDMT's pre-execution model consists of a set of "tricks" which are used to "simulate" control flow. Implicit data-driven sequencing replaces the control-driven sequencing mechanism. Within this sequencing model, implicit and greedy control take the place of conditional control structures . Full and induction unrolling replace loop structures .

Register integration also disallows the dataflow graph encoded by DDTs from being optimized in any way, requiring the DDMT microarchitecture to support stores and memory communication within DDTs .

Chapter 3

DDT Selection

A *data-driven thread (DDT)* is a computation copied from the main program that pre-executes one or several problem instruction instances (PIIs). DDT selection is the most significant factor in determining the performance impact of DDMT. Good DDTs pre-execute performance-degrading instances (PDIs) using minimal additional sequencing and execution bandwidth. Bad DDTs pre-execute non-performance degrading instances (NPDIs), do so no faster than the master thread, and slow the total system down by consuming too many resources. The goal of DDT selection is to maximize *DDT latency tolerance*—the expected amount of microarchitectural latency (memory hierarchy latency for loads or pipeline latency for branches) that is moved from the main thread to DDTs—and to minimize *DDT overhead*—the expected number of sequenced and executed DDT instructions—required to achieve this latency tolerance. This chapter introduces the problem of DDT selection, describes metrics for estimating the utility of candidate DDTs, and presents an algorithm that uses these metrics to *automati-*

cally extract promising DDTs from program traces.

The restrictions placed on DDTs by register integration—specifically the data-flow graph correspondence requirement—constrain the space of DDTs we may consider to those that occur as (sub-) computations within dynamic program executions. These restrictions turn the completely open problem of constructing DDTs from scratch into the more bounded problem of *enumerating* all possible DDTs and *choosing* from among them. The enumeration portion of the task can be automated by examining program traces.

Just because DDT selection is a search over an automatically generated bounded space, does not mean that finding an optimal solution—a set of DDTs that, when pre-executed, will yield the shortest master thread execution time—is easy. In fact, we do not even know how to approach this problem globally. Our approach is to decompose the problem into smaller subproblems we think we can approximately solve. Then, we ignore potential interactions and assume that "optimal" subproblem solutions collectively yield a global optimum. Concretely, we assume that DDTs are independent of one another—or if they interact, they do so constructively—and channel our efforts towards optimizing individual DDTs with respect to individual problem instruction instances. In this enterprise, we take into account a number of factors including latency tolerance, bandwidth consumption, and likelihood of eventual integration. Even here, we use the word optimize in the same sense in which it is used when talking about compilers—i.e., we have only approximate ways of telling whether one DDT is better than another and absolutely no way of telling whether a particular DDT is optimal.

The rest of the chapter is organized as follows. We describe the DDT selection algorithm which is composed of three phases. Each phase is described in detail, with a focus on the appropriate metrics that must be optimized at each step. The end of the chapter discusses issues regarding implementation of the algorithm in real world scenarios. However, we stress that it is the collection of metrics and the DDT "optimization" framework that is the real contribution of this work, not any real world implementation.

3.1 Algorithm Overview

All DDTs are *dynamic*, *backward data-dependence slices* of problem instances. We explain this composite term. A *backward slice* is another term for a "computation". A backward slice is trivially constructed by starting with the targeted instruction, walking backwards through a given program representation and adding any instruction that satisfies an unsatisfied input dependence for an instruction already in the slice. A backward *data-dependence* slice is simply a backwards slice that includes only instructions that—transitively—satisfy the target PII's data dependences. Instructions that satisfy control dependences are not included in DDTs because, as we have seen, a DDT cannot interpret control decisions. A DDT *can* contain branches, but only as target PIIs—i.e., computation endpoints. Finally, a *dynamic* backward data-dependence slice is a slice extracted from a dynamic program graph—i.e., a program trace—rather than a static program graph. Our DDTs are dynamic slices because, to be integrated, they must match dynamic program dataflow graphs. Even if we were to construct DDTs using static program graphs, it would be with the sole purpose of unrolling these into dynamic graphs. Doing this without control-flow is difficult.

The selection algorithm we present is trace-driven—it extracts DDTs by examining program traces. A trace-driven approach is natural for performing the dynamic backward-slicing required to find the kind of DDTs we need. Input to the algorithm is a program trace which is annotated with the execution latency of every instruction. In addition, all load and store addresses are annotated in the trace, allowing us to establish memory dependences. The trace is also annotated with all PDIs—load instances that *actually* miss in the cache and branch instances that are *actually* mispredicted.

The reason for distinguishing PDIs from NPDIs—loads that hit in the cache and correctly predicted branches—is an important and recurring theme throughout the chapter, so we will state it officially here. We are only interested in pre-executing PDIs. Since an NPDI has no extra latency to tolerate, pre-executing it only introduces overhead but does not improve performance. Of course, there is no guarantee that the PII pre-executed by a given DDT will be a PDI. In fact, there is no guarantee that for a given PII pre-executed by a DDT, a corresponding master thread PII even exists. However, selecting DDTs using only backward slices of PDIs helps reduce the probability that a pre-executed PII will be an NPDI.

The traces used by our algorithm contain *microarchitectural information*—execution latencies, the identities of all mis-predicted branch instances, etc.—that is not generated by typical architectural tracing tools [9, 86]. The traces we use are generated by a timing simulator. In practice, traces annotated with the appropriate information may be generated in different ways. A microarchitectural simulator is a practical option, especially if DDT selection is performed offline. Alternatively, hardware performance counters [3, 23] or performance-monitoring instructions like informing loads [45] may be used to augment architectural traces. Finally, if DDT selection is implemented in hardware, this microarchitectural information would be naturally available to it.

Our algorithm consists of three sequential phases:

- *Selection of problem instructions (PIs).* We identify the static instructions whose dynamic instances we want to target with DDTs.
- *Selection of single-performance-degrading-instance (PDI) DDTs*. We select DDTs that each attack a single PDI of a particular PI that occur along a particular (but hopefully recurring) path in the program. This is the phase in which we perform the actual backward slicing.
- *Selection of multiple-performance-degrading-instance (PII) DDTs*. We construct DDTs that attack multiple PDIs by essentially merging all sets of single-PDI DDTs with partially overlapping prefixes produced by the previous phase.

The rest of the section will describe each phase and the metrics and analysis it employs in detail.

3.2 Phase I: Selecting Problem Instructions

The first phase in DDT selection is the identification of *problem instructions*

(PIs)—static loads and branches whose performance-degrading instances (PDIs) DDTs will pre-execute. That PIs can be identified via profiling is obvious. What concerns us is the precise definition of what constitutes a PI—i.e., what puts the P in PI.

Ultimately, we are interested in all static instructions for which pre-execution may be profitable. One possibility is to try and construct DDTs for *every* static load and branch with at least one dynamic PDI and decide on a final set of DDTs using the cumulative latency tolerance and overhead metrics calculated for each DDT candidate. However, the second phase of DDT selection is computationally quite expensive and it is wise to restrict the definition of a PI *a priori* using rough profitability metrics. Our algorithm uses a combination of three metrics.

3.2.1 Metric: Problem Ratio

A static instruction's *problem ratio* is the ratio of its performance-degrading instances (PDIs) to its total dynamic instances (PIIs). In other words, a static load's problem ratio is its miss rate and a static branch's problem ratio is its misprediction rate. Since pre-execution benefits only PDIs pre-executed but incurs overhead for all PIIs pre-executed, the PI definition should include some information about the expected fraction of PIIs that are actually PDIs. A static instruction with a low problem ratio is likely not to be a good candidate for pre-execution. Pre-executing instances of such instructions may cover many PDIs. However, the number of DDTs executed will be far greater than the number of PDIs covered, and the increased overhead may result in an overall slowdown.

$$\text{problem-ratio}_A = \#PDI_A / \#PII_A \qquad \text{EQ 1.}$$

The problem ratio is one metric that should not be overly restrictive in the PI identification stage. It is possible that an instruction's PDIs and its NPDIs may occur along distinct sets of execution paths, and that DDTs can be constructed that will be triggered only along the PDI paths. In this case, a DDT would only be pre-executed when the corresponding PII is highly likely to be a PDI. Thus, the problem ratio is not always the best metric, since it does not accurately reflect the metric we care about—PDIs covered to PIIs *pre-executed*. Unfortunately, using path information to separate PDIs from NPDIs is difficult at this point in the process and so this refined metric cannot actually be used.

3.2.2 Metric: Problem Contribution

A static instruction's *problem contribution* is the ratio of its own PDIs to all PDIs of all static instructions—i.e., the fraction of PDIs it contributes to the execution of a program. Problem contribution and problem ratio are complementary metrics. Each on its own is useful but insufficient. An instruction may have a high problem ratio—e.g., ten PIIs, nine of which are PDIs—and be a worse pre-execution candidate than an instruction with a low problem ratio—e.g., one million PDIs out of five million PIIs. Similarly, an instruction may have a higher problem contribution than another and be a worse pre-execution candidate. However, an instruction that has both a higher problem ratio *and* problem contribution than another is almost certain to be the better pre-execution target.

$$\text{problem-contribution}_A = \#PDI_A / \#PDI_{\text{all instructions}} \qquad \text{EQ 2.}$$

3.2.3 Metric: Problem Penalty (Desired Latency Tolerance)

A final metric that may be applied to a static instruction is its average *problem penalty*—the average penalty of its PDIs. The logic behind this criterion is as follows. Even if a branch has a high problem ratio and contribution, do we still want to pre-execute it if its natural (unoptimized) resolution latency is only one cycle—i.e., the branch executes immediately as it enters the window? The answer is no, as it will take at least that one cycle to sequence the DDT itself.

Problem ratio and problem contribution are used to narrow the PI definition—and hence the scope of search—for the subsequent DDT selection phases, but are discarded at that point. In contrast, the problem penalty is relayed to the subsequent phases to let the DDT selection algorithm know *how much* latency a given DDT should attempt to tolerate for a given PI. This information is important, because creating DDTs that attempt to blindly maximize latency tolerance is unwise. There is no sense in creating a DDT that will initiate a 10-cycle L1 cache miss 1000 cycles before the master thread executes the corresponding load. Achieving the 990 additional cycles of latency tolerance will not improve master thread performance and will likely result in a longer DDT that will consume more sequencing bandwidth than absolutely necessary. The "early prefetching" effect, in which prefetched data is evicted before it is used by the master thread, could also become a problem. For DDMT, the early prefetching effect can actually manifest in the physical register file as well as a computed result may be evicted from the IT and its physical register freed before the master thread has a chance to integrate it. Because of this second role, the problem penalty is also called the average *desired latency tolerance (LT_{des})*. It is this second use that motivates our particular definition of the metric.

Our definition of LT_{des} differs for branches and loads. A branch is pre-executed in the hope that its outcome can be integrated, resolving a misprediction. We define LT_{des} for a branch such that achieving that level of latency tolerance will result in the branch being integrated in the completed state, and immediately resolved, providing the full benefit of instantaneous misprediction resolution. Since mis-predictions are resolved at the register renaming stage, LT_{des} for a branch is the original branch resolution latency (in cycles) measured from register renaming—i.e., the difference between the cycle at which the branch was renamed and the cycle at which it was finally resolved. Equation 3 shows the calculation for the average problem penalty of a given static branch.

$$\text{problem-penalty}_{BR} = LT_{des\text{-}BR} = SUM_{PDI\text{-}BR}(T_{complete} - T_{rename}) / \#PDI_{BR} \qquad \text{EQ 3.}$$

LT_{des} for a load is its *execution latency*. Instantaneous completion for problem loads would be nice to have, but enough latency tolerance to make problem loads hit in the L1 cache will still provide a significant ILP boost. The additional latency tolerance needed to provide instantaneous completion—which may also require tolerating the latency of a large portion of the load's computation—may be too costly in terms of additional DDT instructions. Equation 4 shows the problem-penalty/LT_{des} calculation for a given static load.

$$\text{problem-penalty}_{LD} = LT_{des\text{-}LD} = SUM_{PDI\text{-}LD}(T_{complete} - T_{issue}) / \#PDI_{LD} \qquad \text{EQ 4.}$$

The load problem-penalty threshold can be used to target DDT selection either broadly to loads that cause L1 misses, or more narrowly to more expensive loads that miss in both the L1 *and* the L2 cache. Setting the load problem-penalty threshold to the minimum L1 miss latency means that even loads that only miss in the L1 cache can be considered PIs. Setting the load problem-penalty threshold to be higher than the minimum L1 miss latency means that in order to be considered a PI, a static load must cause some number of L2 misses. The higher the threshold, the more L2 misses are required. Of course, setting the problem-penalty threshold above the maximum load latency means that no load—even one that always goes to memory—will be considered a PI.

Notice, in both Equation 3 and Equation 4, problem-penalty/LT_{des} is computed as the average latency over all *PDIs* of a static instruction. The latencies of NPDIs—i.e., correctly predicted branch instances and load instances that naturally hit in the cache—are not entered into this calculation. We want DDTs that tolerate the full latency of PDIs. We would like to minimize the pre-execution of DDTs for NPDIs, but once such a DDT executes we do not care about its latency tolerance properties. Entering NPDI latencies into the calculation, artificially lowers LT_{des} and will cause us to select DDTs that don't tolerate enough latency when they do pre-execute for PDIs.

Note, our definition of problem penalty is based on latency metrics only. It does not account for the impact of that latency on end performance. For instance, the PDIs of a given static load may have an average execution latency that exceeds the problem penalty threshold and yet not adversely impact performance due to an abundance of nearby parallelism. There is no need to pre-execute for these PDIs, yet no way to know within our framework that such pre-execution is useless. In a similar case, the PDIs of a given PI may have an average problem latency of 40 cycles, yet only contribute an average of 10 cycles to execution time, again due to nearby parallelism. Our algorithm would search for DDTs to tolerate 40 cycles per PDI, not realizing that shorter DDTs that tolerate only 10 cycles per PDI are sufficient. A *critical-path framework* [37] that is able to compute how much each PDI contributes to execution time, and hence what its real problem penalty—and desired latency tolerance—is, can potentially refine the DDT selection process and improve end DDMT performance.

3.3 Phase II: Selecting Single-Problem-Instance DDTs

The problem instruction selection phase produces a list of static problem loads and branches for which DDTs should be created. This list is the input to the second phase, which analyzes program traces to construct very simple DDTs that attack a single dynamic instance of a problem instruction along a single (although hopefully frequently recurring) execution path.

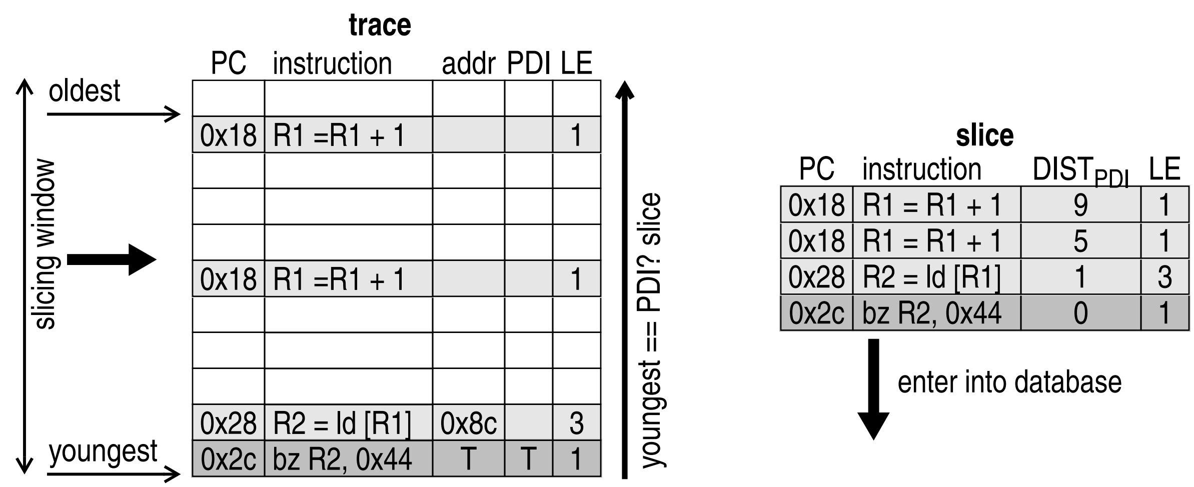
The sequence of instructions that comprises a dynamic backwards slice is unambiguous. Our task is to pick the *slice suffix* (backwards sub-slice) we think will make the best DDT. In this section we describe metrics that help us make this decision. Note, a perfectly valid outcome of this process may be the decision that no sub-slice makes an acceptable DDT.

This phase of DDT selection is the most time consuming and delicate of the three. Actually, we decompose this phase into two sub-phases, one of which is time consuming and the other delicate. The time consuming portion involves processing a program trace and collecting a *database of raw statistics* about all potential DDT candidates for all problem instructions. The delicate portion involves optimizing carefully chosen metrics over this database to come up with the "best" DDTs, this is the portion in which actual DDT selection takes place—the interesting portion. The two sub-phase split is practically motivated. Statistical database construction is both much more expensive *and* almost completely independent of the parameters of DDT selection. The database can be built once, stored, and subsequently post-processed with different sets of parameters to produce different sets of DDTs.

3.3.1 Building the Raw Statistical Database: the Time-Consuming Part

A statistical database of slices is constructed using a list of PIs for which DDTs need to be constructed and a program trace annotated with all load and store addresses, the identities of all PDIs, and the execution latencies of every instruction. The process is illustrated in Figure 3.1. The slicer reads the trace, always remembering the most recent N instructions. These comprise the *slicing window*. When the youngest instruction in the trace is an instance of a PI that is also a PDI, the slicing algorithm walks backwards through the window building a backwards slice for the PDI. Every instruction added to the slice is annotated with its execution latency (LE) and its distance—in dynamic instructions—from the target PDI ($DIST_{PDI}$). Note, $DIST_{PDI}$ is an *architectural measure*—it measures distances in terms of the dynamic retirement stream and is agnostic of out-

Figure 3.1 Slicer and slicing window.



of-order execution. When the oldest instruction in the slicing window is reached, the resulting slice is entered into the slice database. Of course, the slicer is not invoked for NPDIs. The reason for this is simple, but deserves a second mention. We have no interest in selecting DDTs that will pre-execute NPDIs!

The database represents slices as trees with static PIs at the roots. A *slice tree* is not a dependence graph. All nodes in a dependence graph belong to the same computation. The nodes in a slice tree belong to different, but partially overlapping, computations. Each node in a slice tree explicitly represents a single instruction but implicitly represents an entire slice whose trigger is that instruction. The slice is constructed by walking from that node up the tree to the problem instruction root. We use a tree representation because it naturally represents the very important notion of common slice suffixes. In a slice tree, every slice is a suffix sub-slice of all of its child slices.

Figure 3.2 Statistical backward slice database.

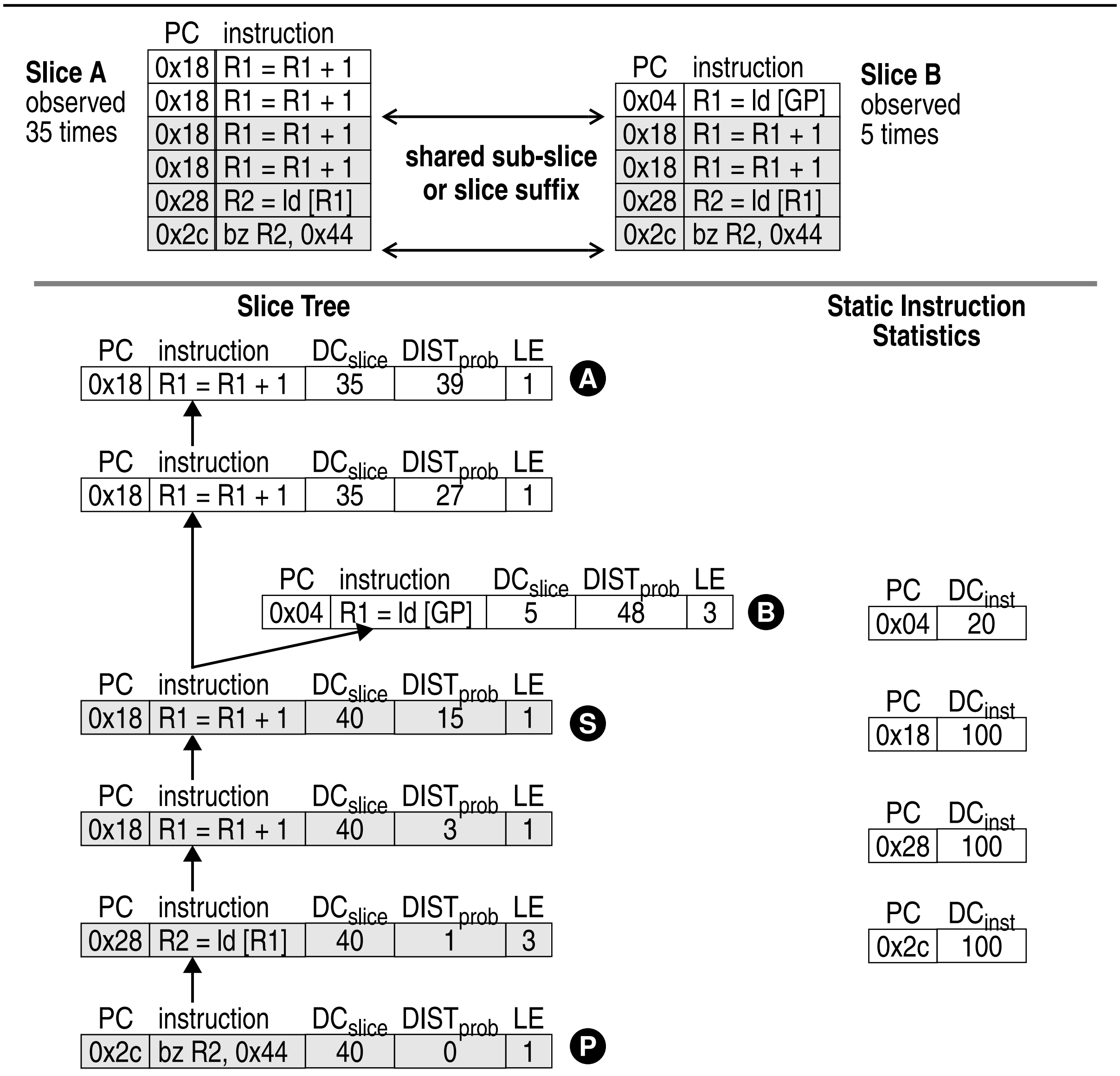


Figure 3.2 illustrates the structure of the database . The top of the figure shows two separate slices , A and B, with a common suffi x. The representation of these two slices in the slice tree is shown at the bottom of the figure . In the slice tree , slice A is represented by the instruction at marker #A, slice B is represented by the instruction at marker #B, and their shared sub-slice is represented by the

instruction at marker #S. Again, a slice is constructed by walking from its representative instruction (trigger) to the problem head of the tree (marker #P). For instance, if we walk from instruction #A to the head of the tree at instruction #P, we recover slice A.

In addition to representing slices compactly, the slice database also collects statistics that help in the DDT selection process. Statistics fall into two groups. The first group comprises a static instruction's *dynamic instance count (DC_{inst})*, the number of times that an instruction appears in a given dynamic program sample (trace). An instruction's dynamic instance count has nothing to do with it being any part of a slice or DDT. Every static instruction has a dynamic instance count.

The second group of statistics apply *only* to static instructions that are part of slices and are *slice-sensitive*—they are kept separately for each static instruction in each position of every slice. For instance, a static instruction's *dynamic slice count (DC_{slice})* is the number of times that instruction appears *in a particular position* in a particular dynamic slice. If an instruction appears multiple times within a given slice, each instance has its own dynamic slice count. Similarly, if an instruction appears in multiple slices, each of those instances also has its own DC_{slice}. The DC_{slice} of a DDT's trigger instruction is equal to the number of times that particular slice appeared as the computation of a PDI. An instruction's DC_{slice} is always equal to or less than its DC_{inst}. Note, the tree structure of our representation allows us to correctly count DC_{inst} for shared slice suffixes. In the figure, slice A was observed 35 times and slice B 5 times. In the tree representation, their shared suffix is marked as having been observed a total of 40 times.

Two other slice-sensitive statistics of interest are an instruction's average *execution latency (LE)* and its average *trigger distances* *($DIST_{trig}$)* for both the control- and data-driven settings. Average execution latency is self explanatory. However, the backward structure of the slice tree and the fact that any instruction in the tree is a potential trigger means that every instruction may need to track multiple average trigger distances. To deal with this information compactly, a slice tree actually store in each entry an instruction's average *problem distance* *($DIST_{prob}$)*—its distance from the problem root of the tree. An instruction's trigger distance from any trigger can then be calculated as the triggers $DIST_{prob}$ minus its own. Notice, by this calculation a trigger's distance from itself is always zero. Also notice, an instruction's problem distance in the data-driven setting does not need to be represented explicitly. This number is always the depth of the path from the given instruction to the problem root. Again, we use Figure 3.2 as an example. We compute the control- and data- driven trigger distances for instruction at marker #S in the DDT with trigger at marker #A. Instruction #S's control-driven trigger distance (24) is computed by subtracting its $DIST_{prob}$ (15) from #A's (39). Its data-driven trigger distance (3) is obtained by subtracting its tree depth (3) from #A's (6).

3.3.2 Algorithm Structure and Hard Termination Conditions

Once the database is constructed, the actual single-PDI DDT selection algorithm traverses slice trees in pre-order—considering increasingly larger slices—

to find the best DDTs. For a given static PI—i.e., a given slice tree—the algorithm may find any number of acceptable DDTs. These DDTs are not mutually exclusive and even if they are, the choice among them will be made by the next and final selection phase.

The traversal algorithm has two hard (as opposed to soft) termination conditions. These conditions only terminate search along a particular sub-tree, not altogether for the entire slice tree, and both signal failure. The first condition is enforced when the current DDT candidate under consideration exceeds a certain length. The *length constraint* is a concession to the finite size of the DDT cache (DDTC). The second condition is activated when the trigger instruction of the current DDT candidate exceeds a certain control-driven distance from the target problem instruction. This *scope constraint* is a concession to the finite slicing window of the slice collector shown in Figure 3.1.

The length and scope constraints are externally tunable parameters that allow us to roughly control the size of DDTs. Longer DDTs typically tolerate more latency per PDI covered, but obviously incur a higher overhead. They typically also cover fewer PDIs.

Within the hard length and scope constraints, the traversal algorithm tries to maximize the cumulative latency tolerance a DDT will have over all of its projected pre-executions while minimizing cumulative overhead. The next subsections—Section 3.3.3 through Section 3.3.7—will formalize these notions in statistical metrics and provide a working example.

3.3.3 Metric: Estimating Execution Times using SCDH

To estimate the latency tolerance of a DDT, we must be able to estimate *a priori* the difference in execution times between a given computation executing in the master thread and that same computation executing as a DDT. To estimate execution time *differences*, we need the ability to estimate *execution times*.

Our execution time estimate is a new metric called *sequencing-constrained data-flow-height (SCDH)*, a composite metric that captures the effects of both data-dependences and limited sequencing bandwidth. For conventional data-flow height calculations, the input height (DH_{in}) of an instruction represents the time at which the instruction becomes data-ready and is computed as the maximum of the output heights (DH_{out}) of those instructions on which it is data dependent. DH_{out}—the time at which the instruction completes—is computed by adding the instruction's execution latency (LE) to its DH_{in}.

$$DH_{in} = MAX_{dataflow\text{-}predecessors}(DH_{out}) \qquad \text{EQ 5.}$$

$$DH_{out} = DH_{in} + LE \qquad \text{EQ 6.}$$

SCDH takes sequencing into account by including a sequencing constraint (SC) in the input height calculation. $SCDH_{in}$ is the maximum of the output heights ($SCDH_{out}$) of an instruction's dataflow predecessors and this sequencing constraint—it represents the earliest time at which the instruction is both data-ready and sequenced. $SCDH_{out}$ is computed in the same way as DH_{out}—by adding the instruction's latency to its $SCDH_{in}$. The SCDH of a single-problem-instance DDT is the $SCDH_{out}$ of its targeted problem instruction. The sequencing constraint (SC) can be calculated in several ways; the simple one we use divides the instruction's trigger distance ($DIST_{trigger}$) by the available sequencing bandwidth (BW_{seq}).

$$SC = DIST_{trigger} / BW_{seq} \quad \text{EQ 7.}$$

$$SCDH_{in} = MAX(SC, MAX_{dataflow\text{-}predecessors}(SCDH_{out})) \quad \text{EQ 8.}$$

$$SCDH_{out} = SCDH_{in} + LE \quad \text{EQ 9.}$$

The SCDH can be used to calculate execution times in both a control-driven (master thread) setting, $SCDH_{control\text{-}driven}$, and a data-driven (DDT) setting, $SCDH_{data\text{-}driven}$. The dataflow relationships and execution latencies are the same in both settings, but the trigger distances ($DIST_{trigger}$) and available sequencing bandwidths (BW_{seq}) are different. In the control-driven setting, trigger distances are non-consecutive while the available sequencing bandwidth is equal to the sequencing bandwidth consumption of the master thread. In a data-driven setting, trigger distances are dense and the available sequencing bandwidth is as much as we decide to give the DDT.

The differences in trigger distances are straightforward. However, the differences in available sequencing bandwidths deserver further treatment. In the control-driven setting, the available sequencing bandwidth is equal to the rate at which the master thread *actually sequences instructions*. In the global limit, this is the master thread's IPC. However, we have found that simply plugging the master thread's base IPC as BW_{seq} produces poor DDTs. As it turns out, the master thread sequences instructions at a higher bandwidth than its IPC. Not only does it sequence wrong-path instructions as well as the retired instructions which are counted in IPC, but its retired-instruction sequencing rate will hopefully increase with the addition of DDMT. After a brief search, we have decided to compute the available sequencing bandwidth for the control-driven master thread as the average of the master thread's IPC and the width of the processor, weighted 2-to-1 in favor of the IPC. Equation 10 gives the precise formula. By using this formula, we guarantee that for a machine of width greater than or equal to three, the control-driven BW_{seq} will never be less than one, leading the master thread to believe that it could sequence the entire program faster than the master thread.

$$BW_{seq\text{-}CD} = (2 * IPC_{CD} + WIDTH_{seq}) / 3 \qquad \text{EQ 10.}$$

As for data-driven BW_{seq}, we choose the constant 1—i.e., we allocate a DDT sequencing bandwidth equal to 1 instruction per cycle. We will address this choice in greater detail in the following chapter in Section 4.2.5.

Ultimately, what interests us are not the absolute execution time estimates, but the difference between them. This difference, $SCDH_{diff}$ quantifies how much faster a given computation will execute as a DDT than it will execute within the master thread.

$$SCDH_{diff} = SCDH_{control\text{-}driven} - SCDH_{data\text{-}driven} \qquad \text{EQ 11.}$$

In Figure 3.3 shows an $SCDH_{diff}$ calculation for the DDT from our running example. We assume that the integer operations have unit latencies and the load has a latency of 3 cycles. The $SCDH_{control\text{-}driven}$ and $SCDH_{data\text{-}driven}$ calculations are each represented by four columns—$DIST_{trig}$ is the instruction's dynamic trig-

Figure 3.3 Sample $SCDH_{diff}$ calculation.

	candidate slice	$SCDH_{control\text{-}driven}$ (BW_{seq} = 4)				$SCDH_{data\text{-}driven}$ (BW_{seq} = 1)			
	instruction	$DIST_{trig}$	SC	$SCDH_{in}$	$SCDH_{out}$	$DIST_{trig}$	SC	$SCDH_{in}$	$SCDH_{out}$
trigger →	R1= R1 + 1	0	0	0	1	0	0	0	1
	R1= R1 + 1	12	3	3	4	1	1	1	2
	R1= R1 + 1	24	6	6	7	2	2	2	3
	R2= ld[R1 + 4]	26	7	7	10	3	3	3	6
	beq R2, 0x201c	27	7	10	11	4	4	6	7

$SCDH_{diff}$
4

ger distance within its execution context, SC is computed by dividing $DIST_{trig}$ by the available sequencing width (BW_{seq}), $SCDH_{in}$ and $SCDH_{out}$ is computed using datafl ow-rules , SC and the corresponding latencies . The calculation is for an 8-wide mac hine . However , we assume that the master thread has a base IPC of 2, so, for the control-driven calculation, we use a sequencing width of 4—(2*2 + 8) / 3 via Equation 10. Since a DDT does not get full sequencing bandwidth but must siphon bandwidth from the master thread, its sequencing bandwidth should be lower. In the calculation, we set BW_{seq} for the data-driven calculation to 1, to simulate the DDT getting to sequence a single instruction every cycle. Our DDMT implementation uses a variant of this policy: on an 8-wide processor a DDT gets to sequence 8 instructions once every 8 cycles. Note , allocating DDTs sequencing bandwidth of 1 instruction per cycle is a design choice, one that will be discussed in Chapter 4.

Figure 3.3 shows where a DDT gets its performance advantage . At DDT fork, the complete program and the DDT both begin executing from the trigger instruction. However, corresponding instructions are always further away from the trigger in the master thread context. Although it has less sequencing bandwidth at its disposal, the DDT manages to sequence the entire computation in 4 cycles, while the complete program takes 7 cycles to sequence it. Combined with data-dependences, this 3 cycle fetch advantage the DDT has over the main thread translates into a 4 cycle execution time advantage. On this particular sequencing configuration, the DDT will cover 4 cycles of latency per mis-predicted instance of the problem branch.

SCDH is an intuitive metric. However, it oversimplifies both the control- and data- driven execution models in ways that typically result in under-estimation of execution times. Since the metric that concerns us is $SCDH_{diff}$, an under-estimation of $SCDH_{control\text{-}driven}$ is conservative, whereas an under-estimation of $SCDH_{data\text{-}driven}$ is aggressive. Under-estimation of $SCDH_{control\text{-}driven}$ occurs because SCDH ignores sources of control-driven sequencing under-utilization—most significantly unrelated branch mis-predictions, but also instruction cache misses, cache block mis-alignments, and other issues—as well the nature of the other work in the main thread. Under-estimation of $SCDH_{data\text{-}driven}$ is a result of the SCDH ignoring contentious parallel execution. Unfortunately, more accurate estimates probably require detail equivalent to full simulation.

3.3.4 Metric: Aggregate Latency Tolerance

$SCDH_{diff}$ is not a measure of latency tolerance per se. Latency tolerance is bounded by the amount of latency present to tolerate. $SCDH_{diff}$ is an unbounded measure of the execution time difference between the DDT and the master thread contexts. To calculate the *effective latency tolerance* (LT_{eff}) of a given DDT, we bound its execution time advantage with the latency it is required to tolerate: the desired latency tolerance (LT_{des}) of its target PI.

However, the metric we wish to optimize is not the latency tolerance of an individual DDT but rather its *aggregate latency tolerance* (LT_{agg}) accumulated over all PDIs tackled via its pre-executions. Just as the effective latency tolerance metric does not count latency that needs no tolerance, aggregate latency tolerance does not count latency tolerance if the targeted instance was an NPDI (had no latency to tolerate). The aggregate latency tolerance of a static DDT is therefore its effective latency tolerance per execution multiplied by the number of PDIs attacked (DC_{slice}), *not* its total number of executions (DC_{inst}).

$$LT_{eff} = MIN(SCDH_{diff}, LT_{des}) \qquad \text{EQ 12.}$$

$$LT_{agg} = LT_{eff} * DC_{slice} \qquad \text{EQ 13.}$$

LT_{des} tells the selection algorithm the desired latency tolerance for the PDIs of each PI. What is not set is how much of this desired latency tolerance must be achieved for a DDT to be considered acceptable. Certainly, if a load has an LT_{des} of 40 cycles we would accept a DDT that tolerated 35 cycles per PDI if that was the best DDT we could find. We may or may not want to accept a DDT that tolerates only 5 cycles per PDI covered, however. This behavior can be tuned via an external parameter, the *latency coverage acceptability factor (LCAF)*. An LCAF of 100% means that the LT_{des} must be achieved in full for a DDT to be accepted. An

LCAF of 50% means that even DDTs that cover half the desired latency per PDI are acceptable. Although not a component in any of our metrics, the LCAF allows us to tune DDT selection externally by setting a lower bound on the “quality” of DDTs we are willing to accept.

3.3.5 Metric: Aggregate Overhead

No amount of latency tolerance is desirable if the cost of executing the DDT is greater than the latency tolerated. More concretely, given a choice between a 30-instruction DDT that can hide 5 cycles of latency and a 6 instruction DDT that can hide 4 cycles of latency for the same set of dynamic problem instances, we cannot clearly choose the first over the second. Sequencing 24 extra instructions in the longer DDT is likely to slow the program down by more than the lone cycle that DDT will gain in latency tolerance over its shorter counterpart.

We define the *overhead (OH)* of a DDT as the number of cycles it takes to sequence the DDT. This is the number of sequencing cycles effectively stolen from the master thread by the DDT, and is the most direct and effective way of measuring overhead. Since bandwidth consumption decreases along the pipe—more instructions are sequenced than are executed, more instructions are executed than are retired, and so on—all other forms of contention will be either less than this one or, as in the case of bus bandwidth utilization, not easily estimated.

Since overhead is mostly a matter of opportunity cost, a DDT should only be penalized for stealing bandwidth the master thread was going to consume. For instance, if on an 8-wide processor, we expect the master thread to sequence at half peak bandwidth (i.e., at a sustained rate of 4 instructions per cycle) then a DDT should only be penalized for half of its aggregate bandwidth consumption. One in every two sequencing slots taken by the DDT was not going to be taken by the master thread anyway. We discount the overhead for a given DDT by the expected *master thread sequencing utilization* which we compute as the ratio of its effective sequencing bandwidth to the processor's peak sequencing bandwidth ($BW_{seq\text{-}CD}$ / $WIDTH_{seq}$). Note, we also use the master thread's IPC when computing $SCDH_{control\text{-}driven}$ in our latency tolerance calculation.

Unlike a DDT's positive impact which is expressed only when that DDT pre-executes a PDI, a DDT's overhead is felt every time it executes, regardless of whether the target PII is a PDI or an NPDI and even if a corresponding PII does not exist at all. A DDT's *aggregate overhead* (OH_{agg}) is its per-instance overhead times the number of times it executes—the number of times its trigger is dynamically observed (DC_{inst}).

$$OH = (SIZE_{DDT} / WIDTH_{seq}) * (BW_{seq\text{-}CD} / WIDTH_{seq}) \qquad \text{EQ 14.}$$

$$OH_{agg} = DC_{inst} * OH \qquad \text{EQ 15.}$$

3.3.6 Ultimate Metric: Aggregate Advantage

The ultimate utility metric for a DDT is its *aggregate advantage* (ADV_{agg}), the difference of the aggregate latency tolerance and the aggregate overhead.

$$ADV_{agg} = LT_{agg} - OH_{agg} \qquad \text{EQ 16.}$$

3.3.7 Working Example

To illustrate the function of the metrics within the algorithm, we walk an example traversal in Figure 3.4. The traversal corresponds to the left-hand tree (chain) of sub-slices from Figure 3.2. For each successively longer DDT candidate we show the ADV_{agg} calculation, describe why this calculation corresponds to our intuition, and show the decision made by the algorithm at each stage. For the purpose of this example, we assume that LT_{des} of the problem branch is 4 cycles, the sequencing bandwidth available to the DDT (BW_{seq}) is 1, that sequencing bandwidth available to the master thread is 8, and that the master thread's base IPC is 2. Again, the master thread's effective sequencing bandwidth consumption, $BW_{seq\text{-}CD}$, is 4.

The shortest possible DDT has its trigger as the instruction at marker #1. The problem instruction itself is not evaluated as a trigger because its DDT is empty. At the top of each iteration, we show the $SCDH_{diff}$ calculation. $SCDH_{control\text{-}driven}$ is on the left, $SCDH_{data\text{-}driven}$ is on the right. Trigger distances are obtained from the slice tree in the control-driven calculation and are, of course, sequential in the data-driven calculation. The sequencing constraint (SC) at each instruction is obtained by dividing $DIST_{trig}$ by the available sequencing bandwidth ($BW_{seq\text{-}CD}$)—4 in the control-driven setting, 1 in the DDT setting. $SCDH_{in}$'s and $SCDH_{out}$'s are computed using data-flow rules, SC, and the average execution latencies (LE) from the statistical database. Since the DDT is sequential in both realms, $SCDH_{diff}$ is 0. Consequently, the effective latency tolerance (LT_{eff}) and the aggregate latency tolerance (LT_{agg}) are also 0. However, the DDT does have some overhead. The DDT is of size 1 (SIZE) and is executed 100 times (DC_{inst}). The

Figure 3.4 Working example of single-PDI DDT selection algorithm.

trigger

DDT	CD#	SC	$SCDH_{in}$	$SCDH_{out}$	DD#	SC	$SCDH_{in}$	$SCDH_{out}$
R2= ld[R1 + 4]	0	0	0	3	0	0	0	3
beq R2, 0x201c	1	1	3	4	1	1	3	4

	$SCDH_{diff}$	LT_{eff}	DC_{slice}	LT_{agg}	DC_{inst}	SIZE	OH_{agg}	ADV_{agg}
iteration 1	0	0	40	0	100	1	3	-6

trigger

DDT	CD#	SC	$SCDH_{in}$	$SCDH_{out}$	DD#	SC	$SCDH_{in}$	$SCDH_{out}$
R1= R1 + 1	0	0	0	1	0	0	0	1
R2= ld[R1 + 4]	2	1	1	4	1	1	1	4
beq R2, 0x201c	3	1	4	5	2	2	4	5

	$SCDH_{diff}$	LT_{eff}	DC_{slice}	LT_{agg}	DC_{inst}	SIZE	OH_{agg}	ADV_{agg}
iteration 2	0	0	40	0	100	2	13	-12

trigger

DDT	CD#	SC	$SCDH_{in}$	$SCDH_{out}$	DD#	SC	$SCDH_{in}$	$SCDH_{out}$
R1= R1 + 1	0	0	0	1	0	0	0	1
R1= R1 + 1	12	3	3	4	1	1	1	2
R2= ld[R1 + 4]	14	4	4	7	2	2	2	5
beq R2, 0x201c	15	4	7	8	3	3	5	6

	$SCDH_{diff}$	LT_{eff}	DC_{slice}	LT_{agg}	DC_{inst}	SIZE	OH_{agg}	ADV_{agg}
iteration 3	2	2	40	80	100	3	19	**61**

trigger

DDT	CD#	SC	$SCDH_{in}$	$SCDH_{out}$	DD#	SC	$SCDH_{in}$	$SCDH_{out}$
R1= R1 + 1	0	0	0	1	0	0	0	1
R1= R1 + 1	12	3	3	4	1	1	1	2
R1= R1 + 1	24	6	6	7	2	2	2	3
R2= ld[R1 + 4]	26	7	7	10	3	3	3	6
beq R2, 0x201c	27	7	10	11	4	4	6	7

	$SCDH_{diff}$	LT_{eff}	DC_{slice}	LT_{agg}	DC_{inst}	SIZE	OH_{agg}	ADV_{agg}
iteration 4	4	4	35	140	100	4	25	**115**

trigger

DDT	CD#	SC	$SCDH_{in}$	$SCDH_{out}$	DD#	SC	$SCDH_{in}$	$SCDH_{out}$
R1= R1 + 1	0	0	0	1	0	0	0	1
R1= R1 + 1	12	3	3	4	1	1	1	2
R1= R1 + 1	24	6	6	7	2	2	2	3
R1= R1 + 1	36	9	9	10	3	3	3	4
R2= ld[R1 + 4]	38	10	10	13	4	4	4	7
beq R2, 0x201c	39	10	13	14	5	5	7	8

	$SCDH_{diff}$	LT_{eff}	DC_{slice}	LT_{agg}	DC_{inst}	SIZE	OH_{agg}	ADV_{agg}
iteration 5	6	4	35	140	100	5	31	109

aggregate overhead (OH_{agg}) is computed by taking the total number of DDT instructions sequenced (100) dividing by the processor's sequencing bandwidth (8) and discounting by the master thread's sequencing utilization factor (4 / 8). (100 / 8) * (4 / 8) is 6.25 which we round to 6. The aggregate advantage (ADV_{agg}) therefore is -6. Executing this DDT will actually slow the entire system down. This is obviously not an acceptable DDT.

Moving to the next longer DDT in iteration 2 makes things temporarily worse. Dynamically, the DDT instructions are still too close to the trigger to allow compressed data-driven sequencing to achieve an execution advantage—i.e., a positive $SCDH_{diff}$—over the complete program. Consequently, LT_{agg} is still 0. At the same time, we have doubled SIZE, doubling OH_{agg} from 6 to 12. Since the ADV_{agg} is -12, we must continue. We do not know for sure that the situation will improve, but we have nothing to lose by looking.

Things do improve in iteration 3. The trigger is now a full loop iteration away from the rest of the DDT, allowing data-driven sequencing to accelerate execution. This DDT has an $SCDH_{diff}$ of 2 cycles per pre-executed instance. Since this is lower than LT_{des} of 4, the LT_{eff} is equal to $SCDH_{diff}$. With slice executions (DC_{slice}) expected to attack 40 problem instances, LT_{agg} is 80. Again, the DDT grows in size by one instruction per executed instance, bringing OH_{agg} to 19. However, ADV_{agg} is now positive meaning that executing this DDT will improve performance. At this point, we can stop or choose to go on in search of a DDT with an even higher ADV_{agg}. We choose the latter path.

The situation improves further in iteration 4 as we implicitly begin to use induction unrolling. The DDT now contains two induction instances providing the latency tolerance of two loop iterations. $SCDH_{diff}$ (4) now matches LT_{des} (4) giving an LT_{eff} of 4. Although LT_{eff} doubles, the number of problem instances attacked drops from 40 to 35. Hence, LT_{agg} goes to 160. OH_{agg} increases linearly (with SIZE) to 25. ADV_{agg} for this DDT is 115 cycles, bettering the 61 cycles of the previous DDT and confirming our decision to proceed. Notice, although it attacks fewer problem instances than its predecessor (35 to 40), this DDT is better because it tolerates twice the latency for the problem instances it does attack.

If the DC_{inst} of both the slice and its trigger remain the same, then there is little point in searching any further. Our current DDT already tolerates LT_{des}, any longer DDT will simply consume more overhead. However, we should still proceed because there is always the chance that the longer DDT will cover the same number of dynamic problem instances (DC_{slice}), but execute fewer times (DC_{inst}) to do so. That is not the case in iteration 5. As we explained, the increased $SCDH_{diff}$ does not translate into increased LT_{eff} as there is no more latency to tolerate. At the same time, the increased SIZE increases OH_{agg}, reducing the ADV_{agg}.

The search ends when one of the hard termination conditions—maximum DDT length or maximum slicing scope—is reached. The final DDT is the best one seen to that point. The final DDT chosen for our example sub-tree is the one considered in iteration 4.

3.3.8 Another Metric: Integrability

When pre-executing a given computation, the hope is that ultimately this computation will be integrated by the master thread. Eventual integration is less important for computations that target loads since the cache prefetching effect itself is enough to improve performance. However, it is extremely important for computations that target mis-predicted branches and for supporting induction unrolling. It makes sense that our criteria for a good DDT include metrics that reflect its likelihood, or ease, of eventual integration. If a DDT has a certain shape that makes it difficult to integrate, and that DDT attacks a branch or uses induction unrolling, we may want to reject it.

Recall, in Section 2.1.4 (Figure 2.5) we showed why PC matches are used to differentiate between operationally identical instructions. The reason is that given a choice between integrating the results of two operationally identical instructions, choosing the wrong one means forfeiting the integration of dependent computations. Unfortunately, this scenario can happen even when we use PC matching for integration. Specifically, two instances of the same static instruction may look identical to the integration circuit, but choosing the wrong one will mean forfeiting the ability to integrate downstream computation. It is the potential for this scenario which degrades a DDT's integrability.

Figure 3.5 shows an example. An unrolled computation contains two instances of instruction 0x14, R4 = ld[GP] (markers #1 and #2). Since GP is the only input to these two instances and GP is unchanged, their IT tags {PC=0x14, pin1=p80, pin2=-} look identical. However, it certainly makes a difference which one is integrated by the master thread. If the master thread integrates p6 (marker #3), then

Figure 3.5 A DDT that is difficult to integrate.

master thread

PC	raw instr	renamed instr
0x14	R4 = ld [GP]	? = ld [p80]
0x18	R1 = R1 + R4	p? = p8 + p?
0x14	R4 = ld [GP]	? = ld [p80]
0x18	R1 = R1 + R4	p? = p8 + p?

IT

PC	pin1	pin2	preg
0x14	p80	-	p4
0x18	p8	p4	p5
0x14	p80	-	p6
0x18	p5	p6	p7
0x1c	p7	-	p60
0x20	p60	-	-

DDT

PC	raw instr	renamed instr	
0x14	R4 = ld [GP]	p4 = ld [p80]	1
0x18	R1 = R1 + R4	p5 = p8 + p4	
0x14	R4 = ld [GP]	p6 = ld [p80]	2
0x18	R1 = R1 + R4	p7 = p5 + p6	
0x1c	R2 = ld [R1]	p60 = ld[p7]	
0x20	bz R2, 0x2c	bz p61, 0x2c	

subsequent integration of the rest of the computation will be lost. The next instruction has p8 as one external input and the output of the integrated instruction as the other input. However, there is no instruction 0x18 with inputs p8 and p6. This isn't to say that the processor will *always* choose the wrong physical register, but the probability that it will is high as without any more information the choice must be made randomly.

The DDT selection algorithm can recognize these scenarios. The question is what should it do when it does recognize one? One of three things can be done. The first two options are trivial. The selection algorithm can either err on the side of caution and discard the DDT or throw caution to the wind and select the DDT anyway hoping that correct integration will take place most of the time. There is a third option.

The problem in this scenario—and the basic problem in all such scenarios—is that the inputs of the two instances are *both DDT-external*—i.e., unchanged by older instructions within the DDT—and hence necessarily identical. We can fix this problem by giving each instance of such an instruction a *false register depen-*

Figure 3.6 Integration disambiguation mechanism for difficult-to-integrate DDTs.

master thread

PC	raw instr	renamed
	1	
0x14	R4 = ld [GP]	p4 = ld [p80]
0x18	R1 = R1 + R4	p5 = p8 + p4
0x14	R4 = ld [GP]	? = ld [p80]
0x18	R1 = R1 + R4	p? = p8 + p?

IT

PC	pin1	pin2	**pIDR**	preg
0x14	p80	-	**p8**	p4
0x18	p8	p4	-	p5
0x14	p80	-	**p5**	p6
0x18	p5	p6	-	p7
0x1c	p7	-	-	p60
0x20	p60	-	-	-

DDT

PC	raw instr	**IDR**	renamed
0x14	R4 = ld [GP]	**R1**	p4 = ld [p80]
0x18	R1 = R1 + R4	-	p5 = p8 + p4
0x14	R4 = ld [GP]	**R1**	p6 = ld [p80]
0x18	R1 = R1 + R4	-	p7 = p5 + p6
0x1c	R2 = ld [R1]	-	p60 = ld[p7]
0x20	bz R2, 0x2c	-	bz p61, 0x2c

dence which will be used to distinguish instances of the instruction during register integration. To do this we add a third input register dependence to each IT entry and annotate DDT instructions with an additional logical register name—*the integration disambiguation register (IDR)*—that will implement this false dependence.

The solution is illustrated in Figure 3.6. We use R1 as the IDR for the two instances of 0x14. When creating the IT entries for these instructions, we add to the entries in the new pIDR field, the mapping of R1 at the time—p8 and p5 respectively. Now, when the master thread attempts to integrate the first instance of 0x14 (marker #1), the tags for the two 0x14 IT entries no longer look identical. During integration, the current mapping of R1 is read and compared to the pIDR field of both entries. As the mapping of R1 is p8, the first IT entry corresponding to output register p4 is properly chosen for integration.

Integration disambiguation is effective, but does require the integration circuit to potentially read an extra input register for every instruction. Note, this is *not* an ISA change. An IDR is encoded with each DDT instruction in the DDTC, but is not added to instructions from the main program text.

3.3.9 Unrolling and Single-PDI DDTs

This phase of the DDT selection process—the selection of single-PDI DDTs—makes no explicit provisions for unrolling, either of the full or the induction variety. Unoverlapped full unrolling is meaningless in this phase because it requires the presence of multiple PDIs per DDT. As shown in the example, induction unrolling falls out naturally from the DDT selection process. The process as we have defined it is—at this point—blind to the degree of unrolling within the DDT. Because the DDTs are created by dynamic backward slicing, induction unrolling is automatically performed to the level dictated by LT_{des} and the external parameter LC AF.

3.4 Phase III: Multiple-PDI DDTs

Per its name, a multiple-PDI DDT is a DDT that attacks multiple PDIs in a single execution. These PDIs may be multiple instances of the same PI as in a fully unrolled DDT, or instances of different PIs. At first, it may appear that the idea of a multiple-PDI DDT runs counter to the pre-execution philosophy. After all, by inserting instructions unrelated to a given PDI into a DDT, we are potentially introducing sequencing delays with respect to that PDI. This is true in theory. In practice, PDIs tend to be clustered—e.g., a frequently mis-predicted branch that depends on a frequently cache missing load is a common idiom—and to share large portions of their dataflow graphs. A single DDT in which these overlapping sub-graphs are shared and sequenced only once is more efficient than separate DDTs for each PDI which would require the shared portion to be sequenced multiple times.

Phase three of DDT selection involves creating multiple-PDI DDTs by merging single-PDI DDTs with common data-flow prefixes. A DDT's *data-flow prefix* is its trigger instruction plus any contiguous chunk of data-flow sub-graph connected either directly to the trigger instruction or to any other instruction external to the DDT. Input to this phase is a set of DDTs represented as lists of instructions. The component DDTs need not be of the single-PDI variety. The merge algorithm can handle groups of multiple-PDI DDTs as well.

3.4.1 Alternatives to Merging and Their Limitations

The fusion of data-flow prefix-overlapping DDTs is somewhat tricky to perform automatically. Before we describe *how* it is done, we want to say *why* we do it at all—i.e., why we don't just extract DDTs that attack multiple PDIs to begin with. In truth, this *is* an option and one we have explored. However, while it avoids merging, this approach makes the previous phase much more convoluted and more likely to result in sub-optimal DDT selection.

First, just because two PDIs are nearby in the dynamic instruction stream, does not mean that they should be pre-executed by a single DDT. For instance, it may be that both are simultaneously problematic only very rarely. Second, trying to satisfy the latency tolerance criteria of multiple PDIs simultaneously is difficult, especially if these are not compatible—i.e., one would require a very much longer DDT than the other. Finally, since we are constructing DDTs only for PDIs, not for NPDIs, if we were to initially include multiple PDIs in each DDT, the number of DDTs we would need to consider would explode combinatorially. Constructing DDTs for three static PIs simultaneously will result in seven rather than three sets of DDTs—three for attacking each individually, three for attacking any combination of two of the three and one for attacking all three simultaneously.

3.4.2 Merging Algorithm

Merging a general set of arbitrarily shaped DDTs with differing degrees of overlap in their data-flow prefixes is a very complex task. In fact, depending on the shape and exact content of the individual DDT dataflow graphs, a merge may be impossible (we have already seen an example of an impossible merge resulting from greedy control flow in Chapter 2). What our algorithm actually does is perform a simple, heuristic merge and then check to see that the register dependence graph of each of the original DDTs is still intact. If the merged DDT passes the register-dependence test, it is accepted. Otherwise, the merge algorithm removes the component DDT whose register dependence graph was violated from consideration and attempts to perform the merge on the remaining component DDTs. The failure and retrial process repeats until, eventually, a single DDT remains and a trivial merge takes place. Component DDTs are a priori sorted by decreasing aggregate advantage so that less advantageous DDTs are removed from consideration first.

The heuristic merging algorithm is based on control-driven distances from the trigger instruction. These are the same distances ($DIST_{trig}$) used to calculate the input sequencing constraints for our execution time estimator—the $SCDH_{control\text{-}driven}$. Quite simply, instructions from individual DDTs are added to the merged DDT in order of increasing $DIST_{trig}$ with the algorithm proceeding as long as any of the component DDTs have instructions to add to the merged DDT.

Adding an instruction proceeds in three stages. Initially, the first unadded instruction in each DDT is examined, and the one with the lowest $DIST_{trig}$ is selected for addition. Next, all the component DDTs are again scanned. If the first unadded instruction of any component DDT matches—by PC—the selected instruction, then we assume that this instruction and the added instruction are one and the same—i.e., this instruction is shared between two or more component DDTs—and advance the first unadded instruction pointer for that component. Then, the selected instruction is added to the merged DDT. Finally, the trigger distances of all unadded instructions are *adjusted* to pretend as if the most recently added instruction is the new trigger. This is a normalization step that eliminates trigger distance artifacts that can subvert a successful merge.

An important component of the heuristic merge is a mechanism for checking that the interleaving of instructions from different DDTs has left the register dependences of each of the original component DDTs intact. In other words, we want to make sure that the merge process does not create any false register dependences within the DDT. We use a dependence tracking mechanism that is essentially a simulated form of register renaming to accomplish this. Each component DDT as well as the blank merged DDT starts with an empty simulated register map table in which each logical register name is mapped to a sequence number. At the outset, all map tables are identical. The register dependence check is made during the second stage, as we match the first unadded instruction in each component DDT with the instruction we have selected to add. If the PCs of the instructions match, we perform a mock "renaming" of the instruction using both the map table of its component DDT and the one of the merged DDT. In order for register dependences to be preserved, the sequence numbers obtained by each of these renamings must match pair-wise. A mismatch signals that—in this particular merge—an original register dependence has been permanently violated. When the new instruction is added to the merged DDT, we allocate a new sequence number for its output logical register and update the merged DDT's map table, and the tables of all DDT's whose first unadded instruction pointer we advanced. A working example in the next subsection will clarify this process.

Note, just because a register dependence is violated in the naive merge, does not mean that there is *no* merge under which all original register dependences are valid. After all, the trigger distances that guide the heuristic merge are statistical. Theoretically, it is possible to serially perform all possible merges combinatorially and pick the first one that preserves register dependences. However, this is potentially very slow. The heuristic algorithm is quite fast and, as our experience shows, rarely fails to perform a merge when one is indeed possible.

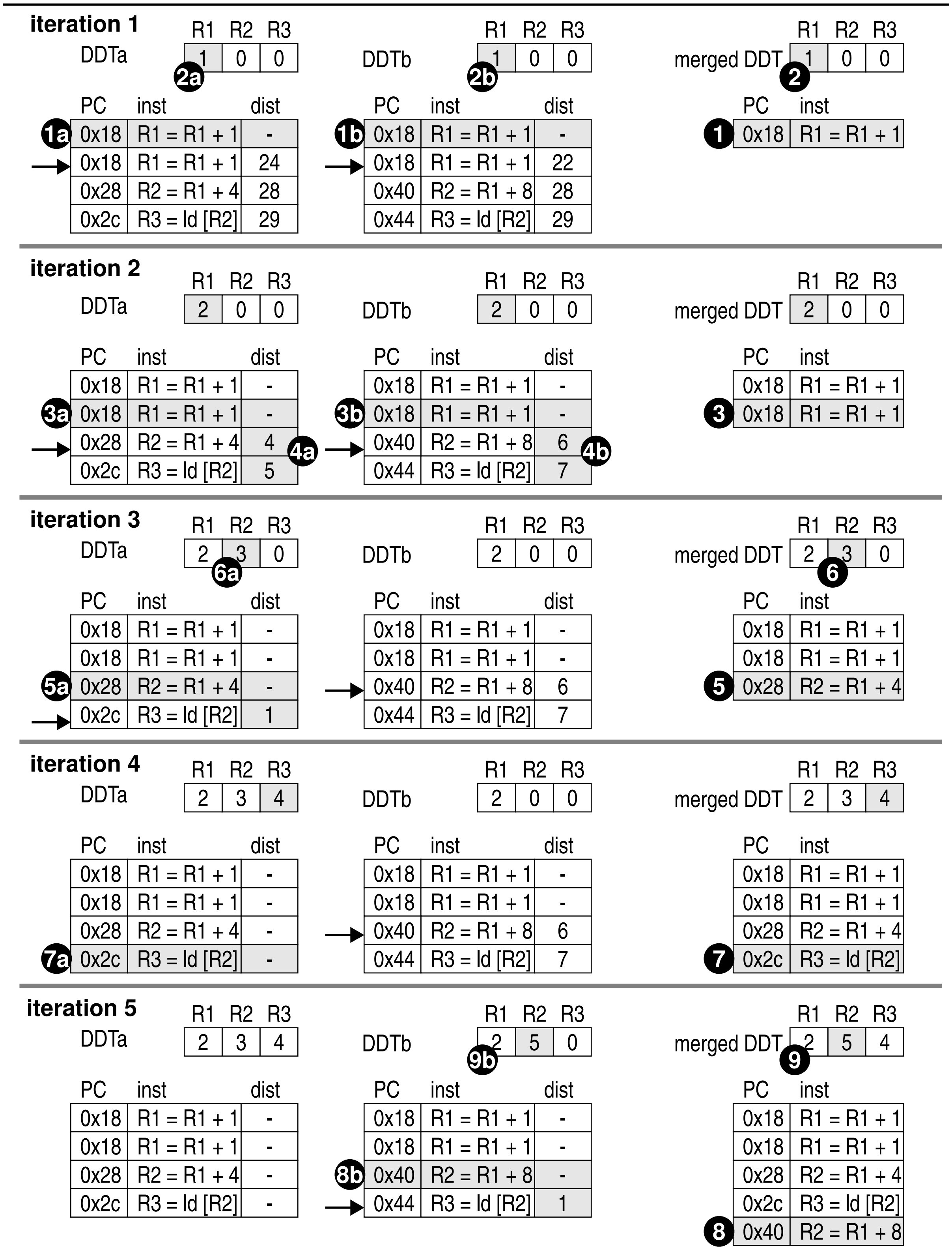
3.4.3 Working Example

We work through an example of the merge process in Figure 3.7. In the figure, we merge a pair of three-instruction DDTs (each DDT has a trigger instruction as well, of course). The DDTs share their trigger and first instruction. The merged DDT has five instructions and a trigger for a total of 6 merge steps. The figure shows the first 5 steps. The sixth step follows trivially from the last step shown and needs no illustration. At each step the participating instructions and structures are highlighted and the important events are marked.

The first step involves merging the two triggers (markers #1a and #1b) together (marker #1). Matching triggers are a requirement of all DDT merges. No dependence violation checks are performed at this point since the trigger does not read any value written by a merged DDT instruction. However, we do track the trigger's own output value via mock renaming for the detection of future violations (markers #2a, #2b and 2). The first unadded instruction for each component DDT is set to the first instruction. There is no need for distance adjustment at this point, as the trigger's distance from itself is zero.

Proceeding to the second iteration, the next instruction that must be added to the merged DDT is the one with the lowest remaining trigger distance. This is the first unadded instruction of DDTb (marker #3b). However, a second scan indicates that this instruction matches the first unadded instruction of DDTa (marker #3a), thus we assume that these two instructions are in fact one and the same. Now that we have decided which instruction to add we must check for reg-

Figure 3.7 Working example of single-PDI DDT merging algorithm.



ister dependence violations. We rename the input register (R1) of the new instruction using the component and merged mock map tables. All three values (marker #2a, #2b and #2) written in the previous iteration match, confirming that all dependences are intact and merging can proceed. The final steps are to add a copy of the instruction to the merged DDT, advance the first-unadded-instruction pointers in both component DDTs, update the mock rename tables, and adjust the trigger distances of all downstream instructions in both DDTs. To perform the adjustment, we subtract the trigger distance of the added instruction from the distances of the remaining instructions (markers #4a and #4b).

Moving to iteration 3, the remaining instruction with the lowest trigger distance is the second instruction of DDTa (marker #5a). Unlike the previous iteration, this instruction does not match the first unadded instruction from DDTb. At this point, we correctly deduce, that the two DDTs are no longer sharing instructions and their remaining graphs must be interleaved in some way. Performing the dependence check again signals no violations. We add the instruction to the merged DDT (marker #5), but advance the next-unadded-instruction pointer, adjust the trigger distances and update the mock renaming table only in DDTa (marker #6a). Of course, we must update the mock renaming table in the merged DDT as well (marker #6). DDTb, which did not contribute an instruction to the merge in this round is untouched.

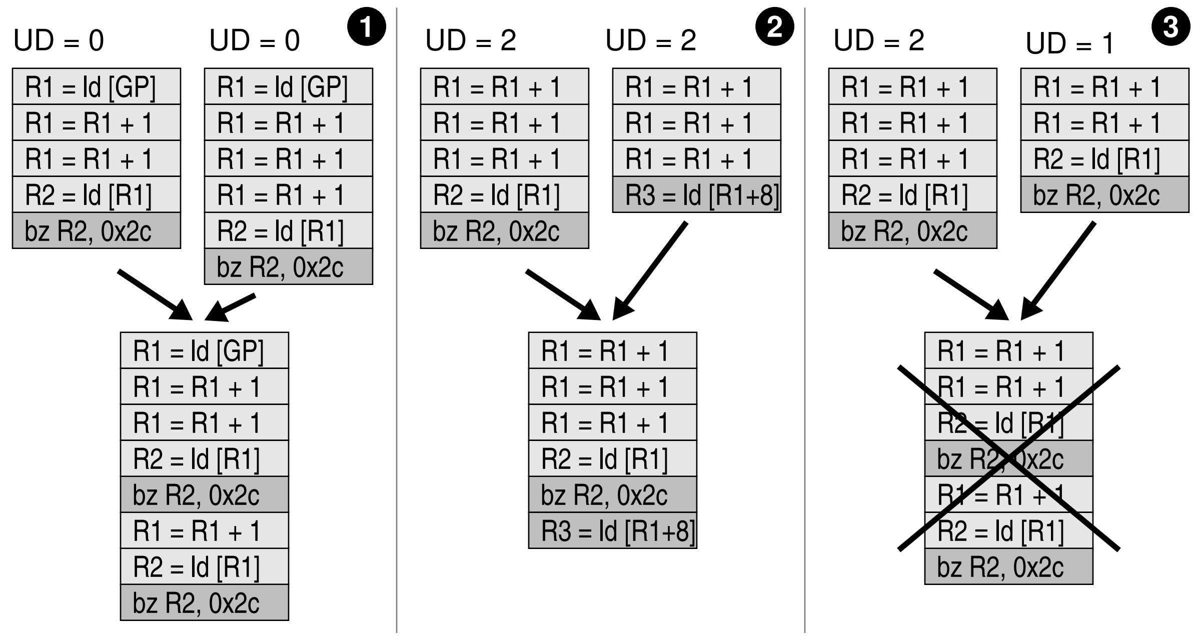
In iteration 4, the remaining instruction with the lowest trigger distance from the previous iteration is the last instruction of DDTa (marker #7a). Again, it does not match the first unadded instruction of DDTb. We repeat the steps of the previous iteration, adding the instruction to the merged DDT (marker #7) and updating the proper mock renaming tables. The next-unadded-instruction pointer of DDTa is advanced past its end. From this point forward, only DDTb instructions are left to add.

The final two iterations are summarized in a single snapshot and their steps are mechanically similar to the ones we have seen. The only point of note is the register dependence violation detection procedure which must verify that by inserting instructions from DDTa, the dependences of DDTb have not been violated. We check the input dependence (R1) of the first unadded instruction (marker #8a) in the map tables for DDTb (marker #9a) and the merged DDT (marker #9). Both entries—2—match, meaning that the merge is still valid. This meshes with our intuition. The current instruction reads R1. However, of the two exclusively DDTa instructions that were added before it, neither wrote to R1. Hence, no name interference takes place between the two DDTs and a successful merge is possible. The instruction is added (marker #8), the next-to-add pointer of DDTb is advanced, and the final instruction in DDTb is added to the merged DDT by passing checks we have already discussed.

3.4.4 Multiple-PDI DDTs and Unrolling

The merging process is the only phase of DDT selection that is explicitly aware of unrolling. In fact, a DDT that employs unoverlapped full unrolling can only be created by merging component DDTs together. As far as unrolling is concerned, the task of the merging phase is to merge as many DDTs together as pos-

Figure 3.8 Merging unrolled DDTs.



sible while not creating any overlapped fully unrolled DDTs .

There is a simple way of accomplishing this. Recall, in Chapter 2, we defined the *unrolling degree* of a DDT to be number of times its trigger instruction appears within its body. We also explained that, under this definition, the unrolling degree of an unoverlapped fully unrolled DDT is zero. The purpose of that definition was to help us with the task of merging unrolled DDTs. If we adhere to this definition, then proper merging of unrolled DDTs can be performed by following one simple rule—only DDTs of equal unrolling degrees may be merged.

Figure 3.8 shows how this definition works in the three different unrolling scenarios. Recall, the first instruction in each DDT is its trigger. The scenario on the left (marker #1) is a case of unoverlapped full unrolling. In this case, the unrolling degree (UD) of each of the component DDTs is zero and merging can take place. In the second scenario (marker #2) the unrolling degrees are again equal, but this time they are non-zero. This merge is also legal as it yields an induction unrolled DDT. Note, although there are two separate PDI computations, they are both from the same loop iteration. The final scenario (marker #3) shows the would-be creation of an undesirable overlapped fully unrolled DDT—PDI computations from multiple loop iterations in one DDT. The merge algorithm avoids merging these two DDTs as they have different unrolling degrees under our definition.

Ultimately, since only one DDT per static trigger instruction is allowed, only a single unrolling degree per trigger is allowed. If a set of component DDTs contains DDTs with multiple unrolling degrees, then equal-unrolling-degree subset with the highest total aggregate advantage (ADV_{agg}) is chosen for merging. DDTs with other unrolling degrees are cast out.

3.5 Implementing the Algorithm in the Real World

Finding good DDTs is a non-trivial, time-consuming task that requires detailed information about the program, and in particular, dynamic executions of the program. These execution costs, algorithmic complexities and information requirements place practical constraints on its implementation. We envision three potential implementations for DDT selection: (1) a static, offline implementation that relies on executable extensions, (2) a purely microarchitectural hardware implementation, and (3) a virtual machine (VM) implementation that is executed in software but below the architectural interface. We briefly discuss each possibility.

3.5.1 Static, Off-line Implementation

In a static implementation, the DDT selection algorithm executes off-line using profile information and communicates the structure of the selected DDTs to the processor via the executable. The off-line entity that performs DDT selection is not the compiler, but rather a post-processing executable-editing tool. DDTs must correspond PC for PC with program instructions which implies that slicing must be done post-assembly. In addition, the executable itself is probably needed to generate problem instruction profiles.

A static implementation is attractive because—at least in its current, preliminary formulation—the DDT selection algorithm must be implemented in software and executed infrequently. The kinds of operations and statistical analysis required are simply too complex for implementation in dedicated hardware, and too expensive in terms of execution time to execute online. However, an offline static implementation must cope with a few issues.

The primary issue is the set of changes required to add DDTs to an executable. Certainly, additional code segments will have to be added in which DDTs can be written. However, the per-instruction format of these additional segments will have to differ from that of the main control-driven code segment. Specifically, PCs of DDT instructions will have to be made explicit. There is also a need for per-DDT instruction annotations, as the conventional instruction format does not have room for the data-driven bit or the integration disambiguation register (IDR). These changes can be formulated as additions *external* to the instruction set architecture (ISA), rather than as ISA changes. Again, the format of the control-driven code segments does not need to be changed.

A second issue concerns the interface between the encoded DDT information, the processor and the operating system. Specifically, there must be a system sanctioned way of loading DDTs from the executable into the DDTC. What is needed here is a handle to an event that will break the circular dependence between demand-loading the DDTC and trigger detection which requires a loaded DDTC. One possibility is to expose the DDTC to the operating system via privileged instructions, then overload the instruction TLB miss handler to load DDTs whose trigger instructions lie within the translated page. With a large enough DDTC, DDTC management could be performed in parallel with instruction TLB management [106].

A final, more theoretical issue deals with the kind of profiling information required to generate DDTs using our algorithm. Generating DDTs that correspond to dynamic backward slices is most easily accomplished using dynamic backward slicing which itself requires program traces. Trace collection is possible, but inconvenient and rarely done. It would be more convenient to generate DDTs using summary profiles and static information. We may be able to generate good dynamic DDTs from accurately synthesized dynamic traces. Useful summary profiles are ones that help in synthesizing such traces. A path profile [10] allows accurate dynamic paths to be re-generated. Given a path profile, a natural extension would be to couple it with path-sensitive cache miss and branch misprediction profiles that tell us not only which static problem instructions to attempt to cover with DDTs, but which paths contain the instances that cause most of the dynamic problems. Prior work has demonstrated that most cache misses lie along a certain subset of paths [3]. Dynamic slicing using a path-sensitive problem profile may, in fact, be a more efficient way of doing the same analysis the trace-based algorithm performs. The trace-based approach collects statistics for all slices and then implicitly chooses the ones that execute along the most frequent program paths. The static approach reverses the process, finding frequently executed paths first.

3.5.2 Dedicated Hardware Implementation

In a hardware implementation, the DDT selection algorithm executes on a dedicated engine. The interface between the selected DDTs and the DDTC is purely microarchitectural and the operating system is not involved.

Although complex and slow in its current form, the DDT selection algorithm we described does not fundamentally preclude a hardware implementation. Its formulation uses dynamically available information exclusively. Selection of problem instructions can be done via a hardware profiler in a straightforward manner. The main operation performed by the second phase—backward-slicing—can also be implemented in silicon. Relatively simple hardware has been proposed for incrementally extracting threads that execute under more restricted models [68, 69]. Incremental extraction is sufficient if multiple-input instructions are included in DDTs. A difficulty with incremental backward slicing is ensuring that slicing proceeds from both input operands along the same dynamic path. Hardware mechanisms for extracting general threads—which include multi-input instructions—by *monolithically* slicing backwards in a hardware instruction buffer are currently being investigated [4, 58]. Dedicated programmable instruction stream co-processors, like the instruction path co-processor (I-COP) [18, 19] or the programmable profiling co-processor [105] may also be adapted to perform backward slicing. Slicing is a good match for these co-processors because they are optimized for processing the retirement stream. Merging common dataflow prefix slices may be difficult to do in hardware. However, this is also a largely incremental step that can be ignored if the main processor core supports the forking of multiple DDTs from a single static trigger instruction.

The component of the algorithm which absolutely must be radically streamlined is the second phase analysis which optimizes the difference of expected latency tolerance and projected sequencing overhead. There are two major aspects that must be simplified. One is the estimation of individual DDT latency tolerance—i.e., the calculated SCDH difference. This is a crucial component. However, the SCDH *is* computed using a modified form of register renaming where accumulated sequencing and execution latencies replace physical register numbers. It may be possible to implement a simple version of SCDH in some sequential circuit and use it to post-process slices. The other is the estimation of the ratio between DDT activations and dynamic trigger instances. Performing this computation faithfully requires multiple slicings and subsequent slice comparison. One obvious simplification is to assume a fixed ratio, say 1 to 3. Given a dedicated retirement stream post-processor, dedicated quasi-renamers for performing SCDH approximations and the statistical simplifications, a full hardware implementation of DDT selection is not implausible.

3.5.3 Co-Designed Virtual Machine Implementation

A co-designed virtual machine (VM) [80] implementation combines the favorable aspects of both software only and hardware only implementations. A VM provides a variable-width—and potentially varying—software layer that sits between traditional hardware and software. This layer allows system software to see a consistent architectural interface while the actual hardware/software boundary between the processor and the VM changes. A VM is an attractive environment for DDMT. It allows DDT selection to be performed in software and the resulting information to be communicated to the processor using hooks and extensions that are architecturally inaccessible. This arrangement eliminates any forward or backward compatibility problems that would arise if the instruction set architecture (ISA), or the executable format were expanded to support DDMT. Also, as DDT selection features become available in hardware, VM sections can be rewritten to take advantage of them. Again, these hardware features need not be exposed at the system architecture level. In this respect, a VM provides an easy evolutionary path for pre-execution and DDMT. Early implementations can use software exclusively. Subsequent, higher-performance implementations can take advantage of some dedicated profiling and slicing features to ease software overhead and expand the applicability of pre-execution.

Very few changes to the given formulation are required for a VM implementation. The input to the thread selection flow is some summary problem information and dynamic program traces, both of which are available to a VM. One concern may be the efficiency and complexity of the extraction algorithms themselves. These must be extremely efficient if they are to run on-line and in parallel with the program whose execution they are trying to optimize. Various statistical approximations can be used to streamline the selection process. Special microarchitectural support can also be provided to move some of the simpler, but more tedious functions to hardware.

3.6 Chapter Summary

Pre-executed computation selection is the most difficult and important problem in pre-execution. An efficient implementation of the runtime component of pre-execution is worthless unless the proper low-overhead, high latency-tolerance computations are selected for pre-execution. An important requirement of any pre-execution implementation is an automated way of finding good computations.

In DDMT, the restrictions placed by register integration on the structure of DDTs simplify the process of automated DDT selection. With DDTs required to match dynamic dataflow graphs instruction for instruction, a simple way of obtaining DDTs is to examine dynamic traces of the program and slice backwards along data-dependence edges from problem instruction instances.

The challenge of DDT selection is not in the mechanical part of backward slicing. The challenge is knowing where to stop a slice. The criteria for a good DDT are obvious. A good DDT should tolerate as much of the microarchitectural latency of its target problem instance and consume as little sequencing bandwidth as possible. Cumulatively, over all of its invocations, a DDT should hide more latency by pre-execution than it adds via bandwidth contention. Much statistical and dataflow analysis is required to quantitatively estimate both latency tolerance and overhead and to judge the cumulative effects of DDTs. This chapter tries to formalize the problem of DDT selection by presenting metrics for estimating DDT benefit and cost and methods for calculating these metrics from raw execution statistics.

The end result is a three phase algorithm that identifies static problem instructions, creates DDTs for those problem instructions by optimizing certain formal criteria over backward slices, and finally merges partially overlapping DDTs together to reduce dynamic sequencing overhead. The algorithm is tunable using parameters that allow us to control the number of PDIs attacked, DDT size and the desired level of latency tolerance.

Chapter 4

DDMT Microarchitecture

This chapter describes a proposed implementation of the DDMT runtime component, focusing on its novel aspects. We propose to implement DDMT on a somewhat modified dynamically-scheduled superscalar processor. The scope of the modifications is even smaller if the processor also supports simultaneous multithreading (SMT) [22, 26, 94, 95, 102]. When designing this implementation, the goal of minimization of the number and scope of the required modifications ranked just below performance in importance. Where modifications are required, we strive to formulate them in such a way that simplification or omission will result in graceful loss of performance improvement, not complete loss of function, or incorrect execution.

The chapter is organized as follows. We open with a description of the base processor assumed by our DDMT implementation, and point out—at a high level—the required modifications. We proceed with a short overview of the important events in the life cycle of a DDT, and point out the novel microarchitectural aspects (if any) required at each stage. The chapter concludes with a detailed description of the structure, function, and management policies of every new or modified mechanism. The most important mechanism we introduce is register integration—a general facility that allows different execution contexts within a single sequential program to share results. Within the context of DDMT, register integration implements the passing of pre-executed results from DDTs to the master thread. In addition to register integration, we discuss the microarchitectural aspects of DDT storage, DDT triggering and initialization, DDT sequencing and execution, and the main thread handling of integrated DDT instructions.

The microarchitecture described here is just one possible implementation for DDMT. For one thing, there are multiple possible implementations of the underlying processor, and these effect choices made for DDMT. In addition, there are multiple implementation choices for different components of DDMT regardless of the base implementation.

4.1 Underlying Processor and DDMT

Our proposed implementation assumes an out-of-order execution core that is implemented in a particular popular microarchitecture style. The preferred style is the *physical register file* style used by the MIPS R10000 [103], the Alpha 21264 [49], and the Pentium 4 [41]. In this style, all register state—speculative and non-speculative—is located in a centralized physical register file. The processor maps logical registers to physical registers, and inside the core all values are referred to by a physical register number (i.e., a pointer to the value). The alternative style is the *register update unit (RUU)* style used in the PentiumPro [43], AMD's K7 [25], and IBM's Power microprocessors [84] in which speculative register state lives outside the register file in an expanded ROB called an RUU. In an RUU style processor, speculative results are "attached" to the in-flight master thread instructions. We prefer the physical register file organization because it allows a speculative result to persist beyond the lifetime of the instruction which creates it and because it more easily supports result sharing via register integration.

We mention optional support for SMT because our implementation requires the presence of multiple active physical register contexts—i.e., multiple tagged copies of the register map table. DDMT also performs better with larger physical register files, which SMT processors have. However, these are the only SMT-specific features used by our proposed implementation. We do not use SMT's other features such as multiple fetch sequencers, a fetch thread-scheduler, a tagged branch predictor, or tagged/replicated sequential core structures like the ROB, and load and store queues. If SMT is implemented, an alternative organization to the one we propose may leverage some of its front-end machinery rather than introducing specialized DDMT machinery.

4.1.1 The Role of a Centralized Underlying Organization

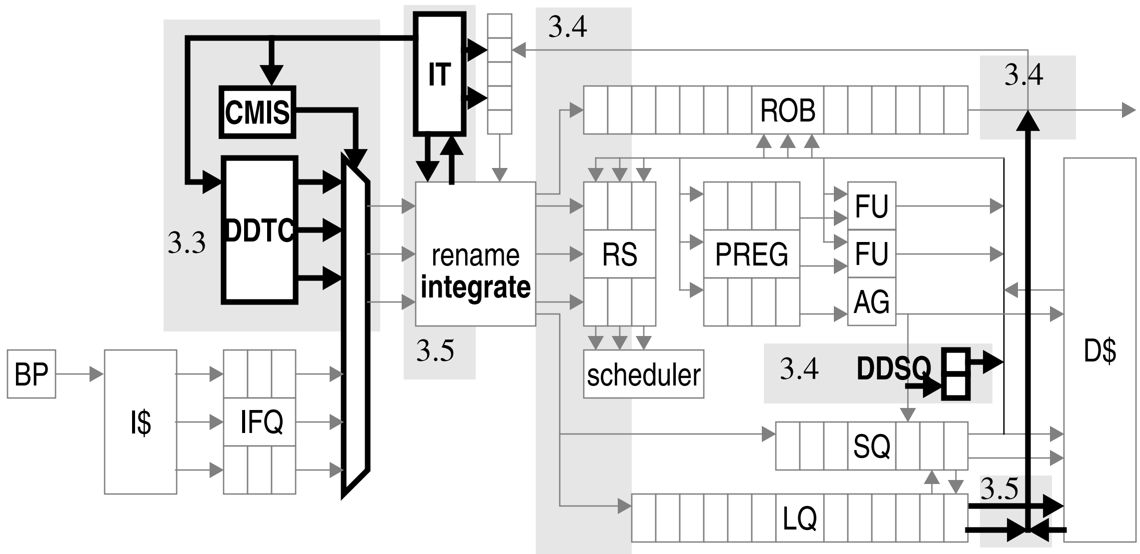
Both a superscalar processor and an SMT processor have centralized execution cores which instructions from multiple threads—in our case the master thread and DDTs—share at a fine granularity. A centralized organization is important to DDMT for several reasons. From an implementation standpoint, it facilitates result sharing between threads. A shared first level data cache allows DDTs to prefetch for the master thread simply by executing the appropriate loads. A shared physical register file enables the implementation of pre-executed result reuse via register integration. From an applicability standpoint, bandwidth and resource sharing means that resources can be diverted to DDTs in a fine grained fashion and on a demand basis. It also implies that full processor utilization is possible even when DDMT is not active. Finally, from a pedagogic standpoint, a centralized organization is attractive because it enables a fair comparison of DDMT-enabled and DDMT-less systems that are very nearly identical from a resource standpoint.

4.1.2 DDMT Modifications

Figure 4.1 shows the DDMT microarchitecture with DDMT-specific additions emphasized. Most of the modifications are small, straightforward, and highly localized. There are three main groups of modifications.

The first group of modifications deals with DDT storage, DDT forking, and the injection of DDT instructions into the pipeline. These changes are restricted to the front end of the machine and will be discussed in Section 4.2. New structures include the *data-driven thread cache (DDTC)* and the combined *context-manager/injection scheduler (CMIS)*. The DDTC is a small instruction memory structure which contains static descriptions—i.e., the static code—of DDTs. The CMIS controls DDT forking, register context allocation, initialization and deallocation, and DDT injection scheduling. In our organization, the DDTC sits in parallel

Figure 4.1 DDMT microarchitecture.



with the instruction fetch queue (IFQ) at the entrance of the register renaming stage. The DDT instructions are injected directly from the DDTC to the register renaming unit.

A second group of changes deals with modifications to the out-of-order execution engine. These are described in Section 4.3. Except for one, these changes are all modifications to the resource allocation and deallocation policies. Although the out-of-order core itself is largely undisturbed, execution resources are allocated differently to DDT instructions as well as to integrating instructions—DDT or master thread instructions that have integrated the result of a previous instruction and need not recompute a fresh value. The one change to the execution core itself is the addition of a small data-memory structure, the *data-driven store queue (DDSQ),* which implements DDT-internal memory communication. The DDSQ sits in parallel with the data cache and conventional store queue (SQ). A given load accesses either the conventional structures or the DDSQ, but not both.

The last and largest group of changes implements register integration. These changes, localized to the register renaming and retirement stages, will be dealt with in Section 4.4. The implementation of register integration requires additions to the register renaming circuit, an additional table—the *integration table (IT)*—to index integrable physical register results, modifications to the physical register allocation and deallocation policies, and modifications to the semantics of mis-speculation recovery. An interface between the load queue and data-cache is added to support *verification of integrated loads via re-execution*.

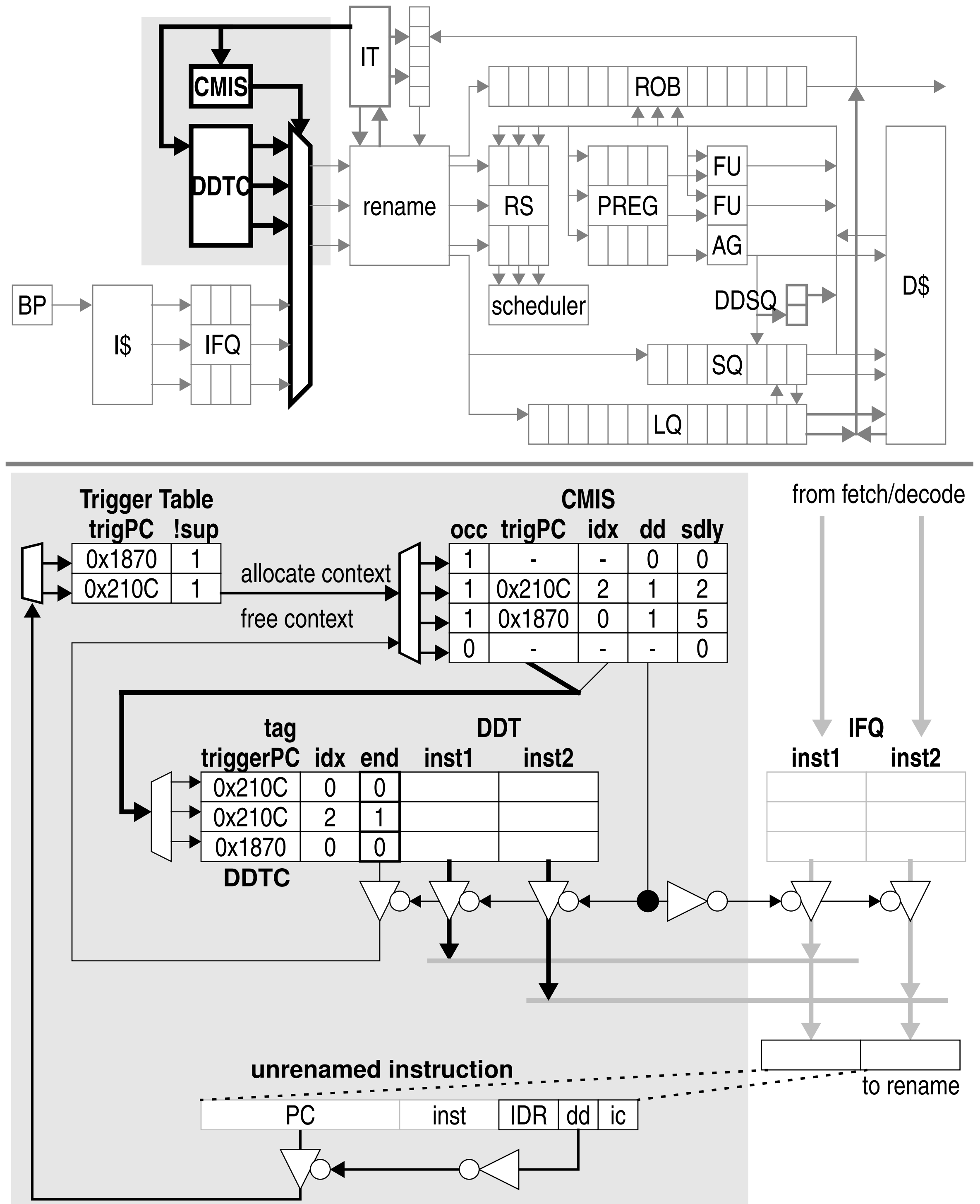
4.2 Front End Aspects

This section describes modifications in the front end of the processor and deals with DDT storage, triggering, context initialization and instruction injection—the process of inserting DDT instructions into the processor.

4.2.1 Pipeline Organization

A high level organization of the DDMT microarchitecture's modified front end is shown in Figure 4.2. The components of a conventional superscalar front end are grayed. The DDMT specific components are data-driven thread cache (DDTC) which stores static descriptors of DDTs, the *Trigger Table* which stores trigger information, and the *register-context-manager/injection-scheduler* which is responsible for managing DDT register contexts and for scheduling DDTs for injection into the pipeline.

Figure 4.2 DDMT front end.



A DDMT front-end pipeline's organization is driven by two important and closely related decisions—(1) the pipeline stage at which triggering should take place and (2) the pipeline stage at which DDT instructions should be injected into the processor. As shown in the figure, in our proposed DDMT implementation both of these take place at the front of the register renaming stage. DDT instructions bypass the control-driven fetch and decode stages and are injected directly into the register renaming stage. In this organization, the DDTC sits in parallel with the head of the IFQ. While placing the DDTC in parallel with the instruction cache and fetching DDT instructions into the IFQ may seem like a more natural choice, our chosen organization provides superior performance as we explain.

A triggered DDT must be initialized with an execution context. This context is a copy of the master thread's register map as it appears immediately after the renaming of the trigger instruction. This copy allows the DDT to pick up external values using register renaming and synchronizes the DDT and main thread with a common mapping that will later be used to start the integration process. The register map copy operation—a "flash copy"—is not an expensive implementation requirement. Support for such operations is typically implemented in physical register file style microarchitectures where it is used to checkpoint the map table and provide fast recovery from branch mis-predictions. The context initialization requirement means the DDT instructions cannot begin executing until the DDT trigger instruction has been renamed by the parent thread.

The combined choice of triggering and injection pipeline stages balances two concerns. To maximize a DDT's latency tolerance capabilities, DDT instructions should be renamed and executed as soon as possible—i.e., immediately after the trigger instruction has been renamed and the DDT register context established. On the other hand, to minimize the incidence of false triggering, triggering itself should take place as late as possible in the life of the trigger instruction, to reduce the likelihood that the trigger itself is mis-speculated. Our chosen organization addresses both criteria. DDT instructions may be injected into the core and begin executing as early as one cycle after context initialization. In addition, fewer mis-speculated instructions are renamed than are fetched, resulting in fewer false triggerings. Injecting DDT instructions at register renaming has the added benefit of ensuring that DDTs are not stuck behind a stalled master thread in the IFQ.

4.2.2 DDTC

Since DDTs do not have explicit control-flow, they cannot be sequenced using a program-counter in the same way in which conventional control-driven code is sequenced. Proper sequencing of a DDT requires an explicit in-order listing of all instructions in that DDT. Storing these explicit descriptions is the job of the *data-driven thread cache* (*DDTC*). The DDTC is a structure that looks very much like a trace cache [67]. Logically, the DDTC is indexed by trigger PC with each entry being a full DDT—a sequential list of all instructions in the DDT, with some additional bits per instruction. However, DDT lengths vary substantially and such an organization may not be the most space efficient. A better DDTC organization is to split DDTs into processor-wide—i.e., 4 or 8 instruction—chunks and index the DDTC using a combination of trigger PC and intra-DDT instruction index. This design not only enables a higher utilization, but also saves the alignment network that would be required to pick the appropriate instruction group out of a particular DDT. A chunked implementation of the DDTC requires that, in addition to its tag and list of instructions, each DDT entry contain a bit that signals whether or not it terminates the DDT. When a terminal chunk is injected, the CMIS is signaled to release the register context and to remove the DDT from the active scheduling list. One point implicit in the above discussion is that a trigger PC is enough to unambiguously describe a DDT, meaning that there is at most one DDT per trigger PC. This invariant can be—and is—enforced by our DDT selection algorithm which fuses DDTs that are triggered by instances of the same static instruction.

The size—in bytes—of the DDTC is a function of the information encoded in it. Logically, for each instruction we need to store its PC, the instruction itself, six bits to encode the *integration dependence register (IDR)* and one bit which for now we call the *internal communication bit (ic)*. The function of the IDR was explained in the previous chapter. We will explain the function of the internal communication bit in Figure 4.3.1.2. For a 64-bit machine, this adds up to 102 bits (roughly 13 bytes) per instruction. There are ways of decreasing the per-instruction storage requirement of the DDTC—for instance, not all PC bits need be represented. However, one size optimization that is obviously a bad idea is not storing the instruction bits themselves, and using the PCs to read them out of the instruction cache. This optimization uses the instruction cache in a way in which it was not meant to be used and one in which it will perform very inefficiently. An instruction cache is laid out to support mostly sequential (control-driven) code access, not to allow parallel access to multiple non-sequential individual instructions. In fact, not only do we want to store actual instruction bits in the DDTC, but placing the DDTC at the entrance to the register renaming stage may require that we store DDTC instructions in an expanded, pre-decoded form.

In this dissertation, we do not discuss the issues surrounding DDTC management. Specifically, we do not name the entity responsible for loading static DDT descriptions into the DDTC. As discussed at the end of the previous chapter, we leave open the possibility of the DDTC being managed by software, hardware, or by a hybrid of the two. Since we do not model the DDTC management entity, we cannot account for the performance effects of a realistic, finite DDTC. Accordingly, our simulations model an infinite DDTC which is magically loaded with the proper DDTs at program load time. However, our experimental evaluation shows that the number of static DDTs used by a program is typically on the order of a few tens. Only in one case did a program use more than 100 static DDTs. Furthermore, it is our *belief* that static DDTs have small working sets associated with different program phases, such that the dynamic cost of DDTC misses would be low.

4.2.3 Trigger Table

Triggering DDTs requires that trigger instructions be flagged in some way. Figure 4.2 shows DDT triggering as taking place with the help of an auxiliary structure called the *Trigger Table*. The Trigger Table is a small, potentially fully associative structure with N ports, where N is the renaming width of the machine. Each Trigger Table entry contains a trigger PC and potentially some state that can be used to *suppress* the forking of that DDT. We will discuss the suppression of DDT fork in Section 4.4.2.

The Trigger Table is accessed by all master thread instructions renamed in a given cycle. Recall, in this dissertation we choose not to trigger DDTs from other DDTs. The data-driven bit on each instruction gates access to the Trigger Table. Trigger Table access bandwidth can be reduced via the use of cached pre-decode bits. An unsuppressed Trigger Table hit results in a request to the register-context-manager/injection scheduler to allocate a free register context to the corresponding DDT and to schedule it for injection.

4.2.4 Context-Manager and Injection Scheduler

Per its name, the *context manager/injection scheduler (CMIS)* is responsible for managing the register contexts—i.e. map table versions—of the processor and for arbitrating control of the register renaming stage between the master thread and any number of active DDTs. The CMIS is a rename-stage analog of the thread fetch scheduler of an SMT processor. If the base processor implements SMT and a fetch-injection organization is implemented, it is likely that the SMT fetch scheduler can be overloaded to perform the functions of the CMIS. However, if we wish to keep a late-injection model, a CMIS will have to be implemented in addition to a thread fetch scheduler, and the two will likely have to coordinate with one another.

The CMIS is a direct mapped structure with as many entries as there are register contexts in the processor. At any time, at least one of these contexts is allocated to a control-driven thread. Each entry contains the following pieces of information. The *occupied bit (occ)* signals that the register context is allocated and on the injection scheduling agenda. The *data-driven bit (dd)* signals that the context is occupied by a DDT. If the data-driven bit is set, the trigger PC and DDT index fields specify the next DDT chunk to inject. The *scheduling delay counter (sdly)* is used in implementing the injection schedule.

4.2.5 Injection Policy

The DDT injection policy governs the rate at which DDT instructions are injected into the processor relative to instruction injection rate of the master thread. An ideal injection policy provides the DDT with enough sequencing bandwidth to allow it to execute as quickly as possible. After all, one of pre-execution's main features is overcoming the sequential fetch bandwidth limitation of control-driven threads. However, injecting DDT instructions too aggressively results in these instructions waiting around in reservation stations which—all things being equal—should be allocated to master thread instructions. Ideally, we would like to inject DDT instructions into the execution core dataflow style—i.e., as soon as their inputs become ready, but no sooner. In fact, this is the sequencing model used by the early pre-execution incarnations including dependence-based prefetching [68] and speculative data-flow [72].

A data-driven sequencing policy is difficult to implement, but we can approximate it with a policy called *DDT-1*. DDT-1 exploits the following observation about the performance of DDTs. A DDT is a computation of a one or more problem instruction instances. This means that most DDTs are close to being true linear dependence chains. The peak execution throughput of a dependence chain is one instruction per cycle. We expect the average execution throughput of most DDTs to be equal to that—most DDTs have a parallelism of slightly greater than one instruction per cycle, but some instructions have longer than unit latencies. DDT-1 injects DDT instructions at a rate that allows the DDT to achieve its expected peak performance—i.e., one instruction per cycle. In our model, instructions from a single thread can be injected each cycle and the policy determines which thread has control of the injection mechanism each cycle. To implement DDT-1 in this model, N DDT instructions are injected once every N cycles on an N-wide machine.

Other possible injection policies are *DDT-priority*, *round-robin*, and ICOUNT [95]. In DDT-priority, DDT instructions are injected in consecutive cycles until completion. Round-robin and ICOUNT interleave DDT and master thread injection—round-robin does so blindly, ICOUNT prioritizes the thread with the fewest number of active resources. Empirically, DDMT is relatively insensitive to the injection policy used.

4.3 Execution Core Aspects

DDMT execution engine modifications come in two flavors—those that deal with DDT instructions and those that apply to integrated instructions. Most of the modifications deal with new semantics for the allocation and freeing of resources. Special support is also needed for DDT memory communication.

4.3.1 Stores and Memory Communication

One issue that must be handled delicately for DDT instructions is memory communication—i.e., store-to-load value forwarding. Recall, to enable integration, DDTs must contain dataflow-contiguous dynamic graphs from the main program. Contiguity implies that stores and memory communication may be included in those graphs as well. It makes no sense to include these instructions if memory communication cannot be implemented correctly. As described in Chapter 2, there are two kinds of DDT memory communication. In *DDT-internal communication* the load reads a value from an older store from within the same DDT. In *DDT-external communication* the load reads a value from either a pre-trigger store from the master thread or from the data cache. The requirements for each kind of communication are different.

4.3.1.1 DDT-External Communication

DDT-external communication has two sub-cases. In the first, the communicating store is still active—i.e., sitting in the parent thread's store queue. In the second, the communicating store has retired to the data cache. In general, these two cases, which differ only microarchitecturally, cannot be differentiated. Nor can they be differentiated from the case in which no communicating store exists at all. Therefore, for DDT-external communication—really for anything but DDT-internal communication—the DDT load must behave exactly like a control-driven load and access the SQ and data cache in parallel, giving data from the SQ priority. Just as a control-driven load checks only the older portion of the SQ, a DDT load checks only the portion of the SQ that is older than its trigger.

Accessing the SQ raises the question of DDT loads' speculation policy—i.e., what the DDT load should do in the presence of older-than-trigger parent stores with unknown addresses. The right thing to do is to issue DDT loads speculatively in these cases. Empirically, waiting for all known store addresses to resolve delays execution significantly and only rarely results in a missed communication [60]. Missed communications do not impact performance unless the load is subsequently integrated, in which case they must be detected and handled. This is the subject of Section 4.4.6.

One way of simplifying the DDT-external memory communication process is to delay the execution of an externally communicating DDT load until all pre-trigger stores from its parent thread have retired. The load must only access the cache in this case. Obviously, this is a lower performance approach and one we do not advocate.

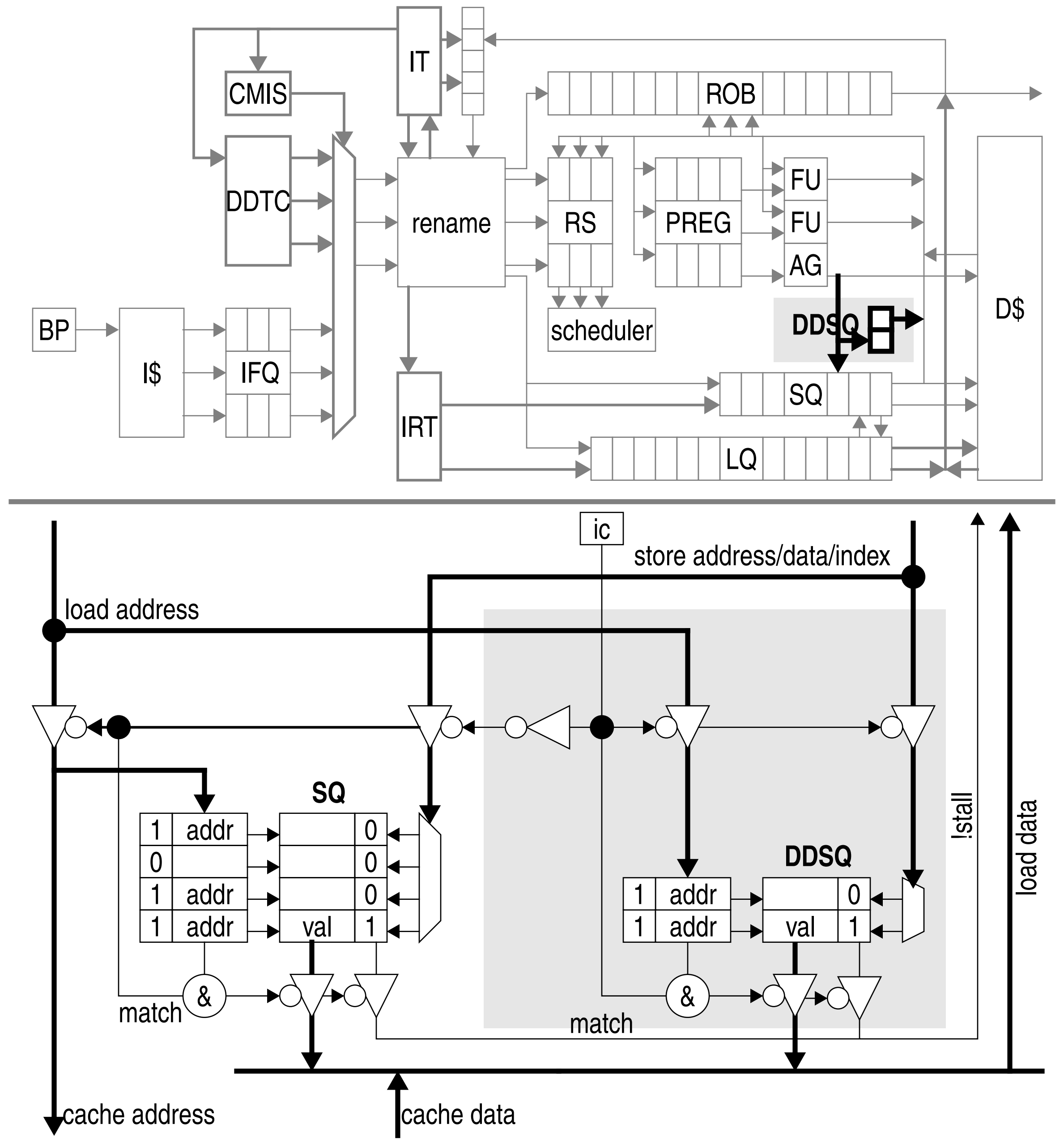
4.3.1.2 DDT-Internal Communication

DDT-internal communication is statically determined at DDT construction time. Theoretically, all DDT-internal memory communication can be "register allocated." However, this optimization will prevent the DDT from being integration. Dynamically there is no question about where the load value should come from, the only problem is that a mechanism is needed to perform the communication. Unlike the DDT-external communication case, here we do not have the option of waiting until the communicating store retires to the data cache as DDT store values are not written to the cache.

The mechanism we use instead is a small store queue that is written only by data-driven stores—the *data-driven store queue* (*DDSQ*). Each entry in the queue contains three elements—an address and a thread context number which are concatenated to make a tag, the store value, and a synchronization bit. A successful DDT-communication can only take place if the tag matches precisely—i.e., the load and store access the same address and are from the same DDT. The synchronization bit is used to delay the load in case the store data is not yet available. The DDSQ allows DDT-internal communication to take place uniformly across a range of dynamic distances without complicating the semantics of DDT stores.

Figure 4.3 shows the setup of the DDSQ in relation to the conventional store queue (SQ) and the data cache. Note, the output signals and data of the DDSQ and SQ are not prioritized with respect to each other. There is no need for the DDSQ to override data found in the SQ and vice versa. A given store or load will access either the DDSQ or the SQ/data cache, but not both. In fact, the two pieces—which in the figure are shown as sharing input and output data paths and signals may in reality be implemented separately. Keeping with the spirit of centralization our implementation models the DDSQ as sharing the conventional load and store scheduling slots, and hence the data paths and signals associated with those slots. Loads and stores are directed to one set of structures or the

Figure 4.3 DDSQ implementation.



other by the statically generated *internal-communication bit (ic)*.

One case that is not handled by the DDSQ is that of a DDT load executing *before* the corresponding DDT store's address has been written into the DDSQ. This is a rare case that is more often a scheduling artifact than a fundamental delay caused by data dependences. DDT-internal store-load communication is typically stack-based—i.e., a save/restore pair. In such a communication, either the communicating parties have the same input address computation or the load has the longer one. Scheduling inversions can be avoided by a scheduling ready memory operations in order as most schedulers do. If an internally communicating DDT load does not find a communicating store in the DDSQ for any reason, it can complete using any value—i.e., the one that is already present in the physical register. The incorrectness of the value does not matter until integration time and will be dealt with then.

There are several other options for implementing DDT-internal communication. Overloading the LQ and SQ is one possibility. This approach adds complexity to already time-sensitive structures and requires that DDT SQ entries not be freed until all potential consuming loads have completed, a tricky constraint to enforce. A second option is to use *speculative memory cloaking* [59], a mechanism that is similar in structure to the DDSQ, but performs tag-based rather than address-based communication. Cloaking is attractive because it is useful in a conventional superscalar context as well. Cloaking learns communication tags by observing conventional communication. If cloaking is to be used for DDT-internal communication, these tags would have to be generated statically.

4.3.2 Allocation and Freeing of Execution Resources

The final modification to the execution engine concerns policies for the allocation and deallocation of execution resources. Modifications must be made for both

DDT instructions and for integrating instructions. Fittingly, the modifications are complementary. Resources that are allocated to DDT instructions and not freed after DDT execution are subsequently not allocated for the integrated version of those instructions while the original resource is freed. We do not discuss the more complex semantics of physical register and IT entry management. These will be covered in the next section which deals with register integration.

The conventional per-instruction resource allocation and deallocation policies are modified by two bits. The statically generated *data-driven bit* distinguishes data-driven instructions from control-driven instructions, and controls the allocation and deallocation of ordering resources: ROB, LQ and SQ entries. Control-driven (master thread) instructions are allocated ordering entries at register rename time, and these are freed when the instruction commits. Data-driven (DDT) instructions are not allocated ordering entries. A data-driven result integrated by a control-driven instruction will have ordering entries allocated for it upon integration (actually, they will be allocated to the integrating instruction).

The dynamically generated *integrating bit* distinguishes integrating instructions from non-integrating ones and modifies the allocation and deallocation of execution resources: RS entries. An instruction that has integrated another instruction's result—whether that instruction be control-driven or data-driven—need not recompute that result. RS entries are freed as usual, when an instruction has issued to the functional units.

The subject of allocating and freeing RS entries brings up the important issue of RS contention. The specific policy decision that concerns us is whether to evict a DDT instruction from its RS when either a new DDT instruction or a master thread instruction about to enter the out-of-order needs an RS entry and none are available. If we decide against spontaneous RS eviction, then the DDT or master thread must stall until an RS is naturally freed by the scheduler. Note, we do not consider evicting a master thread instruction from *its* RS entry as an option.

At first, it may seem that evicting a DDT instruction from its RS to allow a master thread instruction to enter the window is the right thing to do. After all, allowing a DDT to directly cause a stall in the master thread is obviously counter-productive. Empirically, however, the opposite is true. DDTs are typically pre-executed in parallel with low-ILP master thread regions (regions that contain cache misses and mispredicted branches). Hence, it is likely that the master thread instruction that will receive an evicted RS entry will not issue for some time. Waiting an extra cycle or two to give this master thread instruction an RS entry will not change its execution time. Consequently, it is better to stall the master thread and allow the DDT to perform its task as evicting a DDT instruction from its RS entry effectively aborts the downstream DDT computation, making the cycles spent sequencing the DDT wasted ones. Similarly, it is not a good idea to evict one DDT instruction from its RS entry to make room for another DDT instruction. What often happens is that the younger DDT instruction *depends* on the instruction evicted to make room for it. Allowing the younger DDT instruction to enter the window effectively ensures that this instruction will never execute.

4.3.3 Exceptions and Other Disastrous Events

The execution core does not distinguish DDT instructions from master thread instructions (save for DDSQ access). It stands to reason that anything that can happen to a master thread instruction during execution may also happen to a DDT instruction. Actually, even more things that can happen to a DDT instruction than can happen to a master thread instruction.

One concern is the handling of exceptions raised by DDT instructions. Sequential processors defer exceptions until the raising instruction retires, making exceptions precise [31, 79]. We can use register integration to do the same. Simply, we buffer the exception bits in the IT, wait until the corresponding physical register is integrated, then handle the exception when the integrating instruction is retired by the master thread.

Some architectures have hardware handlers for certain types of exceptions. For instance, Intel's P6 microarchitecture [43] has hardware page table walkers that handle soft TLB misses. As hardware TLB miss handling can be invoked speculatively by wrong path instructions, there is no reason why it could not also be invoked by DDT instructions. In fact, by allowing DDT instructions to invoke the hardware TLB miss handler, we are opening a channel for using DDMT to hide yet another source of microarchitectural latency.

Of course, one possible—and always correct—way of handling a DDT exception is simply to abort the instruction and all subsequent instructions within that DDT. This kind of behavior is useful, for instance, when pre-executing the traversal of a linked list. If a DDT instruction attempts to dereference a null pointer, the rest of the traversal is—appropriately—ceased.

Aborting the un-executed portion of a DDT is the solution of choice for handling other exceptional events that may take place during the life of a DDT instruction. One frequent disastrous—to a DDT—event is the freeing of a DDT instruction's DDT-external physical register input. Such freeing takes place when the master thread turns away from the DDT's pre-executed path and overwrites the corresponding architectural register. Continuing DDT pre-execution after this event is allowed—the DDT will execute with garbage values and will not be integrated—but almost certainly useless.

4.4 Register Integration

Register integration is a general technology that exploits the shared centralized physical register file of a dynamically scheduled processor to implement result sharing between instructions. Result sharing is achieved by modifying register renaming to point the output of one instruction to the previously-computed output of another instruction. We call the instruction that initially creates the value the *integrated instruction* and the instruction that subsequently shares the value the *integrating instruction*. The effect of register integration is to obtain a free and instantaneous execution of the integrating instruction.

The crux of register integration is a mechanism that can locate a physical register that contains the value the current instruction will compute, and do so while the instruction is being renamed—i.e., using only information that is available either before or during register renaming—and with high accuracy. Register integration's mechanism for accomplishing this task is the *integration table (IT)*, a table that indexes each physical register using information about the instruction instance that created the value—specifically its PC and input physical registers. During register renaming, an instruction can integrate—i.e., share—any physical register that was created by an instance of the same instruction using the same physical register inputs it itself will use. The rationale for sharing is clear: the same operation (PC) performed on the same inputs (physical registers) will produce the same outputs.

Register integration implements sharing without reading or writing register values. The act of sharing itself is performed by manipulating physical register mappings. The act of deciding that sharing can take place is performed by comparing physical register numbers, essentially "proving" that two instructions are actually instances of the same computation. The no-read/no-write methodology means that sharing is true and that it can take place even before the first instruction has completed execution. When the original instruction completes, the integrating instruction instantly completes with it.

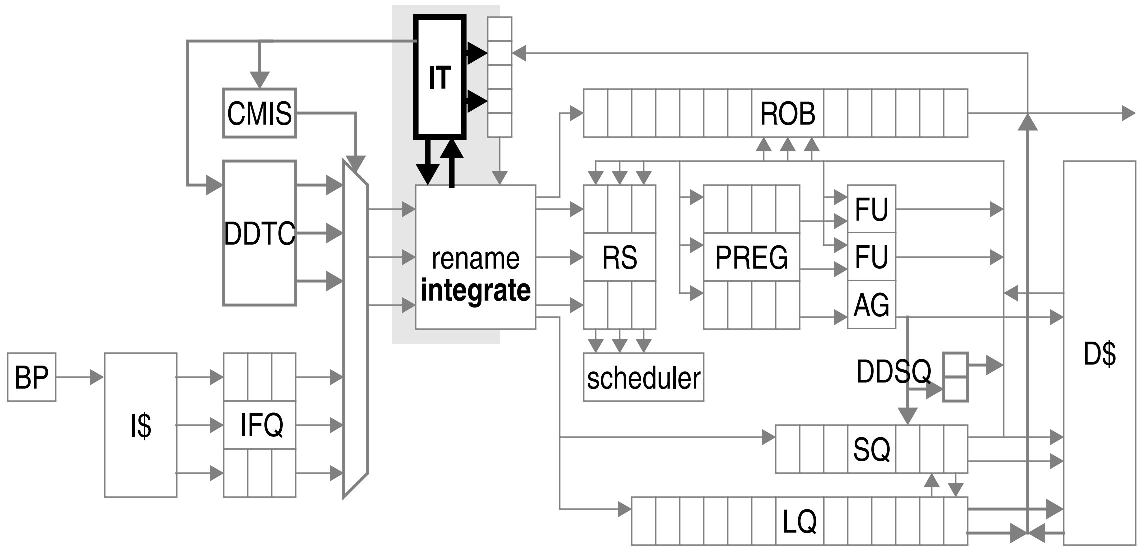
Two important qualities of register integration are its benign impact on correctness and the graceful degradation—and escalation—of its effectiveness. A failure to integrate a computed result does not constitute an incorrect execution, only a lost opportunity to improve performance. Consequently, any physical register actions whose only effect is to limit future integration opportunities (e.g., spontaneously freeing a physical register allocated to a pre-executed result) are always correct. Integration's effectiveness—the number of instructions integrated, and hence, the number of instruction executions saved—increases as integration resources, specifically IT entries and physical registers, are increased.

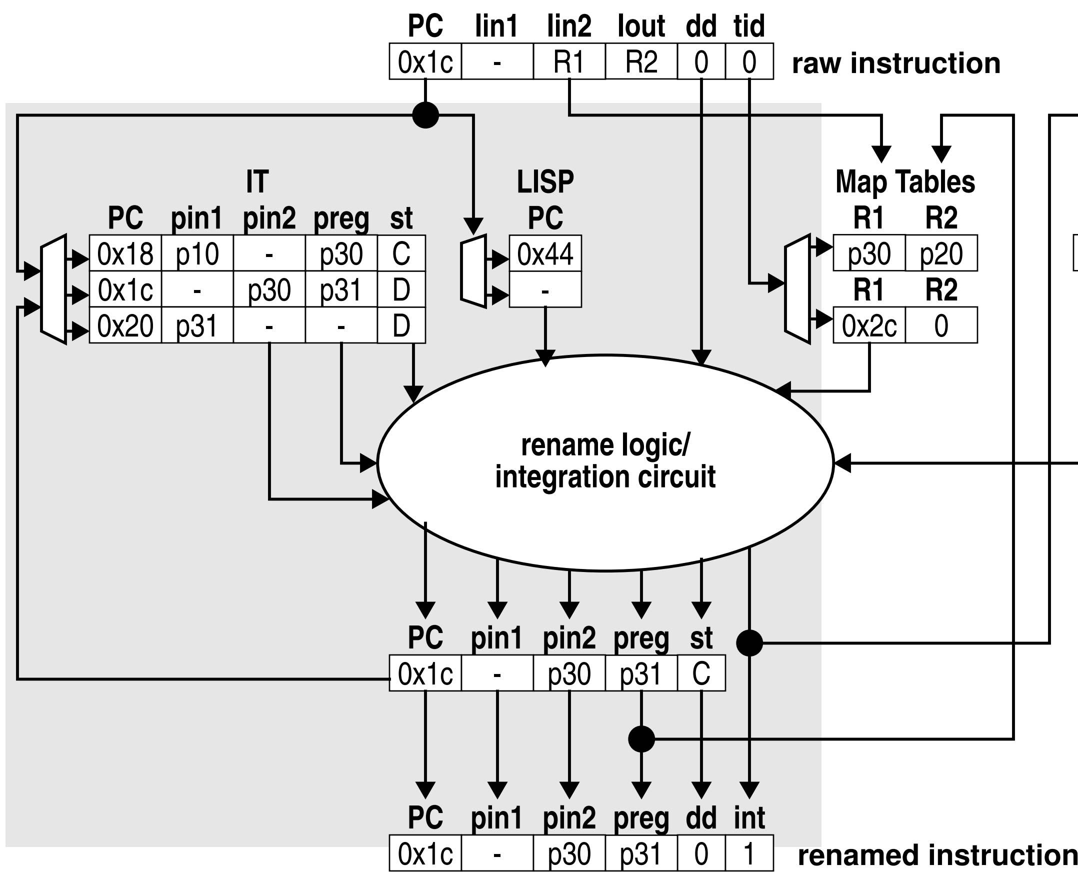
There are two parts to register integration. Most of the action takes place at and around register renaming with some machinery also required at retirement. Figure 4.4 shows a block diagram for the register renaming portion of a processor implementing register integration. The new components are the *integration table (IT)*, the *load integration suppression predictor (LISP)*, and the *integration circuit*. We have introduced the function of the IT. The LISP participates in the integration of loads. The integration circuit takes information read from the map table, free list, IT and LISP and decides which of the currently renamed instructions should integrate which physical registers and which cannot integrate any physical registers and must be allocated new ones. We will discuss these, as well as the retirement stage components, in this section.

4.4.1 Squash Reuse via Register Integration

Chapter 2 introduced register integration as a facility for implementing *pre-execution reuse*, the sharing of DDT pre-executed results by the master thread and by other DDTs. In order to use register integration to implement pre-execution reuse, DDTs must be constructed and initialized in a specific way. As we briefly mentioned in Chapter 2, register integration naturally implements another form of reuse, *squash reuse*. A superscalar processor recovers from a control (or data) mis-speculation by squashing and re-processing—re-fetching and re-executing—all instructions from the point of the mis-speculation forward.

Figure 4.4 Register integration.



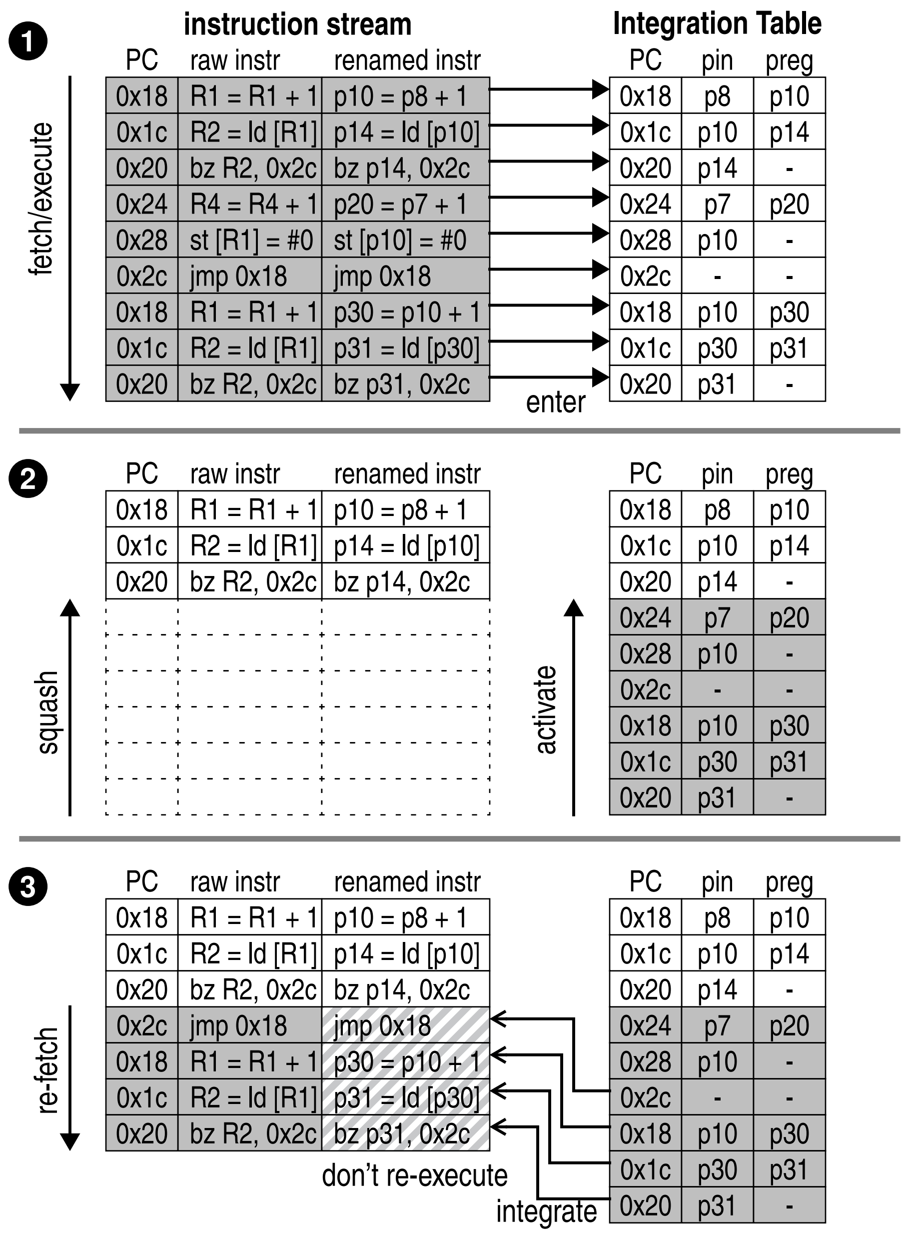


Squashing is wasteful because many squashed instructions are both control- and data- independent of the mis-speculation itself, but must be re-executed anyway. Squash re-use saves the results of squashed instructions, indexes them using the IT, and subsequently allows those results to be integrated by the re-processed versions of those instructions.

Unlike pre-execution reuse, the PC and physical register invariants that allow register integration to recognize and implement the squash reuse scenario are naturally present. Squashed instructions naturally match their re-fetched instructions PC for PC and data-dependence for data-dependence. A natural analog of the map copy operation—which in pre-execution reuse synchronizes the external mappings of the integrated and integrating instructions and allows the initial integration to take place—is also present: a squash naturally restores the map table state to its pre mis-speculation state, preserving the mappings of all pre mis-speculation instructions. This implies that squashed instructions and their re-executed counterparts naturally have the same external mappings that are needed to seed integration's inductive process.

The squash reuse scenario is illustrated in Figure 4.5. Squash reuse takes place in three phases. Initially, the program executes along a mis-speculated path and creates IT entries for every result it computes (marker #1). Then, when a squash happens, the squashed results are activated for potential integration (marker #2). Finally, as the program retraces reconvergent portions of the path which it has squashed, any instructions whose physical register inputs are still

Figure 4.5 Register integration squash reuse scenario.



valid are integrated (marker #3).

Notice, even though *only* squashed instructions are eligible for integration—in-flight or retired master thread instructions are integration ineligible—we do not create IT entries during the recovery process. Rather, we speculatively create

IT entries for all master thread instructions and then make the appropriate results integration eligible on a squash. We do this for two reasons. First, creating IT entries during register renaming is already implemented for pre-execution reuse. Second, recovery is usually monolithic and implemented as restoration from a checkpoint. There may not be time during recovery to create IT entries for many squashed instructions.

As hinted at in Chapter 2, squash reuse provides several benefits for pre-execution reuse. First, squash reuse provides a DDMT-independent selling point for register integration. The adoption of register integration for any reason other than DDMT will ease the adoption of DDMT itself, since the integration structures comprise the bulk of the additional hardware required to implement DDMT. From a performance aspect, the implementation of squash reuse actually solves a big dilemma for pre-execution reuse. One problem with integration is that it occurs during register renaming, when it is not yet known whether the integrating instruction will be retired, squashed, or squashed, re-fetched, and then retired. Without squash reuse, it becomes quite important (but quite difficult) to distinguish the first and third cases, since in the first integration will ultimately help performance and, in the second, it will not. The presence of integration-based squash reuse eliminates the need to make this distinction. A DDT instruction result can be integrated, the integrating instruction can be squashed, and the original result can be re-integrated via squash reuse.

4.4.2 Uses of Register Integration in DDMT

Register integration has several performance advantages in both squash reuse and pre-execution reuse capacities. In addition to saving the re-execution of integrating instructions, register integration also breaks the dataflow limit in some sense by collapsing data-dependences—a chain of dependent instructions cannot be executed in a single cycle but it can be integrated and reused in a single cycle! In addition to reducing the penalty of branch mis-prediction indirectly—by reusing squashed work—integration may also reduce the penalty directly. An integrating branch instruction that itself was mis-predicted may be resolved as early as the register renaming stage, rather than the conventional execution stage. Unlike other forms of result reuse, integration achieves this without reading from or writing to the physical registers themselves.

Register integration has even more benefits within the DDMT context. Integration allows DDTs to perform entire computations *on behalf* of the main thread. Without integration, a DDT can impact performance only by prefetching data (communicating pre-executed branch outcomes is also possible, but this requires an additional mechanism). DDT-to-DDT integration—which is implemented naturally by the integration circuit—allows DDTs to seamlessly overlap, letting younger DDTs to leverage work already performed in older DDTs and enhancing the performance of important pre-execution idioms like induction unrolling. Finally, register integration—actually, the failure of register integration—can also be used as the basis for a DDT suppression mechanism. A DDT needs to be suppressed (prevented from forking) if its dynamic instances do more harm than good—i.e., if they pre-execute problem instructions that are subsequently not executed by the master thread, or that are executed by the master thread before they are pre-executed by the DDT. The failure to integrate signals both of these conditions. Integration-based throttling is especially useful for DDTs that pre-execute problem branches as failure to integrate a pre-executed branch outcome effectively eliminates the benefit of the corresponding DDT.

As described in Section 4.2.3, DDT fork suppression can be implemented via the Trigger Table. The decision to suppress can be made based on the ratio—or some approximation thereof—of DDT target problem instances integrated to DDTs forked. A DDT can be suppressed if this ratio falls below a certain level. Since a suppressed DDT does not provide any negative suppression feedback—its ratio will never increase—suppressed DDT are occasionally and randomly "unsuppressed." We do not investigate DDT fork suppression in this dissertation.

4.4.3 A Physical Register Discipline for Integration

A centralized physical register file requires a *discipline* for managing—i.e. allocating and freeing—the physical registers. A centralized physical register file that supports register integration requires a more complex discipline that supports physical register sharing. For example, it would be incorrect for an instruction to free a physical register while another instruction is using it.

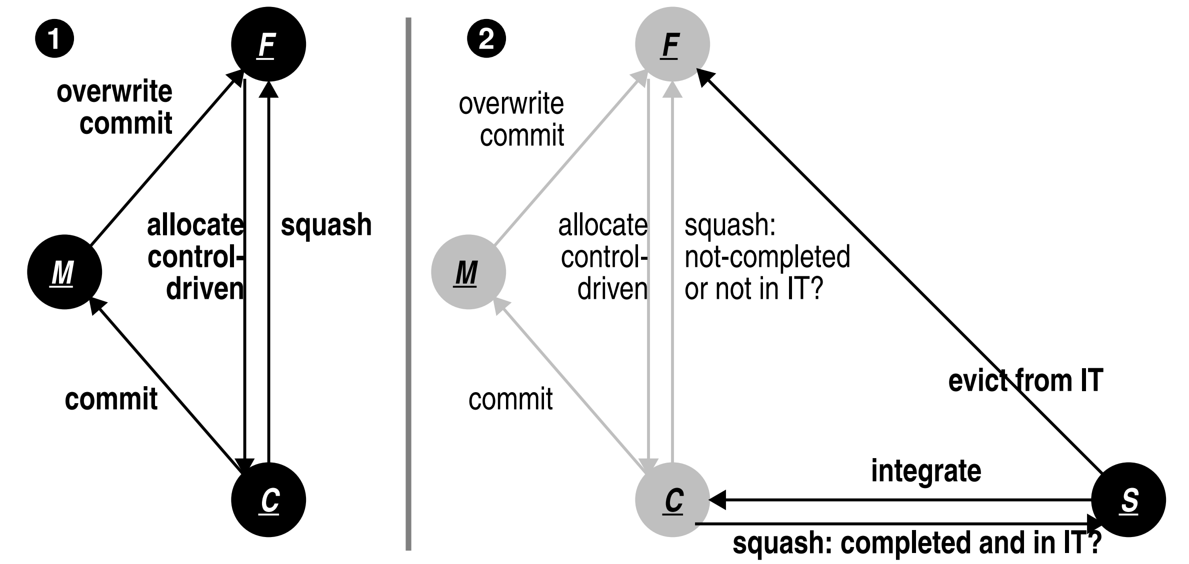
Our physical register discipline for register integration is based on *state transitions* rather than reference counts. We prefer a state based approach because, unlike a reference counting approach, it allows us to control the directions in which sharing (integration) takes place. Our implementation allows integration to occur in three directions—(1) a squashed result may be integrated by a control-driven (master) thread instruction, (2) a data-driven result may be integrated by a control-driven (master) thread instruction, and (3) a data-driven result may be integrated by another data-driven thread instruction. These three integration scenarios are called *squash reuse*, *pre-execution reuse* and *unrolling reuse*, respectively. We *do not* allow data-driven threads to integrate either squashed results or results from the master thread (the reason for this will be explained shortly and has to do with DDT suppression).

Our discipline is both *evolutionary* and *graceful*. It is evolutionary in the sense that it can be viewed as an extension to the conventional discipline. It is graceful in that any resource shortfall simply causes a collapse to the conventional case and produces correct behavior, albeit at lower performance.

In our framework, a physical register can exist in one of six states: *<u>F</u>ree*, *co<u>M</u>mitted*, *<u>C</u>ontrol-driven*, *<u>S</u>quashed*, *<u>D</u>ata-driven* and *<u>I</u>nvalid*. The *<u>M</u>* state is fictitious—it plays no part in integration or physical register management. We include it because it aids in understanding the framework. The state of a physical register is noted only in its IT entry. If a physical register does not have an IT entry—and it is not necessary that every physical register does—its state is implied by its presence in (*<u>F</u>*) or its absence from (*<u>C</u>* and *<u>M</u>*) the free list.

A conventional superscalar processor that does not implement DDTs or register integration implicitly implements a two state discipline in which a physical register is either free (*<u>F</u>*) or allocated. In our discipline, we distinguish two allo-

Figure 4.6 Physical register disciplines: conventional and with squash reuse.



cated sub-cases. The physical register is in the <u>*C*</u> state if it is allocated to the result of an in-flight instruction. It is in the <u>*M*</u> state if it is allocated to the result of a retired (committed) instruction.

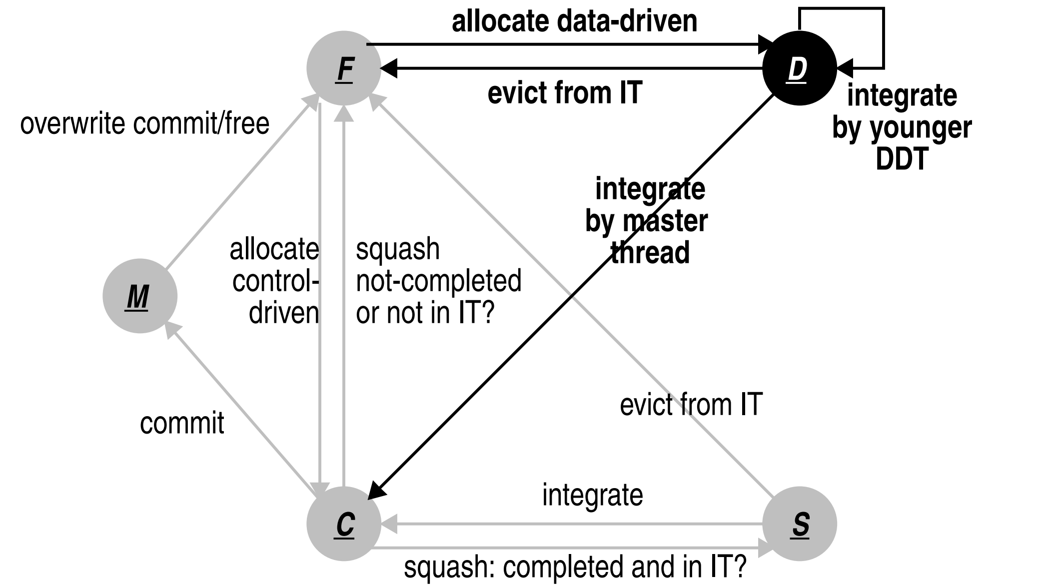
The left side of Figure 4.6 (marker #1) shows these states and the transitions among them. Physical registers in the <u>*F*</u> state are allocated to the <u>*C*</u> state. They transition to the <u>*M*</u> state when the corresponding allocating instruction retires. They transition from the <u>*M*</u> state to the <u>*F*</u> state when the instruction overwriting the architectural-to-physical mapping retires. During mis-speculation recovery, registers allocated to squashed instructions transition from the <u>*C*</u> to the <u>*F*</u> state.

A conventional processor implements its discipline implicitly: the state of a register is implied by its mappings in the map table or its presence in the free list. With the addition of integration and the IT, register states must be made *explicit*. Recall, IT entries are created during register renaming. This means that physical registers in any state can have IT entries. Since we want to implement register integration in certain directions but not in others and cannot infer the state of a register simply by its presence in the IT, states must be made explicit.

The right side of Figure 4.6 (marker #2) shows the $\underline{S}$ state and the added transitions that implement squash reuse. Rather than transition into the $\underline{F}$ state on mis-speculation recovery, registers allocated to squashed instructions *remain allocated* and transition into the $\underline{S}$ state. A physical register that is integrated transitions from the $\underline{S}$ state back to the $\underline{C}$ state. A physical register result can be squashed and re-integrated multiple times, transitioning back and forth between the $\underline{C}$ and $\underline{S}$ states.

These two transitions are not the only changes made to the discipline. An $\underline{S}$ state physical register does not have a free list or map table entry; the only record of its existence is its IT entry as it waits to be integrated. An $\underline{S}$ state register *without* an IT entry is “orphaned” or “leaked”—it will never be either freed or integrated. We can never allow this to happen. Consequently, whenever an $\underline{S}$ state register is evicted from the IT—because of an organizational set (PC) conflict—it must immediately transition into the $\underline{F}$ state and be added to the free list. Similarly, whenever a $\underline{C}$ state register that does not have an IT entry—again due to an organizational conflict—is squashed, it too must transition to the $\underline{F}$ state and be freed. The decision of whether to transition from $\underline{C}$ to either $\underline{F}$ or $\underline{S}$ on a squash is governed by another policy in addition to IT residence. Specifically, we perform the $\underline{C}$ to $\underline{S}$ transition only if the physical register has an IT entry *and* the instruction has issued—completed or in mid execution. Physical registers allo-

Figure 4.7 Physical register discipline for pre-execution reuse.



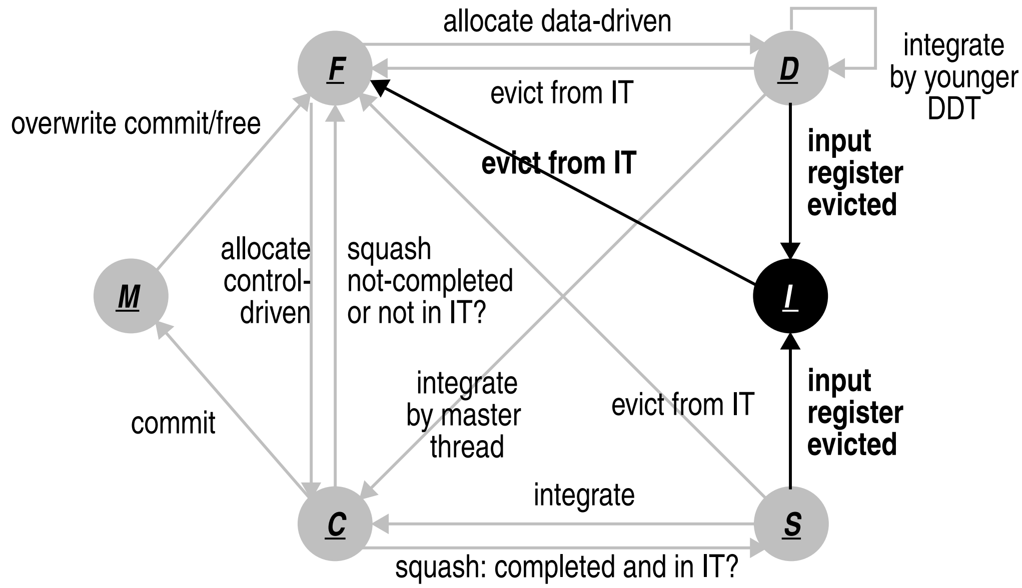
cated to un-issued instructions that are still sitting in reservation stations—regardless of whether they have IT entries or not—transition into the <u>*F*</u> state on mis-speculation recovery. The reasoning behind this policy is that it is the integration of completed instructions that contributes most to performance. Integration provides two main performance benefits: it allows integrating instructions to bypass the execution engine and it collapses dependent chains of completed integrating instructions. Neither of these benefits applies to non-issued integrating instructions.

Figure 4.7 shows the <u>*D*</u> state and the added transitions required to implement pre-execution reuse. Physical registers are allocated into the <u>*D*</u> state by DDTs. From this standpoint, transitions *to* the <u>*D*</u> state are similar to those to the <u>*C*</u> state. The transitions defined *from* the <u>*D*</u> state are similar to those defined from the $\underline{S}$ state. Like an $\underline{S}$ state register, the only record of a $\underline{D}$ state register is its IT entry. To avoid "leaking," a $\underline{D}$ state register must transition to the $\underline{F}$ state on IT eviction. The integration of a $\underline{D}$ state register by either a the master thread or a younger DDT is implemented by the transition from $\underline{D}$ to $\underline{C}$ and the (null) transition from $\underline{D}$ to itself, respectively

Integration from $\underline{C}$ to $\underline{D}$ and $\underline{S}$ to $\underline{D}$ is not implemented. Although integration in these directions does have consistent semantics, we preclude it to aid DDT suppression. Consider an integration from the $\underline{C}$ state to the $\underline{D}$ state. Such an integration means that the master thread has logically passed the DDT. If this condition is permanent, then executing the DDT will be ineffectual and we are better off providing the machine with feedback that will suppress the DDT in the future. If the condition is temporary, then the DDT was poorly chosen. Similar considerations apply for integration from the $\underline{S}$ state to the $\underline{D}$ state. Such an integration implies that the main thread passed the DDT and was subsequently squashed.

A final issue regards incremental invalidation and freeing of $\underline{S}$ and $\underline{D}$ registers once they become *unintegrable*. When a physical register is freed, all $\underline{S}$ and $\underline{D}$ physical registers that depend on it become *unintegrable*. To be integrated, a physical register's inputs must match entries from the map table and these may only be in the $\underline{C}$, $\underline{D}$ or $\underline{M}$ states. However, should the freed physical register be immediately reallocated to the same logical mapping by another instruction—a rare, but not impossible case—its dependent registers will falsely appear to regain their integration eligibility! To prevent this from happening, whenever a

Figure 4.8 Physical register discipline with cascaded invalidation.



register is freed, its dependent _S_ and _D_ state registers—which are found by associatively matching the *pin#* fields of the IT—transition to the _I_ state. The _I_ state is an intermediary to the _F_ state and allows this sort of freeing to cascade incrementally rather than to explode. The _I_ state and its transitions are shown in Figure 4.8.

An incremental invalidation scheme is very inefficient, requiring many parallel associative lookups simply to detect and avoid a single, rare corner case. Fortunately, there is a general way of avoiding unduly complex solutions to rare problems—and that is to ignore the problem until retirement, detect it via re-execution, and recover from it via a squash. A proposal for such a general microarchitecture style is the *dynamic verification architecture (DIVA)* [6, 15]. In DIVA, a high performance core feeds a retirement-candidate instruction stream to a slower, but high bandwidth verification engine which verifies via re-execution the outcomes of individual instructions—and individual instruction components—in parallel. The verification engine converts incorrect behavior by the performance engine into a performance hit. DIVA is a promising technology that may well be widely implemented and, if so, will relieve register integration of the need for incremental invalidation. In our experiments, we do in fact model DIVA, but a very restricted form of it which is used for a different purpose. We model incremental invalidations in the IT without modeling the associative matching bandwidth requirements.

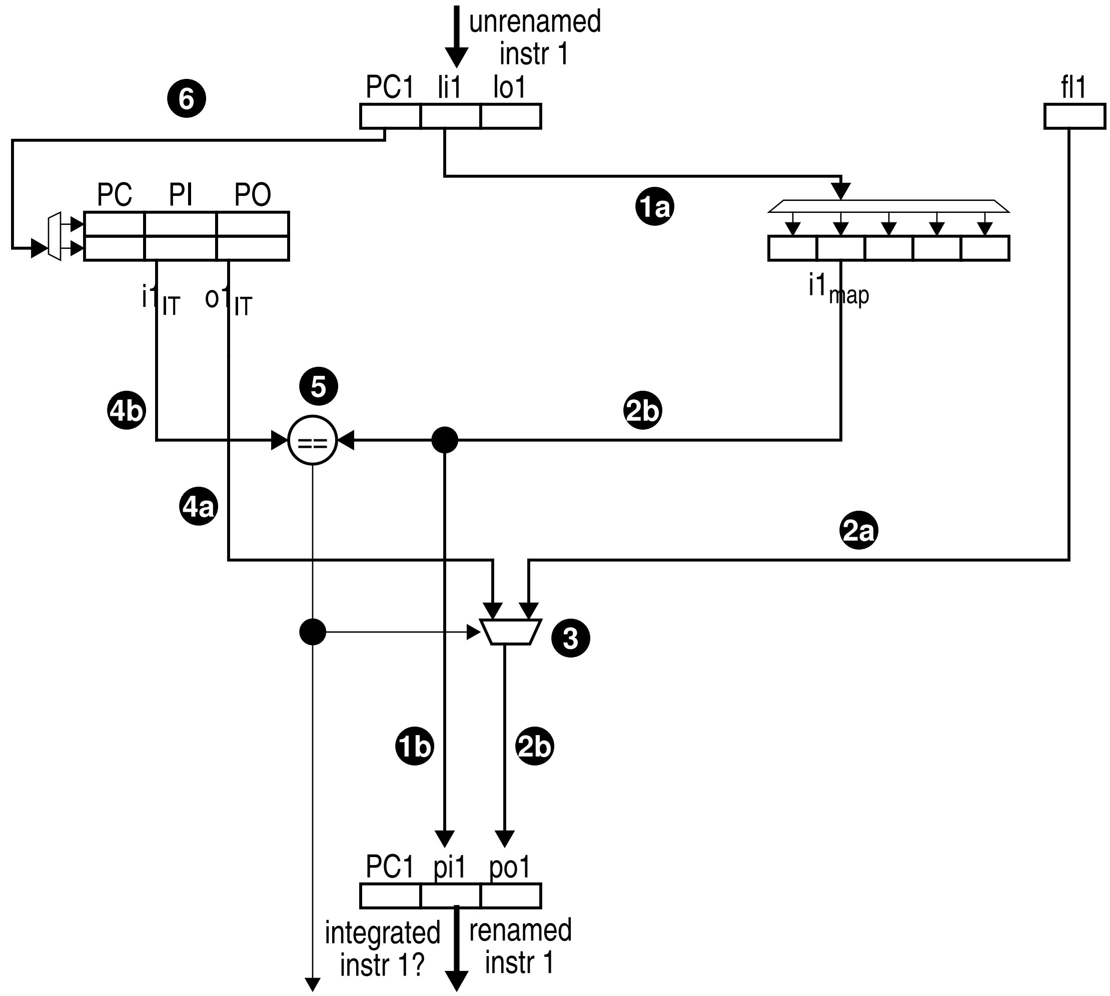
4.4.4 Integration Circuit

The most delicate piece of the integration mechanism is the integration circuit itself. The integration circuit examines each dynamic instruction and decides whether or not that instruction may be integrated. Of course, it must do so for multiple, potentially dependent instructions in parallel. We describe one possible implementation of this logic and its complexity. We begin with a scalar description of the circuit, then proceed to the superscalar case.

4.4.4.1 Scalar Circuit

A simplified scalar register integration circuit is shown in Figure 4.9. The figure makes two straightforward simplifications. First, each instruction has a single input register whereas a real instruction may have up to three input registers. Second, the IT is direct mapped—i.e., contains at most a single entry for each static instruction—whereas a real IT may be set associative and may

Figure 4.9 Scalar register integration circuit.



produce multiple matches per lookup. Once we have explained scalar register integration, we explain the effects of removing these simplifications.

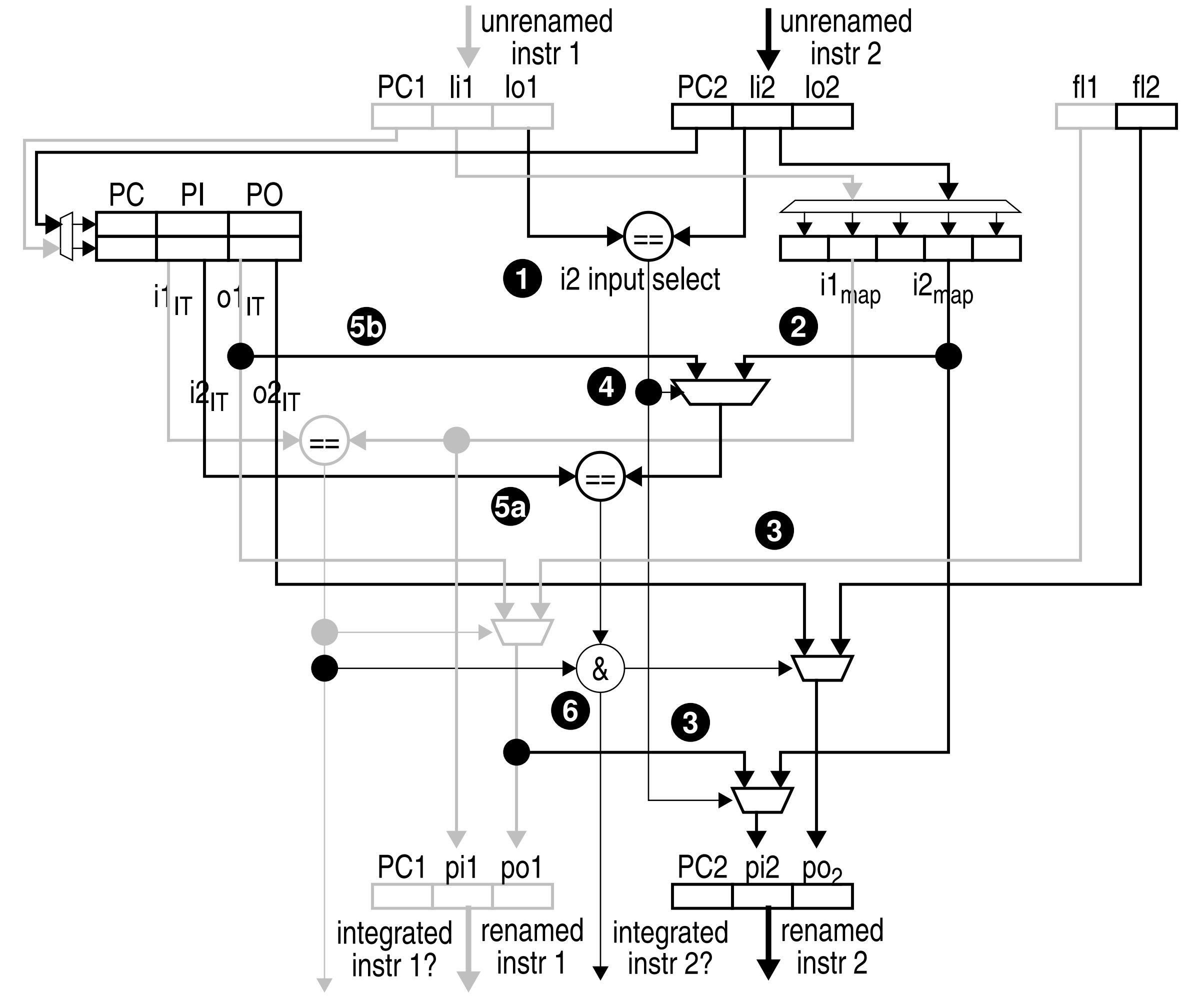
Conventional scalar register renaming occurs in two logical steps. In *input routing*, an instruction's logical input is renamed to a physical input using a map table lookup (markers #1a and #1b). In *output allocation*, its logical output is allocated a new physical register (markers #2a and #2b). This new logical-to-physical mapping is also entered into the map table, allowing future instructions that need the value to obtain their inputs from the correct location (not shown).

Integration adds a piece called *output selection* in which the output mapping must be chosen (marker #3) between a newly allocated physical register (marker #2a) and a physical register obtained from an IT entry (marker #4a). Output selection occurs logically after the input routing circuit since the integration test must compare (marker #5) the input physical register of the sequential instance (marker #1b) with that from the IT entry (marker #4b).

Output selection itself consists of two steps—IT lookup (marker #6) and the integration test (marker #5). One possible organization would perform both steps after input routing using the renamed input physical registers as indices into the IT. However, a more practical organization—and the one shown—splits the two, implementing IT lookup in parallel with input routing and the integration test after both. In this organization, the IT is indexed by PC only and the physical register numbers are used to match tags. This organization is preferred because it has a straightforward parallel-prefix superscalar formulation.

What are the effects of removing the simplifications? Having multiple input registers per instructions requires multiple parallel input routing instances and multiple pair-wise comparisons between input routed physical registers and the corresponding IT entry physical registers. The integration test (marker #5) succeeds if all pairs match. Having multiple IT matches requires multiple parallel copies of the matching circuit with the integration test choosing the IT entry that produces a full pair-wise match.

Figure 4.10 Superscalar register integration circuit.



4.4.4.2 Superscalar Circuit

Figure 4.10 shows a superscalar integration circuit. The superscalar changes are superimposed onto a grayed version of the scalar circuit from the previous subsection and Figure 4.9.

Let us review conventional superscalar renaming. Superscalar renaming is more complex than scalar renaming because its input routing decisions must reflect intra-group dependences. To do so, intra-group dependency-check logic (marker #1) acts in parallel with output allocation. This logic compares the logical input of each instruction in the group with the logical output of each previous in-group instruction; a match overrides the initial input routing retrieved from the map table (marker #2) and routes the input to the newly allocated physical register of the appropriate older intra-group instruction (marker #3). The logical dependence cross-check circuit compares the input of every instruction with the output of every previous intra-group instruction. In our example, since each instruction has a single input and each superscalar group contains two instructions, this results in a single comparison. In general, the number of comparisons needed grows as the square of the superscalar width, N—in particular as $I * N(N-1)/2$ where I is the number of inputs per instruction. As multiple matches may be occur and true dependences are only with the most recent match, the outcomes of the comparisons must be priority encoded. The depth of the longest priority encoder is N.

Integration requires that we implement *output selection* and any corrections it might imply for input routing for subsequent instructions. Input routing corrections take place as the result of the integration of a *chain of dependent instructions* in parallel. When integrating N instructions in parallel using an M-way set associative IT, the initial thinking is that every physical register input of every integration candidate must be compared with every physical register output of *every* older integration candidate. Comparing M inputs with M outputs requires $I*M^2$ comparisons. Comparing M inputs of each of N instructions with the M outputs of each older intra-group instruction requires $I*M^2*N^2$ comparisons. In fact, this was *our* thinking and our reported complexity in an early investigation of register integration [70].

What we did not realize at the time is that the results of the group dependence information computed on the instructions' logical registers (marker #4) can be used to streamline this logic. Comparing inputs with the outputs of *all* previous instructions is just a way of making sure that we perform the comparison with the outputs of the instruction the particular input *may* depend on. However, the group dependence information tells us for each input of each instruction *which if any* older in-group instruction that input depends on. Hence there is no need to compare the M inputs with the M outputs of *every* older instruction as we already know the identity of the instruction with whose outputs they must be compared. We compare the physical register inputs (marker #5a) of the M IT entries to the physical register outputs (marker #5b) of the M IT entries of *only* that instruction. Whether or not the register has an in-group dependence, we still compare the physical register inputs of the M candidates to the physical register obtained from the map table entry as integration of the older instruction may have failed. Finally, just like in conventional register renaming, a priority encoding circuit is required to cascade individual integration decisions (marker #6). The use of the group dependence information to simplify the output selection circuit reduces its comparison complexity from $O(I*N^2*M^2)$ to $O(I*N*M^2)$. The precise formula for total number of physical register comparisons is $(1 + ((N-1) * M * (M+1)) * I$. The depth of the longest priority encoder in the circuit is still N.

Output selection complexity is lower than that of the conventional logical dependence cross-check for low-associativity ITs, but diverges for higher-associativity implementations. For instance, a four-wide machine with a direct-mapped IT requires 14 physical register comparisons to implement output selection. The same machine with a 2-way IT needs 38 comparisons. Just for scale, an 8-wide machine with a 4-way IT—the one we model in our simulations—requires 282 comparisons! Certainly, a highly associative integration circuit is challenging to build. We should mention here that some of the complexity of the integration circuit may be incorporated into the structure of the IT itself. For instance, instead of simply producing *M* integration candidates for *every* instruction, the IT could produce *M* chains of integration candidates for *all instructions collectively*. This would reduce the complexity of the output selection circuit to $I*N*M$. However, it would require a novel IT organization with which we are not familiar.

4.4.5 A Shortcoming of the Current Formulation and an Alternative

Our current IT formulation and physical register framework has a serious shortcoming. This shortcoming is the result of a destructive interaction between the two functions of the IT. Primarily, the IT is an index on the physical register file that allows integration logic to implement integration. However, the IT also acts as an implicit tracking mechanism for squashed and DDT results.

In Section 4.4.3 we stated that any register in the $\underline{S}$ or $\underline{D}$ state that is evicted from the IT, must be immediately freed lest it be orphaned (or leaked) and lost forever. When a physical register transitions into the $\underline{F}$ state, the IT is associatively searched and all entries corresponding to instructions which read that physical register are invalidated—i.e., transition to the *I* state. Invalidations are then incrementally propagated. The net effect is that when a DDT physical register (a *D* state physical register) is evicted from the IT, a chain of events is set in motion that incrementally aborts all downstream DDT computation. In some sense this is acceptable, as the downstream portion of the DDT will not be integrated anyway. What often happens, however, is that the downstream portion of the DDT is destroyed before it has had a chance to pre-execute and prefetch. In essence, the IT's finite associativity not only prevents certain DDTs from being integrated, it prevents them from even being pre-executed for their prefetching effects.

Evictions of DDT physical registers are quite common. To simplify the integration logic, the DDT is organized as a low-associativity structure indexed by PC. This means that simple PC conflicts can cause evictions. The IT replacement policy can help with this. For instance, we can always evict an *F*, *M*, *C*, *S* or *I* state register if one is present (this is what we do). Or, we can choose not to create a *C* state entry if only *D* state registers are present in the set (we do this too). However, there is one case in which an eviction of a *D* state is unavoidable and that is if the unrolling degree of a DDT exceeds the IT associativity. If this is the case, then at least one of the induction instances—they all have the same PC—will not fit in the set, causing the rest of the DDT to be aborted. Consequently, the associativity of the IT is in a very real sense the limiter on the practical degree of unrolling—and the latency tolerance—in a DDT.

This limitation is *not* a fundamental property of either pre-execution, DDMT, or register integration. Rather, it is an artifact of our particular implementation. We have developed a second formulation in which the register tracking function is moved to a second table which explicitly tracks and maintains the state of every physical register. In this new formulation, IT eviction does *not* result in instantaneous freeing of DDT physical registers. As foreshadowed in Section 4.4.3, this formulation also uses re-execution to avoid incremental invalidations. We have only a preliminary implementation of this newer model, and hence will not use it in our performance evaluation. However, we can roughly estimate—probably slightly over-estimate—its performance by using the formulation described here with a fully associative IT.

4.4.6 Register Integration and Correctness

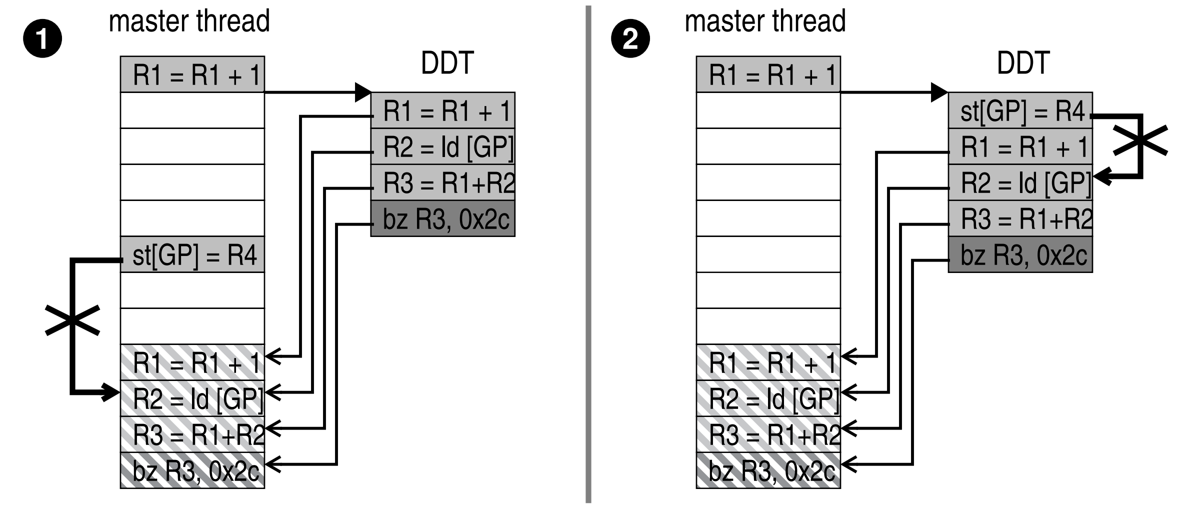
DDTs execute *speculatively*. Since they do not overwrite master thread physical registers and do not write stores to the data cache, incorrect execution within a DDT does not result incorrect behavior by the master thread. That changes once a pre-executed result is integrated. An integrated result is identical to a conventional control-driven result for all intents and purposes. This is good because it means that integrating instructions require no special handling. However, it also means that integrated results—and consequently the DDT instructions that produced them—are bound by correctness obligations.

An integrating instruction can be thought of as having two executions. In the *physical execution*, the operation actually takes place but is not architecturally recorded, either because the instruction is pre-executed in a DDT or because it has been squashed. In the *architectural execution*, the operation is supposed to take place, but doesn't actually do so. For most types of instructions, the basic integration test—the combination of operation and valid input values, denoted by PC and physical registers respectively—guarantees that the results of the physical execution are identical to those that would be produced in the architectural execution, allowing the former to be substituted for the latter.

Loads are the exception. Because physical register numbers are not sufficient to detect load/store collisions, the integration of a particular load is not guaranteed to be safe. As shown in Figure 4.11, one of two things can happen. A conflicting store may have executed between the load's physical and architectural executions (marker #1). Conversely, a store that forwards a value to a load's physical execution may not exist in its architectural execution context (marker #2). Collectively, these occurrences are called *mis-integrations*. Obviously, mis-integrated loads must not be allowed to retire. Stores cannot be mis-integrated as they have only register inputs.

In practice, we do *not* distinguish between the two mis-integration scenarios in the figure. However, we do make a different distinction based on the *temporal* relationship of the store (or lack of store) and the integration event itself. We call a load that is blindly integrated despite a store conflict *pre-mis-integrated* and a load that experiences a post-integration conflict *post-mis-integrated*. Pre-mis-integration and post-mis-integration are actually quite different. There is a natural way to detect and recover from post-mis-integrations due to the short duration

Figure 4.11 Mis-integration scenarios.



in which a post-mis-integration causing store can execute. Pre-mis-integration is a more difficult problem, largely because of the arbitrarily long stretch over which a pre-mis-integration causing store can occur. Explicit steps must be taken to either detect it post-integration or avoid the integration altogether.

In addition to detecting store conflicts from the master thread itself, integrated loads must also detect conflicting stores from other threads.

4.4.6.1 Detecting Mis-Integrations

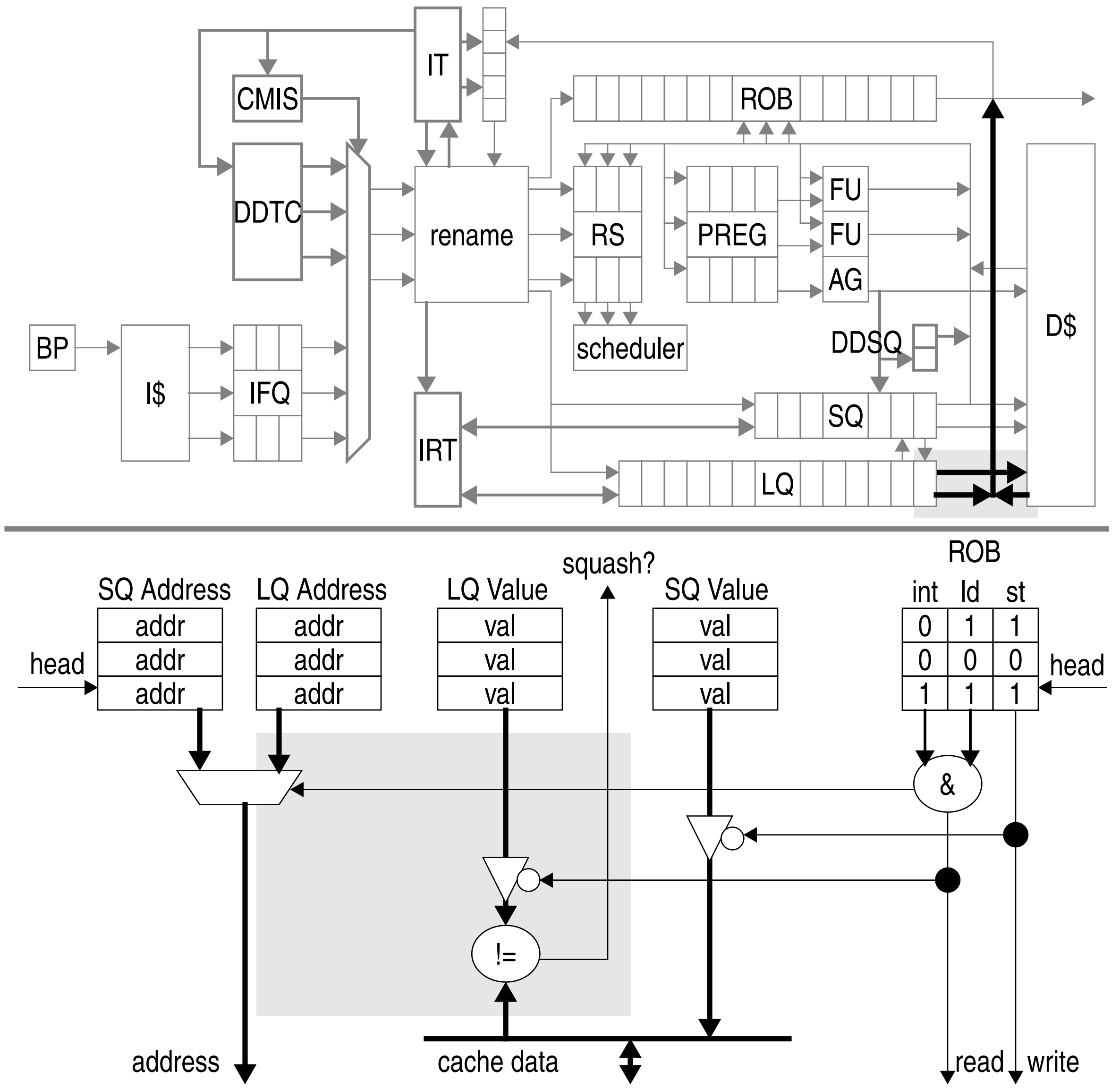
There are two ways to ensure that mis-integrated loads not retired. The first option is to try and *prevent* mis-integrated loads from being integrated in the first place by keeping integration-eligible loads *coherent* with respect to stores from the master thread. In this approach, we maintain address and value information with load IT entries and use master thread stores to "snoop" the IT and invalidate the appropriate loads. A second possibility is to *re-execute* integrated loads and initiate a squash if the re-executed value differs from the pre-executed value. Re-execution is simpler to implement and to explain but may result in many expensive squashes. Snooping reduces squashing but requires a significant amount of associative matching hardware. Also, a good snooping implementation that avoids false invalidations has several nasty corner cases, including the one shown on the right of Figure 4.11 (marker #2). We have experimented with snooping in the past [70, 72, 73], but currently model (and prefer) re-execution.

For re-execution we use a very restricted variant of DIVA [6, 15] in which only the cache access portion of integrated loads is re-executed. A schematic of our re-execution mechanism is shown in Figure 4.12. Re-execution takes place immediately prior to retirement and shares the ports used by stores for data cache writes. Paths are added from the head of the LQ—similar to those that exist from the head of the SQ—to provide the re-execution address the value for comparison. Re-execution latency is equal to cache access latency.

An alternative to DIVA style re-execution is to allocate RS entries to integrated loads and re-execute them using the usual machinery. DIVA-style re-execution requires no changes to the out-of-order core, triggers fewer re-executions and produces no false squashes. The only apparent advantage of scheduler-based re-execution is that it may detect mis-integrations earlier, reducing squash latency.

The performance cost of re-execution has three components—bandwidth requirement, re-execution latency, and squashes due to pre-mis-integration. The

Figure 4.12 Detecting load mis-integrations via re-execution.



bandwidth cost of re-execution is not great. Even with a very aggressive DDMT system, only a small percentage of all retired instructions are integrating loads (empirically, less than 2%). Assuming 10% of retired instructions are stores, re-execution demands a 20% increase in cache bandwidth. However, this additional bandwidth requirement is actually "virtual" in the sense that in the absence of register integration, the master thread would have to re-execute all pre-executed loads anyway. Finally, because integrating loads are not re-executed until they are ready to be retired by the master thread, DDT loads that integrate results from other DDTs are *not* re-executed. Such re-execution is meaningless because DDT loads do not obey sequential semantics with respect to stores within other DDTs. Its effect is that even loads that are integrated multiple times—e.g., during induction unrolling—are re-executed only once.

Re-execution latency is also minimal. A re-executed load will almost certainly hit in the highest level of cache. Nor does re-execution subvert the dataflow graph compression features of integration as re-execution is overlapped with the integration of downstream computation.

In practice, the most significant cost component of re-execution is the increased number of squashes due to mis-integration. We dedicate the next subsection to this particular aspect.

4.4.6.2 Avoiding Mis-Integrations

Mis-integrations are not extremely common, but they do have a noticeable incidence in some programs. While not impacting correctness, mis-integrations degrade performance by introducing squashes. From a performance standpoint, it is better not to integrate a load in the first place than to incur the cost of a subsequent mis-integration. Fortunately, there is a very easy way of doing exactly that even without snooping the IT.

Mis-integrations are a function of *program structure*. A mis-integration occurs when the program does not follow the precise path implicitly followed by a DDT and either includes a relevant store which the DDT omitted or omits a store the DDT included. Program structure is also the reason that mis-integrations are relatively infrequent. DDTs are generated to capture the common computation. If a load habitually reads the value written by an older store, then that store is included in the DDT.

Because mis-integrations are associated with differences between paths followed by the master thread and those followed by DDTs, and because DDTs follow the same dynamic paths repeatedly, we conjecture that it is dynamic loads that lie along certain program paths that cause the majority of mis-integrations. Empirically, this conjecture turns out to be correct—instances of static loads that occur along certain paths repeatedly cause mis-integrations. We use this repetitive behavior as the basis for a *predictor* for suppressing the integration of likely-to-be mis-integrated loads.

The *load integration suppression predictor (LISP)* is shown schematically in Figure 4.4. The LISP is a small cache of saturating up-down counters (SUD), indexed by a combination of load PC and some path information bits. A mis-integration creates an entry in the LISP or increments the SUD of an existing entry. A correct integration decrements the SUD of the corresponding entry, if one exists. The LISP is accessed during register integration. A load that matches an LISP entry with an above-threshold SUD value is not integrated.

4.4.6.3 Memory Model Issues

DDMT—and pre-execution in general—is a technique for accelerating a sequential program. However, there is no reason why DDMT cannot be used to accelerate multiple threads in a multithreaded workload or within a parallel program. In light of load integration, an issue that must be dealt with in these scenarios is the *memory model* supported by DDMT—i.e., the order in which DDT loads observe external stores and the order in which DDT stores are observed by external loads.

There are four different interactions between a memory model and DDMT. First, the order in which DDT stores are observed by other threads is irrelevant as DDT stores are not written to the first-level data cache, much less exposed to the general memory system. Second, the master thread is responsible for exposing integrated stores to other threads, and does so in program order. Third, we do not care about the order in which DDT loads observe stores from other threads as correctness does not become an issue until a load is integrated.

The fourth interaction is the one we care about, this is the order in which integrated loads observe external stores. Note that a similar problem exists even in a single master thread case, where DDT loads must be forced to "observe" master thread stores in program order. This is accomplished by re-executing integrated loads immediately prior to retirement. We believe that this same re-execution mechanism can enable a DDMT-augmented master thread to support any memory model supported by the base system, by supplying integrated loads with corresponding program order instances.

Program-order re-execution has been proposed as a way of implementing memory consistency models for speculative microprocessors [40], but was dismissed in favor of load queue snooping for cache bandwidth consumption reasons. Recently, re-execution has been revisited in the context of microprocessors that employ value prediction [56]. An interesting parallel exists between DDMT and value prediction-driven speculative execution. In each case, the naive implementation attempts to reuse entire computations by only checking the validity of their external inputs, assuming that the data dependences preserve the integrity of the rest of the computation. In each case, loads break this assumption, requiring a more sophisticated verification scheme.

4.5 Chapter Summary

This chapter describes the DDMT microarchitecture we propose and simulate. This microarchitecture is a series of extensions to a conventional dynamically scheduled superscalar processor with a pointer-based style execution core. The centralized organization lends itself to the close inter-thread cooperation present in DDMT. The main novel components of the microarchitecture are the data-driven thread cache (DDTC), the data-driven store-queue (DDSQ) and register integration.

Chapter 5

Experimental Evaluation

In this chapter, we use simulation to evaluate the performance potential of DDMT. There are many factors that determine the performance of DDMT and their interactions are subtle and complex. There are four primary factor groups—DDT selection parameters, microarchitectural parameters specific to DDMT, system parameters that have nothing to do with DDMT directly, and cross-training effects having to do with real implementations of DDT selection. To explore the full-cross product of all of these factor groups is infeasible from a time and space standpoint. In fact, even fully exploring the interactions within a group is infeasible. Instead, we take the following approach. We choose a particular—not necessarily optimal—design point that combines reasonable parameters for every factor group. The only exception is that we choose to ignore the effects of real world DDT selection. We evaluate this design point thoroughly, and then use it as a common baseline for a collection of sensitivity studies.

The chapter is divided into three main sections. Section 5.1 describes our experimental framework. Section 5.2 examines a single DDMT system design point in detail. The central design point is explored in a *limit* DDMT scenario, in which the data set used to select DDTs is identical to the one used to measure their performance impact. The section contains an evaluation of register integration in squash reuse capacity, a study of problem instructions and the performance potential of "perfecting" them, a characterization of automatically selected DDTs, and a performance analysis of DDMT. The final section presents a brief sensitivity analysis on various aspects of DDMT, including microarchitecture aspects, DDT selection parameters, the relationship between DDMT's setup and runtime components, and the potential impact of DDMT on future system configurations. We present only a small set of results in detail and summarize the rest of our findings.

5.1 Experimental Framework and Methodology

This section describes the experimental framework used in our evaluation. We describe our performance simulation environment, the tool-chain used to select DDTs, our benchmark programs and a sampling methodology that allows us both to deal effectively with long simulation runs and large data sets and to model several DDT setup/runtime scenarios.

5.1.1 DDMT Performance Simulator

The DDMT performance model is a cycle-level simulator that uses a combination of execution-driven and trace-driven techniques. The simulator uses the program loader, instruction decoder, and system call modules included in the SimpleScalar [12] release. The timing model including the memory system was (incrementally) written from scratch. The simulator executes instructions in-order during the register renaming stage, allowing it to model perfect memory disambiguation, perfect branch resolution, and perfect memory latency. The execution timing model of the pipeline itself, however, is event-driven as is the memory system model. The simulator uses its early knowledge of the correct execution path only in the various statistics packages. It does not maintain a "golden" register and memory state associated with correct path instructions and "throw away" copy-on-write state for instructions along mis-speculated paths. The simulator models the physical register file and the register renaming process faithfully—it must in order to model register integration. It also models values and value bypassing in all speculative memory structures like the load and store queues and in the DDSQ. During execution, all results are written to speculative storage. Register values are not copied to a golden architectural register file, these are always accessed via the corresponding map table. Store values are written to the memory system on retirement. As specified in the microarchitecture, integrating instructions are *not* executed by the out-of-order engine, instead the value that was written into the physical register during the initial execution is used instead. Re-execution is also modeled faithfully. On retirement, integrated loads are re-executed and the value obtained from the cache is compared to the value in the physical register. This process discovers all mis-integrations.

Table 5.1 Performance simulator configuration: branch predictor and caches.

parameter		configuration
Brach Predictor	Conditional branches	Combined 16K entry bimodal and 16K entry 10-branch history gshare predictors with a 16K entry chooser.
	Taken targets	8K entry, 4-way set-associative BTB for direct jumps and calls. 8K entry, direct mapped, 12-bit, 3-target history BTB for indirect jumps and calls.
	Return addresses	64-entry, speculatively updated return address stack
Memory Hierarchy	L1 I-Cache	32KB, 2-way set associative, 32B line cache, with LRU replacement and an access latency of 1 cycle.
	ITLB	64 entry, 4-way set associative TLB with LRU replacement,and a 2 cycle access latency. Each entry maps a 4K page.
	L1 D-Cache	64KB, 2-way set associative, 32B line data write-back, write-allocate cache, with LRU replacement and an access latency of 2 cycles.
	DTLB	128 entry, 4-way set associative TLB with LRU replacement,and a 2 cycle access latency. Each entry maps a 4K page.
	L2 U-Cache	1MB, 4-way set-associative, 64B line write-back, write-allocate cache, with LRU replacement and an access latency of 6 cycles.
	L2 bus	32B bus operating at processor frequency
	Memory Bus	32B bus operating at 1/4 processor frequency
	Main Memory	Ideal main memory with 70 cycle access latency
	Write Buffer	16 entries
	Miss Handlers	64 outstanding misses

The performance simulator can execute and retire any instruction stream. The *correct* execution of the program is guaranteed via the SimpleScalar toolkit's external I/O (*eio*) utility. For each full benchmark run, a fast functional simulator creates an *eio* trace, which contains a record of all system calls performed during that run. The performance and DDT selection simulators use this trace to replay system calls rather than executing them from scratch each time. This utility enables perfectly repeatable executions. By comparing instruction count and register state at every system call, we can ensure that the architectural execution modeled by the performance simulator is, in fact, the correct one.

The configuration of the performance simulator is detailed in three tables.

Table 5.2 Performance simulator configuration: pipeline and scheduler.

parameter		configuration
Fetch	Width	8
	Branches per cycle	Fetch can proceed past a maximum of two taken branches per cycle. Cache block alignment constraints are not modeled.
	Queue	16 entries
	Latency	3 cycles
Decode	Width	8
	Latency	1 cycle
	Branch Fixup	Direct target mis-prediction is fixed at this point.
Rename	Width	8
	Latency	2 cycles
	Physical Registers	320 = 64 architectural + 128 in-flight master instructions + 128 DDMT
Retire	Total Width	8
	Store/Re-execute Width	4
Ordering Buffers	Reorder buffer	128 entries
	Load/Store queues	64 entry load queue, 32 entry store queue
Execution Resources	Cache/Store Queue Ports	4 fully pipelined cache/store queue read ports. 2 fully pipelined store queue write ports.
	Functional Units	8 simple integer execution units, 3 FP/integer multiply divide units. All operations take 1 cycle. Integer multiplication and FP addition/multiplication take 4 cycles. Integer/FP division takes 20 cycles. The multipliers and adders are fully pipelined. The divider is not pipelined.
Scheduler	Reservation stations	80 centralized entries
	Widths	A maximum of 8 instructions scheduled per cycle. A maximum of 4 memory operations scheduled per cycle, all of which may be loads but only two of which may be stores. A maximum of 4 FP operations scheduled per cycle. A maximum of 2 branches scheduled per cycle.
	Operation priorities	Ready operations are priority sorted and issue in this order according to the availability of functional resources. Loads, branches, and floating point operations are in the high-priority group, all other operations are in a lower-priority group. Instructions within a group are sorted by age.
	Load scheduler	Loads issue in the presence of older stores with unknown addresses. On completion, a store checks all younger issued loads. An address collision signals a load mis-speculation and triggers recovery. To recover from a load mis-speculation, the load and all younger instructions are squashed. The load scheduler learns from past mis-speculations using a 256-entry, fully associative collision history table (CHT).
	Register read latency	2 cycle
	Address generation latency	1 cycle
	Store-forward latency	2 cycle
	Scheduling around cache misses	Cache misses are sent to the next level of the memory hierarchy in series. The scheduler is informed when a miss returns and issues the load again. Primary miss is 10 cycles: 6 cycle L2 hit latency, plus at least 1 cycle on the L2 bus (if there are no queueing delays), plus 3 cycles from the time the load is re-issued.

Table 5.3 Performance simulator configuration: DDMT and register integration.

parameter		configuration
Register integration	Integration table	256-entry, fully-associative IT
	Load Integration Suppression Predictor	256-entry fully associative table of saturating up-down counters (SUD). A suppression threshold of 32 events. A squash increments the counter by 8. A retirement of an integrated load decrements the counter by 1. 4-bits of path information per tag.
DDMT	DDTC/Trigger Table	Infinite, oracle DDTC and Trigger Table
	DDT Fork Suppression	none
	Hardware contexts	4, for up to 3 active DDTs
	Rename Injection policy	8 instructions from every active DDT injected once every 8 cycles. Since the processor supports 3 active DDTs, up to 3 in every 8 cycles can be used to sequence DDT instructions.
	DDSQ	16-entry, 2 cycle access latency to match the cache and store queue

Table 5.1 shows the functional branch predictor and memory hierarchy parameters, Table 5.2 shows the pipeline configuration and Table 5.3 shows the default register integration and DDMT parameters.

The base configuration described in Tables 5.2 and 5.3 is that of an aggressive, 8-wide processor with large caches and branch predictor tables. The processor is similar in size and shape to the announced Alpha 21464 [26]—a processor with support for 4 concurrent threads. The sensitivity analysis section studies the impact of DDMT on narrower processors, processors with smaller caches and branch predictor tables, and processors with faster clocks (i.e., longer pipelines and relative memory latencies). These processors more closely resemble another current high performance processor, Intel's Pentium 4 [41].

Our central configuration employs a 256-entry *fully-associative* IT. As we described in Chapter 4, a highly-associative IT—or at least a highly-associative integration circuit—is impractical to build. However, as we also described in

Chapter 4, our current physical register discipline formulation causes destructive interference between the IT's two roles—integration broker and DDT physical register tracking mechanism. The result of this interference is an inability to pre-execute DDTs of high unrolling degrees, even if those DDTs target loads such that their main benefit is cache prefetching, not the integration of the loaded result. We have developed a new formulation of the physical register discipline that does not suffer from this problem. Since the limitation of the current formulation is artificial—i.e., it is not inherent to pre-execution or register integration—we would like to model this refined formulation in its place. However, we have only a preliminary simulator implementation of this newer formulation.

A full simulated implementation of the new formulation is important future work. In the meantime, our solution to this quandary is to model the new formulation using the current implementation with a fully associative IT. The fully associative IT only approximates the new formulation, and has positive and negative performance impacts compared to a faithful simulation. It over-estimates performance because a fully associative IT produces more successful integrations than one of lower associativity. It under-estimates performance because eviction of the "wrong" physical register still triggers an invalidation cascade that destroys the remaining portion of the corresponding DDT. At this point, we do not know the precise contribution of these two sources of error. However, since these errors are in opposite directions, the net performance error will be smaller than that produced by either factor alone. For completeness, we show results with low-associativity IT configurations with the caveat that these underestimate performance.

5.1.2 DDT Construction Tool-Chain

The DDT construction tool-chain consists of three tools that perform the following tasks: (1) problem instruction selection, (2) selection of single-PDI DDTs, and (3) the merging of single-PDI DDTs into multiple-PDI DDTs, respectively.

Pi-select is the tool that performs problem instruction (PI) selection. It is a simple statistical analyzer that takes raw PI data—instruction counts and dynamic branch misprediction and cache miss counts—and produces a list of PIs that meet the specified criteria. Raw PI data is one of the outputs of the performance simulator.

As described in Chapter 3, the second phase of DDT selection uses two tools. The first is *sim-ddt-build* which takes the list of PIs produced by *pi-select* and uses functional simulation to build the statistical database of all backward slices of all PDIs of these PIs. To allow it to recognize PDIs, *sim-ddt-build* simulates a functional cache hierarchy—not latencies, bandwidths, busses or MSHR's—and a branch predictor which are configured to match those of the performance simulator. The second tool is *ddt-select. Ddt-select* loads the statistical database, and out of all possible slices and sub-slices selects a set of single-PDI DDTs. *Ddt-select* is parametrizable to allow it to select DDTs using different threshold criteria or for processors with different latencies and bandwidths. The precise parameters and their default values are summarized in Table 5.4.

Ddt-merge, which merges the single-PDI DDTs produced by *ddt-select*, is the

Table 5.4 Ddt-select parameters.

	parameter	description	default
Selection Parameters	Maximum Scope	Search for DDTs in a contiguous window of a given size1starting from the problem instance.	1024
	Maximum DDT length	Limit an individual DDT to a certain length	32
	Maximum Unrolling Degree	Limit the unrolling degree (both induction unrolling and unoverlapped full unrolling) to a certain level in order to avoid set conflicts in the integration table.	4
	Latency Coverage Acceptability Factor (LCAF)	Accpet DDTs only if the single instance latency tolerance is within a certain factor of the target latency tolerance.	25%
	Master Thread Base IPC	Used in computing master thread sequencing schedule and in computing DDT overhead.	from simulation
Simulated Machine Parameters	Renaming Width	Used in computing master thread sequencing schedule and in computing DDT overhead	8
	L1 Hit Latency	Used in estimating execution times	3 cycles
	L2 Hit Latency	Used in estimating execution times	10 cycles
	Memory Latency	Used in estimating execution times	70 cycles

final tool in the chain. The merging process is not parametrizable.

5.1.3 Benchmark Programs

We evaluate DDMT using the SPEC CPU2000 Integer benchmark suite. We use all 12 benchmarks with a total of 16 training runs and 15 test runs. The benchmark *eon* has three testing and three training runs—*cook*, *kajiya*, and *rushmeier*—which we refer to as *eon.c*, *eon.k* and *eon.r* individually or *eon* collectively. The benchmark *vpr* has two test and two training runs—*place* and *route*—which we refer to as *vpr.p* and *vpr.r* individually or *vpr* collectively. The benchmark *perlbmk* has two training runs—*scrabbl* and *diffmail*—which we refer to as *perl.s* and *perl.d* individually and as *perl* collectively. The *perlbmk* test input set is not the original one that comes with the SPEC2000 release. The original test set includes a script that loops over other scripts and executes them using the perl shell command. However, while our microarchitectural simulator models a multithreaded processor, its operating system proxy interface does not model a multiprogramming OS and, as a result, is unable to perform the fork and exec system calls used to implement shell calls. To overcome this limitation, we create an alternative input data set by concatenating the individual scripts into a large monolithic script which we then execute directly.

Our evaluation also includes performance results for two micro-benchmarks—*em3d* and *mst*—from the pointer-intensive *Olden* suite [66]. We chose the two programs because we understand their behavior well. *Em3d* computes an iterative solution of an electromagnetic field equation over a bipartite mesh. *Mst* computes the minimum spanning tree of a random graph. Both micro-benchmarks use synthetic inputs. Each micro-benchmark has a single dominant computation loop and, hence, a single dominant DDT to pre-execute problem instructions within a loop iteration. *Em3d*'s loop is nested—the outer loop iterates over all nodes in the system, the inner loop iterates over the neighbor nodes of the current node. *Mst*'s loop iterates over the remaining unadded nodes and finds the one with the shortest edge to the one of the nodes already in the graph. Being pointer-intensive, both benchmarks have a large fraction of serial memory accesses that defy value-analytical address prediction but also present a latency tolerance challenge to pre-execution, which must rely on unrolling idioms to tolerate these latencies. The use of micro-benchmarks is illustrative of the performance potential of DDMT under "ideal program conditions". Micro-benchmarks also magnify DDMT's sensitivity to different parameters, allowing its various aspects to be more readily studied.

The benchmarks are compiled for the Alpha EV6 architecture using the Digital Unix V6.21 compiler with optimization levels and flags specified for producing PEAK executables. The machine used for compilation is an Alpha 21264 [49] which uses a clustered microarchitecture. At a high optimization level, the Digital Unix compiler inserts many nops into the executable to align basic blocks within cache lines, but more so to effect better-than-random cluster assignment. In a typical run, as many as 15% of all instructions are nops of one kind or another. Since the microarchitecture we simulate is not clustered, we have no use for these nops which do little but artificially inflate IPC values. To sidestep this problem, our simulation environment simply ignores nop instructions, skipping over them during fetch without consuming either fetch bandwidth or fetch queue slots. The result is that the dynamic cost of nops is completely removed from simulation. The static cost—the additional instruction footprint they occupy in the caches—remains. Even with this cost, the SPEC2000 benchmarks suffer almost no cache misses using a 32KB L1 instruction cache.

Table 5.5 summarizes the functional characteristics of the training runs of these benchmarks. As our simulation environment uses *sampling*—which we explain in the next sub-section—all numbers shown are measured over sampled runs. Total instructions are the number of instructions in the entire execution. All other measurements are as they appear in a sample. The two bottom row groups—in bold—show a characterization of the microarchitectural metrics rele-

Table 5.5 Functional benchmark characterization.

	em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Total instr's (M)	225	756	59900	26054	1721	8564	2498	8682	4291
Simulated instr's (M)	100.00	100.00	6000.00	2600.00	200.00	900.00	300.00	900.00	500.00
Loads (M)	39.16	28.74	1509.20	7314.86	49.95	226.44	72.85	218.49	116.15
Stores (M)	11.17	4.39	506.01	1377.24	38.26	191.70	62.95	92.38	62.44
Compiler prefetches (M)	5.16	0.00	0.25	0.00	3.02	16.31	5.31	2.15	3.12
Branches (M)	6.22	27.27	782.07	3145.51	23.27	105.17	35.56	146.21	90.55
FP pperations (M)	22.33	0.00	0.00	0.78	27.86	124.48	29.96	2.64	0.30
System calls	68	6	0	140	484	242	605	48	280
16K-entry branch predictor, 64KB 2-way set-associative DL1, 1MB 4-way set-associative L2									
Branch misp. rate (%)	**16.52**	**0.04**	**1.65**	**6.04**	**3.48**	**7.32**	**4.84**	**4.77**	**5.51**
Load L1 miss rate (%)	**30.02**	**41.05**	**2.71**	**0.79**	**0.13**	**0.09**	**0.13**	**1.76**	**2.18**
Load L2 miss rate (%)	**2.52**	**34.80**	**0.64**	**0.02**	**0.00**	**0.00**	**0.00**	**0.72**	**0.49**
2K-entry branch predictor, 8KB direct mapped DL1, 128 KB 4-way set-associative L2									
Branch misp. rate (%)	**16.52**	**1.37**	**1.72**	**9.97**	**6.21**	**9.73**	**7.17**	**7.11**	**9.74**
Load L1 miss rate (%)	**37.02**	**41.47**	**4.36**	**15.21**	**6.57**	**6.58**	**6.92**	**10.29**	**7.94**
Load L2 miss rate (%)	**21.40**	**38.55**	**1.59**	**0.25**	**0.00**	**0.01**	**0.01**	**0.93**	**1.09**

	gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Total instr's (M)	53454	8830	12445	34287	25555	12442	16722	2851	10238
Simulated instr's (M)	5400.00	900.00	1300.00	3500.00	2600.00	1300.00	1700.00	300.00	1100.00
Loads (M)	1081.16	246.03	295.70	971.91	778.86	291.56	458.21	79.27	327.17
Stores (M)	354.42	74.69	125.75	591.58	394.90	96.42	287.10	26.05	129.09
Compiler prefetches (M)	2.51	5.54	0.00	1.43	2.13	0.27	0.51	0.02	0.48
Branches (M)	662.62	185.10	224.22	519.09	389.31	180.25	300.16	35.04	129.31
FP pperations (M)	0.00	0.05	0.90	2.86	10.50	53.45	4.41	19.55	69.46
System calls	0	57	287	656	0	156	3	6	220
16K-entry branch predictor, 64KB 2-way set-associative DL1, 1MB 4-way set-associative L2									
Branch misp. rate (%)	**4.43**	**4.68**	**4.03**	**7.04**	**5.96**	**9.29**	**0.76**	**10.31**	**5.24**
Load L1 miss rate (%)	**3.25**	**36.67**	**4.87**	**0.74**	**0.11**	**8.79**	**0.73**	**4.09**	**4.40**
Load L2 miss rate (%)	**0.00**	**12.14**	**0.72**	**0.04**	**0.00**	**0.00**	**0.09**	**0.00**	**0.98**
2K-entry branch predictor, 8KB direct mapped DL1, 128 KB 4-way set-associative L2									
Branch misp. rate (%)	**6.28**	**5.26**	**5.40**	**11.45**	**9.35**	**10.93**	**2.74**	**11.27**	**5.76**
Load L1 miss rate (%)	**9.76**	**41.06**	**11.91**	**7.05**	**9.47**	**16.69**	**7.50**	**14.78**	**12.08**
Load L2 miss rate (%)	**0.90**	**30.07**	**2.36**	**0.31**	**0.00**	**5.00**	**0.39**	**2.17**	**3.21**

vant to pre-execution. The branch mis-prediction rate is computed by dividing the number of mis-predicted branches—a wrong target address counts as a mis-prediction—by the total number of branches. The load L1 miss rate is computed by dividing the number of loads that incurred L1 cache misses by the total number of loads. The load L2 miss rate is computed similarly. Rates are shown for two cache and branch predictor configurations. Our central configuration, used in the bulk of our experiments, uses large caches and a large branch predictor. On the bottom is a second configuration, with caches and a branch predictor tables that are a factor of 8 smaller than those used in the central configuration. This configuration is used in our sensitivity analysis. Miss and mis-prediction rates were obtained using functional simulation and do not include the effects of speculative execution. The top configuration is our central configuration. We do not show instruction cache and ITLB miss rates, as these are close to zero for all benchmarks. Only *mst* and *mcf* have significant DTLB miss rates.

5.1.4 Sampling

In order to practically deal with performance simulations of training input data sets which often runs for tens of billions of instructions, our simulation environment supports sampling. Both *sim-ddmt* and *sim-ddt-build* have three modes of operation—*active*, *warmup* and *fast-forward*. In active mode, *sim-ddmt* models performance at the cycle level while *sim-ddt-build* builds and analyzes DDTs. All statistics reported are the ones collected during active mode. The simulator operates in active mode for a pre-set number of instructions. In timing simulation, this means a pre-set number of committed main thread instructions. The two other modes are designed to accelerate simulations of full runs. In warmup mode, the simulator functionally executes a given number of instructions and updates the state of the branch predictor and caches. Warmup mode is used to ensure that active mode simulation does not suffer from cold-start memory or branch prediction artifacts. In fast-forward mode, the simulator functionally executes a given number of instructions, but does not update the branch predictor or caches. Fast-forward mode is used to speed up functional simulation between active mode samples.

The performance runs are sampled at an interval of 1B instructions. The sampling rate is 10%—each individual sample contains 100M instructions. Each sample is preceded by a 1%—or 10M instruction—warmup phase. A training run therefore cycles through the following three phases repeatedly—890M instructions are fast-forwarded, 10M are functionally executed to warm up the caches and branch predictor (we have found 10M instructions sufficient for our benchmarks and cache sizes), and 100M are simulated at the cycle level. The exception is the initial sample which fast forwards only 90M instructions. The overall effect is that a 100M instruction active sample is taken after the first 100M instructions have and every 1B instructions thereafter. Sampling is uniform. We do not sample only the “interesting”—translation “pre-execution-conducive”—parts of the program. It is possible that the 1B sampling period used is in phase with some periodic behavior taking place in the program, resulting in non-representative results. Our comparison with unsampled runs indicates that sampling can induce errors in both directions. For the SPEC2000 benchmarks, we have observed errors of +10% in runs of *mcf* and *vpr.route* and errors of -5% in runs of *gzip*. In all other benchmarks, sampling results in errors of no greater than 4% in either direction. Sampling produces larger errors in the microbenchmarks, -22% in *em3d* and -42% in *mst*. The dynamic instruction counts of the training runs of these benchmarks are too short to allow more than a single sample to be taken. As a result, some high IPC initialization code in both programs is not sampled. While the absolute error of sampling is quite low, we are more concerned with its *relative error*—the error introduced by sampling into our speedup calculations. To clarify our error definition, on an unsampled speedup of 10%, a +50% error implies that the measured sampled speedup is 15% while a -50% error implies that the measured sampled speedup is 5%. We have found that sampling introduces an error of as much as +138% in *mst*— a 93% speedup is measured on a sampled run while an unsampled run observes only a 39% speedup. All other measured speedup errors lie in the range of -68% to +53%. Most are in the range of -40% to +28%. Part of the reason for the speedup errors we observe is that our unsampled DDMT experiments pre-execute DDTs selected from a slice database that was itself constructed from a *sampled* run of the program.

Sampling also has another benefit. It allows us to create different relationships between the DDMT setup and runtime phases to model different idealized and more real world scenarios. Specifically, our evaluation uses three scenarios. Most of our data is presented in a *limit* scenario, in which DDT construction and performance measurements take place over the same sample. Although idealized, one can imagine that *limit* corresponds to a real world scenario in which DDTs are dynamically and continuously extracted by dedicated hardware or by a dedicated software thread. Two other scenarios we model using sampling are the *static* and *dynamic* scenarios. The static scenario models offline DDT construction. To model the *static* scenario we extract DDT using samples of the test—rather than the training—inputs of the programs. The *dynamic* scenario models online, VM-style DDT construction. To model this scenario, we construct DDTs using training input samples of the 10M instructions that execute immediately prior to each 100M performance sample. The *static* and *dynamic* scenarios are not used until Section 5.3.3, when their effects are explicitly measured.

5.2 Analysis of a Single DDMT Design Point

We begin our evaluation with a detailed analysis of a single point in the design space of DDMT. Everything about this particular design point is fixed—from the configuration of the base microarchitecture (shown in Table 5.1, Table 5.2, and Table 5.3), to the parameters of DDT selection, to the sizes, bandwidths and policies of the DDMT specific components, to the modeled relationship between DDT selection and the DDMT runtime system. The particular combination of DDMT-specific and DDMT-agnostic parameters was not chosen with the purpose of presenting DDMT in the best light. Although that would have been our preference, doing so means that all combinations of all parameters were simulated and the best one chosen. Instead, we chose a base machine configuration that we feel is representative of aggressive next generation processors, and what we feel are reasonable settings for the DDT selection process. Subsequent sections perform a sensitivity analysis, exploring the effects of variations in different parameters.

In the performance evaluation of any technique, it is useful to establish an *upper bound* for that performance. An upper bound serves three purposes. First, it allows us to measure the performance potential of the technique. Second, it lets us gauge the "quality" of our DDMT implementation—i.e., its ability to reach this potential. Finally, if the performance evaluation is performed via simulation, an upper bound acts as a safety check that can flag errors in the model.

There are three sequential implementation "stages" within DDMT—the PI definition, the DDT selection algorithm, and the DDMT runtime system. Ultimately, we only care about the performance of the third stage. However, we want to evaluate each of the three stages so that we can attribute performance—or performance shortfall—to the proper stage. Specifically, it is not the fault of the DDMT runtime system that a PDI latency went uncovered if the DDT selection algorithm could not select a DDT for that PDI. Similarly, it is not the fault of the DDT selection algorithm for not selecting a DDT for a given PDI if the corresponding static instruction was not identified as a PI by the chosen PI definition.

To perform such an evaluation, we use three different upper bounds. We evaluate our PI definition using an upper bound established by modeling a system with ideal caches and branch resolution. Next, we model ideal load and branch resolution latency only for PIs that fall within our chosen definition and use that as the upper bound for evaluating DDT selection. Finally, we model a system with ideal load and branch resolution latency only for those PIs for which DDTs were successfully selected. This serves as the upper bound for evaluating the DDMT run-time system.

While on the subject of upper bounds, we mention that our use of the *limit* scenario *in some sense* represents an upper bound on more realistic setup/runtime implementation combinations. However, due to interactions between the PI definition, DDT selection, and register integration this upper bound can be exceeded.

Before we begin with performance upper bounds, we must establish a performance baseline.

5.2.1 Establishing a Baseline

Register integration is an important part of DDMT. *Pre-execution reuse* enables fast resolution of pre-executed branches, while *unrolling reuse*—the integration of one DDT's result by another DDT—helps support induction unrolling. Register integration also implements *squash reuse* which has nothing directly to do with DDMT, although it interacts synergistically with pre-execution reuse, allowing pre-executed instructions to be integrated, squashed and re-integrated. This synergy makes squash reuse quite attractive to implement in conjunction with DDMT. Moreover, one has to go to some lengths to implement the other forms of reuse without implementing squash reuse as well. It follows that an implementation of DDMT and register integration will include an implementation of squash-reuse.

The presence of squash reuse presents an evaluation dilemma. On one hand, register integration is part of DDMT. On the other, we need to isolate the effects of DDMT which is, after all, the subject of this dissertation. Our solution is to evaluate register integration in squash reuse capacity here, and then use the enhanced system as the baseline for future experiments.

Our baseline configuration for evaluating squash reuse, called *base*, is an 8-wide superscalar processor without register integration. A second configuration, called *RI*, is the same 8-wide processor but with 320 physical registers and a 256-entry, fully-associative IT. Our reason for choosing 320 physical registers is the following. 64 physical registers are needed to hold the last committed value of every architectural instruction. In a superscalar processor with a 128 entry reorder buffer, an additional 128 registers are needed to hold the results of all speculative in-flight instructions. We add an additional 128 physical registers to hold squashed—and later DDT—values. Note, the number of physical registers and the number of IT entries need not be equal. Not every physical register needs an IT entry and the IT may contain empty entries. We use a smaller IT since, at any one point, the 64 *M* state (committed) registers do not need IT entries. This optimization is enabled by our use of a fully associative IT and the proper replacement policy. High degrees of associativity do not significantly help squash reuse as reuse completed instructions spanning several loop iterations is uncommon. The RI configuration also includes an aggressive 256-entry, path-sensitive load integration suppression predictor (LISP).

Table 5.6 summarizes the performance of squash reuse. We briefly explain

Table 5.6 Performance of register integration-based squash reuse.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Base	TIs fetched (M)	153.38	101.60	7395.88	4592.21	264.31	1491.95	438.14	1379.71	1061.32
	TIs renamed (M)	117.89	101.21	6943.86	3968.46	245.19	1321.58	396.87	1242.26	903.46
	TIs executed (M)	103.22	100.88	6360.80	3121.13	220.00	1084.09	340.50	1022.17	674.46
	BMR lat. (c)	3.86	404.99	12.96	10.28	10.68	10.07	10.10	21.35	14.68
	Load lat. (c)	21.17	54.74	5.66	3.10	3.01	3.01	3.01	4.68	3.85
	IPC	**1.28**	**0.29**	**3.91**	**3.51**	**3.46**	**2.88**	**3.32**	**1.94**	**2.37**
RI	TIs fetched (M)	153.38	101.60	7381.95	4527.98	263.22	1462.49	432.92	1371.57	1034.46
	TIs renamed (M)	117.89	101.21	6930.62	3894.45	244.14	1292.70	392.01	1233.65	877.51
	TIs executed (M)	103.04	100.48	6282.10	2840.59	213.68	1013.90	325.73	975.69	602.48
	BMR lat. (c)	3.86	404.81	12.78	9.41	10.51	9.50	9.76	21.07	13.86
	Load lat. (c)	21.17	54.73	5.64	2.84	2.96	2.87	2.92	4.59	3.56
	IPC	**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
	Speedup (%)	**0.00**	**0.03**	**0.17**	**1.38**	**0.49**	**2.03**	**1.15**	**0.44**	**2.13**
	TIs squashed (M)	1.59	0.81	239.95	457.42	16.74	149.43	32.99	103.98	141.71
	TIs integrated (M)	0.14	0.39	56.44	204.66	5.43	48.93	11.66	36.41	46.16
	Fetch reduction (%)	0.00	0.00	0.19	1.40	0.41	1.97	1.19	0.59	2.53
	Exec reduction (%)	0.18	0.40	1.24	8.99	2.87	6.47	4.34	4.55	10.67

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Base	TIs fetched (M)	9115.01	1937.01	2373.68	7699.03	5253.43	2966.89	1915.97	679.48	1922.02
	TIs renamed (M)	8108.63	1652.40	2053.02	6565.49	4545.75	2402.43	1850.32	558.03	1673.37
	TIs executed (M)	6812.99	1207.13	1634.99	4508.36	3161.04	1633.64	1772.42	407.22	1345.95
	BMR lat. (c)	18.13	54.27	20.63	12.50	14.91	12.13	12.54	11.54	35.03
	Load lat. (c)	3.41	21.76	4.28	3.68	3.01	3.81	3.23	3.37	4.44
	IPC	**2.70**	**0.62**	**1.81**	**2.55**	**2.73**	**2.13**	**4.52**	**2.58**	**1.66**
RI	TIs fetched (M)	8997.10	1909.59	2329.20	7526.94	5159.72	2847.94	1913.38	648.75	1907.63
	TIs renamed (M)	7993.90	1625.76	2012.33	6399.18	4454.68	2293.74	1847.96	529.65	1662.21
	TIs executed (M)	6052.98	1126.67	1498.72	4056.57	2904.26	1507.00	1744.24	341.40	1240.72
	BMR lat. (c)	17.25	53.94	20.12	11.81	14.25	11.19	12.37	10.34	34.82
	Load lat. (c)	3.10	21.82	4.04	3.45	2.83	3.62	3.20	2.86	4.20
	IPC	**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
	Speedup (%)	**1.56**	**0.87**	**0.94**	**2.03**	**1.76**	**3.63**	**0.19**	**4.77**	**1.32**
	TIs squashed (M)	1275.52	215.73	271.81	760.28	510.56	218.61	67.45	93.14	214.32
	TIs integrated (M)	553.33	44.79	98.70	288.89	183.91	83.68	24.69	47.76	83.23
	Fetch reduction (%)	1.29	1.42	1.87	2.24	1.78	4.01	0.14	4.52	0.75
	Exec reduction (%)	11.16	6.67	8.33	10.02	8.12	7.75	1.59	16.16	7.82

what our metrics measure. The raw counts of instructions fetched, renamed and executed are measured over *all* instructions in the program, not just correct path instructions (if only correct path instructions were measured here, these three counts would be equal). These metrics are shortened to *TIs* (total instructions) in the table and are given in millions of instructions (M). IPC is computed by dividing the number of instructions retired by the number of cycles simulated. The raw counts, average latencies and IPCs can be directly compared across configurations. Percent speedup is computed as $(IPC_{RI}-IPC_{BASE})/IPC_{BASE}$. We consistently use these conventions when displaying data.

We also present two average diagnostic latencies, the *average load execution latency* and the average *branch misprediction resolution (BMR) latency*. The average load latency is measured as the difference between the completion time and the issue time of all *retired* loads. The branch misprediction resolution latency is measured as the difference between the completion time and the register renaming time of all *retired* branches that were initially mis-predicted. The effects of wrong path instructions are not entered into average latency calculations as we do not care about the latency of wrong-path loads, whose late completion cannot stall the processor, or about the resolution latency of a wrong-path mis-predicted branches, which do not impact the correct path fetch schedule. In order to correctly model the effect of integrating instructions, we must modify these two definitions. The load latency of a single retired load and the resolution latency of a single retired mispredicted branch are defined as:

$$L_{LD} = MAX(0, T_{RESULT\text{-}complete} - MAX(T_{RESULT\text{-}issue}, T_{INSTR\text{-}rename})) \quad \text{EQ 17.}$$

$$L_{BR} = MAX(0, T_{RESULT\text{-}complete} - T_{INSTR\text{-}rename}) \quad \text{EQ 18.}$$

These definitions account for the fact that for an integrating instruction, execution may complete before the instruction is renamed. Under these definitions, a load integrating a completed result has an execution latency of zero. A mis-predicted branch integrating a completed result and performing instantaneous resolution also, appropriately, has a resolution latency of zero. Note, with the presence of register integration, it is possible for the average load latency to drop below the perfect load latency, which for our configuration is three cycles—1 cycle for address generation latency plus a 2 cycle cache/SQ hit. In fact, this happens routinely for benchmarks with few cache misses.

A second row group in the RI experiment tab presents squash reuse specific data. We list the total number of instructions squashed and total number of integrated instructions that were eventually retired. Both figures are listed in millions of instructions (M). Again, we are not concerned with instructions that have been integrated and subsequently squashed, these do not effect externally visible performance. Similarly, we do not want to double-count instructions that were integrated, squashed, and re-integrated. Consequently, latency and integration statistics are all collected during retirement. We also report the percent reduction in instructions fetched and executed, computed as (instructions$_{BASE}$-instructions$_{RI}$)/instructions$_{BASE}$.

Squash reuse with the basic integration configuration achieves a speedup of 4% for *twolf*. The rest of the programs experience a 1%-2% improvement in performance. Again, we ignore the fact that average performance degrades by about 2% when an extra cycle is added to the register renaming stage to account for the added latency of the integration process. Our rationale for doing so is that we are not concerned with these speedups per se, we are simply using the performance results as a baseline for another study.

The first order effect of squash reuse is to reduce the total number of instructions executed by the program. To that end it is successful, reducing consumption of execution bandwidth by up to 16% with an average reduction of about 7%. However, a reduction in instructions executed does not translate directly into speedup. Well designed processors have balanced pipelines, in which no one stage is the bottleneck. Any technique that relieves bandwidth demands of a given stage without lowering bandwidth demands by an equal amount on all prior stages will produce only negligible speedup. Since register integration only eliminates the execute stage from the processing of an integrated instruction, it does not directly reduce the number of instructions fetched. In other words, it frees up execution bandwidth for new instructions, but does not directly free up more fetch bandwidth to fetch those new instructions. Register integration produces its speedups via second-order effects—primarily, by accelerating the resolution of mispredicted branches, reducing the number of instructions fetched and expediting the fetch of correct path instructions. Register integration actually accelerates branch resolution in two ways. First, if either a mispredicted branch or part of its computation is successfully integrated, then the branch completes and resolves earlier as its dataflow graph is essentially compressed. Second, integrated instructions that do not contribute directly to the computation of a mispredicted branch remove themselves from scheduling and resource contention with instructions that do, allowing the former to execute earlier. Via accelerated misprediction resolution, register integration is able to reduce the number of instructions fetched by up to 5% with an average of about 2%. As we reasoned, for squash reuse these figures correlate well with end performance improvement. This correlation breaks down for pre-execution reuse which achieves speedups by prefetching as well.

One opportunity for integration to do harm is by precipitating squashes via mis-integration. Although not shown in the table, this is the case in most of the benchmarks. Load integration suppression can eliminate the bulk of this problem. Our aggressive LISP keeps the number of mis-integrations three orders of magnitude below the number of integrated results. This is sufficient to ensure that in the squash reuse scenario, mis-integrations are not responsible for producing a slowdown.

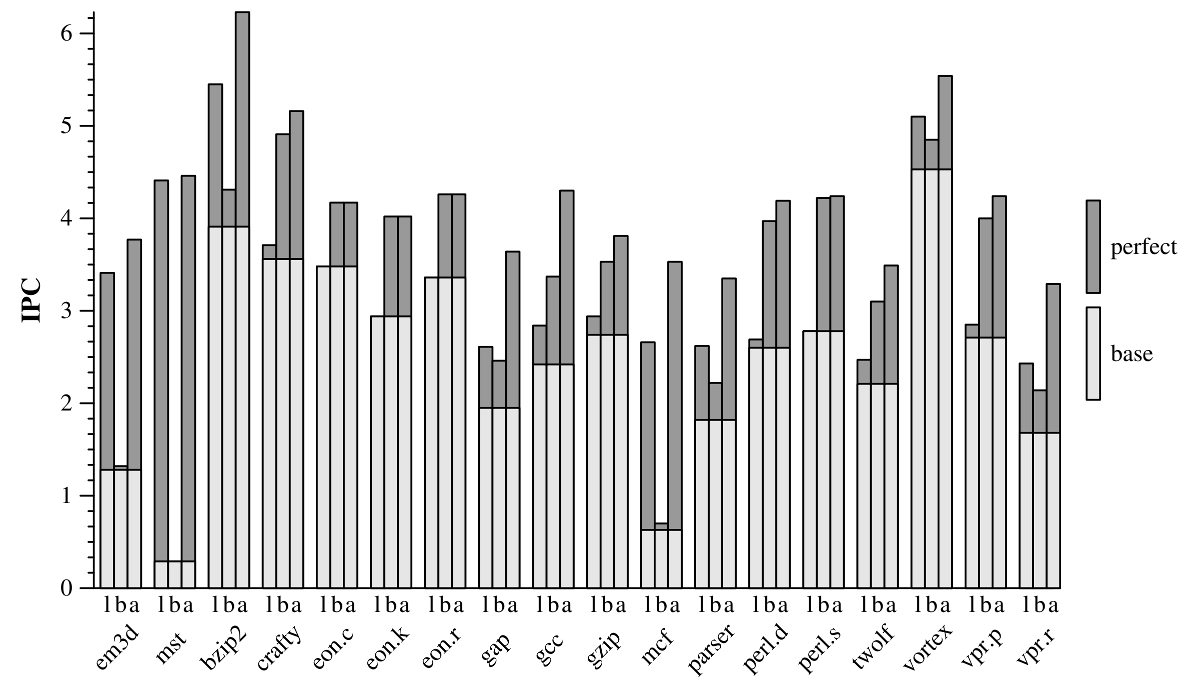
The degree to which a program benefits from squash reuse is highly dependent on the program's structure. Certain programs are naturally amenable to integration-based squash-reuse while some cannot take advantage of it. Factors include the unoptimized incidence of squashes, the size of the code within the conditional arms, which effects the probability of executing reconvergent code before the branch is resolved, and the data dependence nature of work in the reconvergent regions. We do not discuss the structural properties of individual programs in this context, but point the reader to previous work [70]. That work also contains a thorough evaluation of register integration in squash reuse capacity, including analysis of sensitivity to parameters like IT size and associativity and integration latency.

Squash reuse can also be implemented using a *reuse buffer (RB)* [82], a predecessor to the IT that based reuse on values and architectural register names rather than on microarchitectural physical register names. Due to their fundamentally different implementations, the squash reuse captured by the RB and that captured by the IT do not exactly overlap. However, a good, generally applicable discussion of squash reuse and its relation to the structural properties of programs can also be found here [81].

5.2.2 Absolute Upper Bound on DDMT Performance

Since DDMT attacks memory and branch misprediction resolution latencies, the absolute upper bound for performance of the DDMT system in its entirety is the performance of the program with all of these latencies "perfected"—i.e., magically eliminated. Note, we do not model *zero latencies* for branches and loads. To make this study more relevant to DDMT, we model *perfect branch resolution* rather than perfect branch prediction. We allow branch mispredictions to take place and instructions to be fetched along the wrong path. However, we correct all branch mispredictions magically at register renaming—emulating the performance effect of integrating a pre-executed branch—such that wrong path instructions are not executed. The latency of a perfect load is 3 cycles—the cache hit latency. Perfect caches are modeled in a straightforward way.

Figure 5.1 Performance potential of perfect load and branch resolution latencies.



The performance potential for perfect memory and branch resolution is shown in Figure 5.1 and in Table 5.7. The graph shows speedups due to perfecting loads (l, the left-most bars in each group), speedups due to perfecting branches (b) and speedups due to perfecting both loads and branches together (a). The table breaks down supporting data similarly.

Even with perfect memory and perfect branch resolution, the performance of most programs does not approach the peak capabilities of the machine—i.e., IPC of 8. A major part of this is due to our modeling of *perfect branch resolution,* rather than perfect branch prediction. In this model, the cost in cycles of each branch misprediction is (at least) the number of pipeline stages between fetch and register renaming—6 in our model. Other factors that limit performance are

Table 5.7 Performance potential of perfect load and branch resolution latencies.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
Load	Static PIs	150	298	931	3661	1367	1520	1557	5178	26590
	PIIs executed (M)	39.16	28.75	1509.20	731.49	49.95	226.45	72.85	218.49	116.15
	PDIs covered (M)	11.69	11.75	40.13	5.76	0.07	0.21	0.10	3.94	2.08
	Dyn. eff. (%)	29.85	40.87	2.66	0.79	0.14	0.09	0.13	1.81	1.79
	IPC	**3.41**	**4.41**	**5.45**	**3.71**	**3.48**	**2.94**	**3.36**	**2.61**	**2.84**
	Speedup (%)	**166.50**	**1404.29**	**39.12**	**4.37**	**0.09**	**0.09**	**0.16**	**33.67**	**17.74**
Branch	Static PIs	146	260	646	2308	597	664	681	4152	21132
	PIIs executed (M)	6.22	27.27	782.07	314.55	23.27	105.17	35.56	146.21	90.55
	PDIs covered (M)	0.99	0.01	13.00	18.98	0.85	7.86	1.76	7.01	5.01
	Dyn. eff. (%)	15.86	0.04	1.66	6.03	3.65	7.47	4.95	4.80	5.53
	IPC	**1.32**	**0.29**	**4.31**	**4.91**	**4.17**	**4.02**	**4.26**	**2.46**	**3.37**
	Speedup (%)	**2.98**	**0.03**	**10.19**	**37.98**	**19.77**	**36.68**	**26.78**	**26.30**	**39.29**
Both	Dyn. eff. (%)	28.60	21.08	2.35	2.36	1.25	2.43	1.71	3.06	3.66
	IPC	**3.77**	**4.46**	**6.23**	**5.16**	**4.17**	**4.02**	**4.26**	**3.64**	**4.30**
	Speedup (%)	**194.74**	**1421.81**	**59.10**	**45.05**	**19.81**	**36.75**	**26.94**	**86.81**	**78.01**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Load	Static PIs	1156	457	4357	5214	2181	2486	9681	572	2580
	PIIs executed (M)	1081.16	246.03	295.70	971.91	778.86	291.56	458.21	79.27	327.17
	PDIs covered (M)	34.39	83.28	13.22	7.22	1.02	23.47	3.32	3.05	13.62
	Dyn. eff. (%)	3.18	33.85	4.47	0.74	0.13	8.05	0.72	3.85	4.16
	IPC	**2.94**	**2.66**	**2.62**	**2.69**	**2.78**	**2.47**	**5.10**	**2.85**	**2.43**
	Speedup (%)	**7.10**	**324.27**	**43.57**	**3.63**	**0.26**	**12.03**	**12.57**	**5.33**	**44.77**
Branch	Static PIs	847	382	3833	3407	1191	1717	6269	394	1484
	PIIs executed (M)	662.62	185.10	224.22	519.09	389.31	180.25	300.16	35.04	129.31
	PDIs covered (M)	29.31	9.09	9.36	36.47	23.28	17.02	2.25	3.61	6.98
	Dyn. eff. (%)	4.42	4.91	4.17	7.03	5.98	9.44	0.75	10.29	5.40
	IPC	**3.53**	**0.70**	**2.22**	**3.97**	**4.22**	**3.10**	**4.85**	**4.00**	**2.14**
	Speedup (%)	**28.82**	**11.81**	**21.63**	**52.88**	**52.12**	**40.48**	**7.08**	**47.71**	**27.35**
Both	Dyn. eff. (%)	3.70	23.00	4.57	2.92	2.06	9.14	0.74	5.99	4.68
	IPC	**3.81**	**3.53**	**3.35**	**4.19**	**4.24**	**3.49**	**5.54**	**4.24**	**3.29**
	Speedup (%)	**38.85**	**462.68**	**83.50**	**61.13**	**52.69**	**58.03**	**22.16**	**56.61**	**95.48**

real memory disambiguation, a realistic mixture of functional unit resources, non-unit execution latencies (especially the 3-cycle load latency) and the naturally a vailable in-window parallelism of the program.

In addition to IPC and speedup—which we measure over the RI configuration—the table presents four quantities that are relevant to pre-execution. *Static PIs* is the number of static problem instructions perfected. In this case, since we are modeling perfect branch resolution and perfect memory latency, this number includes every static branch with at least one misprediction and every static load with at least one cache miss—essentially every static load and branch in the sample. *PIIs executed* is the count of all dynamic instances of all problem instructions. *PDIs covered* is the count of all performance degrading instances of all PIs.

The *dynamic efficiency* of a pre-execution implementation or a potential pre-execution implementation can be measured as the number of PDIs covered to PIIs pre-executed (a more accurate measure would be the number of PDIs covered to instructions pre-executed, but such a measure can't be taken at this point). If the dynamic efficiency is low, then pre-execution is doing a lot of work for little return. As Table 5.7 shows, an unconstrained PI definition—i.e., "perfecting" or attacking all PDIs—is a low-efficiency approach. Except for the Olden micro-benchmarks and *mcf*, this strategy would require executing at least 10 PIIs, and sometimes as many as 100 PIIs for every PDI. This observation motivates our use of a narrow PI definition.

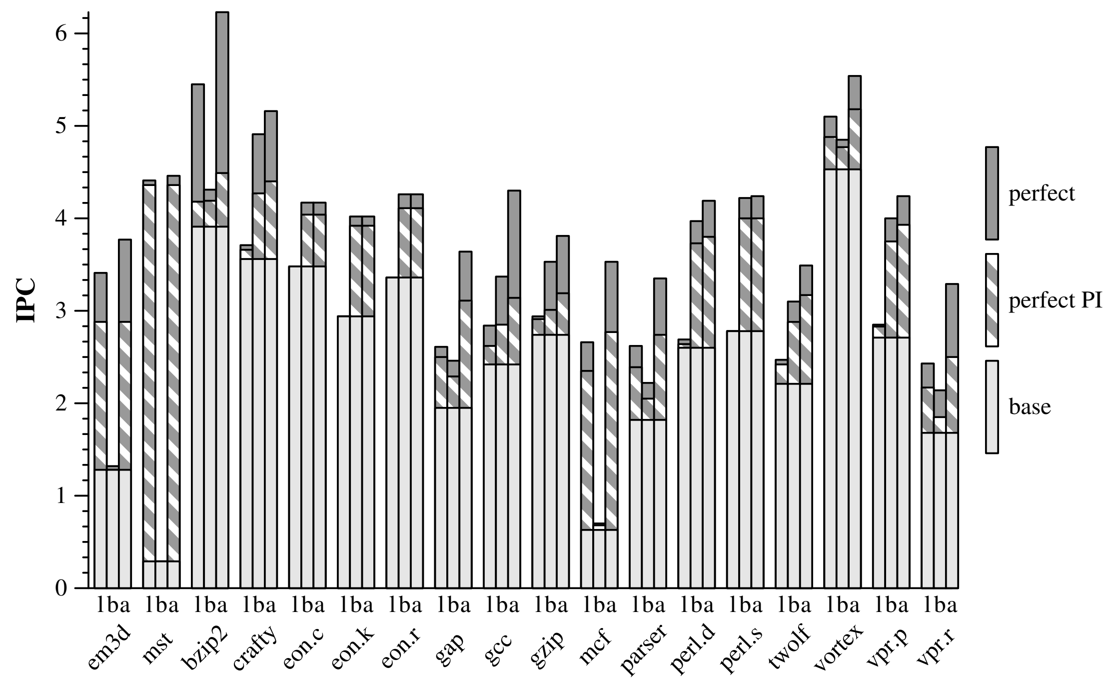
5.2.3 Performance Potential of Perfecting Problem Instructions

DDMT does not attack *all* PDIs. Instead, it relies on the fact that most PDIs are caused by a few static problem instructions (PIs) and that the dynamic efficiency of pre-executing PI instances is relatively much higher than that of the general instruction population. In this section, we evaluate the performance potential of DDMT using a restricted PI definition—a set of thresholds for distinguishing PIs from non-PIs. This section also serves as an evaluation of the PI definition itself. We perform this evaluation by comparing measures of the restricted definition to the corresponding ones in the unrestricted case.

Recall, there are three aspects to the "problemness" of an instruction: the average *problem latency*, the *problem ratio* and the *problem contribution*. For our central design point we have chosen our definition thresholds as follows. The problem contribution threshold is 0.2%, meaning that a PI must contribute at least one in every 500 PDIs of its kind in the program—e.g., a branch PI must contribute at least one in every 500 mispredictions. The problem ratio threshold is 10%, meaning that in order to be considered a PI at least one in every 10 PIIs must be a PDI. Our problem penalty thresholds are 10 cycles for loads and 5 cycles for branches, the minimum latencies for an L1 cache miss and the resolution of a branch mis-prediction, respectively. The graph in Figure 5.2 shows the performance potential of perfecting PIs that obey this definition. The data is overlaid on the graph from Figure 5.1. Again, performance potential is shown for perfecting problem *loads only (l)*, problem *branches only (b)*, and both problem *loads and branches (a)*. Tables 5.8, 5.9 and 5.10 provide details that can be used to "evaluate" the PI definition.

To evaluate a PI definition, we compare its metrics—i.e., the number of static PIs it includes, the number of PDIs it covers, the number of PIIs it pre-executes to

Figure 5.2 Performance potential of perfecting PIs.



do so, and the speedup it produces—to the corresponding metrics of the unconstrained definition from the previous section. The three tables perform these comparisons for loads, branches and a comprehensive PI definition. Each table is divided into two row groups—the top group labeled *All* reprises the metrics of the unconstrained definition, the bottom row group labeled *PIs* shows the metrics for the new, constrained definition. The bottom four rows of the PIs experiment show the raw unconstrained metrics as percentages of their constrained counterparts. We call these percentages "coverages". For instance, the *PDI coverage* of a PI definition is its percentage of PDIs covered vis-a-vis the perfect memory system and oracle branch resolution model. Its *speedup coverage* is the speedup it achieves as a percentage of the total possible speedup. A good PI definition will have high

Table 5.8 Performance potential of perfecting load PIs.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
All	Static PIs	150	298	931	3661	1367	1520	1557	5178	26590
	PIIs executed (M)	39.16	28.75	1509.20	731.49	49.95	226.45	72.85	218.49	116.15
	PDIs covered (M)	11.69	11.75	40.13	5.76	0.07	0.21	0.10	3.94	2.08
	Dyn. eff. (%)	29.85	40.87	2.66	0.79	0.14	0.09	0.13	1.81	1.79
	IPC	**3.41**	**4.41**	**5.45**	**3.71**	**3.48**	**2.94**	**3.36**	**2.61**	**2.84**
	Speedup (%)	**166.50**	**1404.29**	**39.12**	**4.37**	**0.09**	**0.09**	**0.16**	**33.67**	**17.74**
PIs	Static PIs	10	6	11	32	4	6	4	31	83
	PIIs executed (M)	18.28	12.67	18.06	14.95	0.11	0.33	0.08	6.47	3.42
	PDIs covered (M)	11.60	11.74	9.05	3.03	0.05	0.13	0.05	2.64	0.85
	Dyn. eff. (%)	63.43	92.66	50.13	20.29	49.69	38.99	62.43	40.87	24.99
	IPC	**3.41**	**4.41**	**5.45**	**3.71**	**3.48**	**2.94**	**3.36**	**2.61**	**2.84**
	Speedup (%)	**125.16**	**1385.92**	**6.75**	**2.77**	**0.05**	**0.03**	**0.04**	**28.24**	**8.41**
	PI coverage (%)	6.67	2.01	1.18	0.87	0.29	0.39	0.26	0.60	0.31
	PII coverage (%)	46.69	44.09	1.20	2.04	0.21	0.15	0.11	2.96	2.95
	PDI coverage (%)	99.21	99.94	22.56	52.64	73.36	60.94	53.16	67.00	41.06
	Spd. coverage(%)	75.17	98.69	17.26	63.45	51.52	34.62	28.30	83.89	47.44

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
All	Static PIs	1156	457	4357	5214	2181	2486	9681	572	2580
	PIIs executed (M)	1081.16	246.03	295.70	971.91	778.86	291.56	458.21	79.27	327.17
	PDIs covered (M)	34.39	83.28	13.22	7.22	1.02	23.47	3.32	3.05	13.62
	Dyn. eff. (%)	3.18	33.85	4.47	0.74	0.13	8.05	0.72	3.85	4.16
	IPC	**2.94**	**2.66**	**2.62**	**2.69**	**2.78**	**2.47**	**5.10**	**2.85**	**2.43**
	Speedup (%)	**7.10**	**324.27**	**43.57**	**3.63**	**0.26**	**12.03**	**12.57**	**5.33**	**44.77**
PIs	Static PIs	3	21	46	56	10	39	28	18	20
	PIIs executed (M)	79.36	113.70	23.70	20.82	0.78	35.71	5.80	6.40	26.69
	PDIs covered (M)	20.89	81.02	9.84	4.35	0.20	19.66	1.72	2.43	9.06
	Dyn. eff. (%)	26.32	71.26	41.51	20.90	25.81	55.04	29.60	37.96	33.95
	IPC	**2.94**	**2.66**	**2.62**	**2.69**	**2.78**	**2.47**	**5.10**	**2.85**	**2.43**
	Speedup (%)	**6.01**	**275.78**	**30.96**	**1.56**	**0.05**	**9.59**	**7.64**	**4.49**	**29.27**
	PI coverage (%)	0.26	4.60	1.06	1.07	0.46	1.57	0.29	3.15	0.78
	PII coverage (%)	7.34	46.21	8.02	2.14	0.10	12.25	1.27	8.07	8.16
	PDI coverage (%)	60.75	97.29	74.44	60.30	19.64	83.74	51.73	79.64	66.54
	Spd. coverage(%)	84.59	85.05	71.05	42.90	17.81	79.66	60.74	84.20	65.38

Table 5.9 Performance potential of perfecting branch PIs.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
All	Static PIs	146	260	646	2308	597	664	681	4152	21132
	PIIs executed (M)	6.22	27.27	782.07	314.55	23.27	105.17	35.56	146.21	90.55
	PDIs covered (M)	0.99	0.01	13.00	18.98	0.85	7.86	1.76	7.01	5.01
	Dyn. eff. (%)	15.86	0.04	1.66	6.03	3.65	7.47	4.95	4.80	5.53
	IPC	**1.32**	**0.29**	**4.31**	**4.91**	**4.17**	**4.02**	**4.26**	**2.46**	**3.37**
	Speedup (%)	**2.98**	**0.03**	**10.19**	**37.98**	**19.77**	**36.68**	**26.78**	**26.30**	**39.29**
PIs	Static PIs	0	0	24	68	38	60	53	78	81
	PIIs executed (M)	0.00	0.00	35.08	56.69	2.94	21.79	6.33	21.32	7.64
	PDIs covered (M)	0.00	0.00	6.79	10.83	0.64	6.79	1.45	4.98	2.36
	Dyn. eff. (%)	0.00	0.00	19.36	19.11	21.63	31.19	22.85	23.37	30.95
	IPC	**1.32**	**0.29**	**4.31**	**4.91**	**4.17**	**4.02**	**4.26**	**2.46**	**3.37**
	Speedup (%)	**0.00**	**0.00**	**6.93**	**19.88**	**16.04**	**33.43**	**22.46**	**17.43**	**18.03**
	PI coverage (%)	0.00	0.00	3.72	2.95	6.37	9.04	7.78	1.88	0.38
	PII coverage (%)	0.00	0.00	4.49	18.02	12.63	20.72	17.81	14.58	8.44
	PDI coverage (%)	0.00	0.00	52.27	57.08	74.92	86.46	82.20	71.03	47.23
	Spd. coverage(%)	0.00	0.00	67.99	52.34	81.13	91.13	83.88	66.28	45.90

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
All	Static PIs	847	382	3833	3407	1191	1717	6269	394	1484
	PIIs executed (M)	662.62	185.10	224.22	519.09	389.31	180.25	300.16	35.04	129.31
	PDIs covered (M)	29.31	9.09	9.36	36.47	23.28	17.02	2.25	3.61	6.98
	Dyn. eff. (%)	4.42	4.91	4.17	7.03	5.98	9.44	0.75	10.29	5.40
	IPC	**3.53**	**0.70**	**2.22**	**3.97**	**4.22**	**3.10**	**4.85**	**4.00**	**2.14**
	Speedup (%)	**28.82**	**11.81**	**21.63**	**52.88**	**52.12**	**40.48**	**7.08**	**47.71**	**27.35**
PIs	Static PIs	28	19	78	31	28	59	33	27	16
	PIIs executed (M)	65.23	26.59	30.04	64.94	50.71	65.31	4.16	10.81	17.64
	PDIs covered (M)	12.81	5.89	5.81	30.93	21.00	14.03	1.54	2.93	4.98
	Dyn. eff. (%)	19.64	22.16	19.34	47.63	41.42	21.49	36.91	27.14	28.25
	IPC	**3.53**	**0.70**	**2.22**	**3.97**	**4.22**	**3.10**	**4.85**	**4.00**	**2.14**
	Speedup (%)	**9.66**	**7.75**	**12.56**	**43.48**	**44.01**	**30.32**	**5.18**	**38.54**	**10.10**
	PI coverage (%)	3.31	4.97	2.03	0.91	2.35	3.44	0.53	6.85	1.08
	PII coverage (%)	9.85	14.37	13.40	12.51	13.03	36.23	1.39	30.84	13.64
	PDI coverage (%)	43.72	64.82	62.08	84.81	90.22	82.44	68.36	81.30	71.41
	Spd. coverage(%)	33.51	65.68	58.05	82.23	84.45	74.91	73.18	80.79	36.95

Table 5.10 Performance potential of perfecting load and branch PIs.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
All	Static PIs	296	558	1577	5969	1964	2184	2238	9330	47722
	PIIs executed (M)	45.39	56.02	2291.27	1046.04	73.22	331.62	108.41	364.70	206.70
	PDIs covered (M)	12.98	11.81	53.92	24.74	0.91	8.06	1.85	11.18	7.56
	Dyn. eff. (%)	28.60	21.08	2.35	2.36	1.25	2.43	1.71	3.06	3.66
	IPC	**3.77**	**4.46**	**6.23**	**5.16**	**4.17**	**4.02**	**4.26**	**3.64**	**4.30**
	Speedup (%)	**194.74**	**1421.81**	**59.10**	**45.05**	**19.81**	**36.75**	**26.94**	**86.81**	**78.01**
PIs	Static PIs	10	6	35	100	42	66	57	109	164
	PIIs executed (M)	18.28	12.67	53.14	71.64	3.05	22.12	6.42	27.79	11.06
	PDIs covered (M)	11.60	11.74	16.42	13.83	0.69	6.93	1.50	7.69	3.27
	Dyn. eff. (%)	63.43	92.66	30.89	19.31	22.51	31.34	23.37	27.68	29.54
	IPC	**3.77**	**4.46**	**6.23**	**5.16**	**4.17**	**4.02**	**4.26**	**3.64**	**4.30**
	Speedup (%)	**125.16**	**1385.92**	**14.61**	**23.71**	**16.04**	**33.43**	**22.47**	**59.36**	**29.86**
	PI coverage (%)	3.38	1.08	2.22	1.68	2.14	3.02	2.55	1.17	0.34
	PII coverage (%)	40.29	22.62	2.32	6.85	4.16	6.67	5.92	7.62	5.35
	PDI coverage (%)	89.34	99.45	30.45	55.91	75.01	85.97	80.87	68.83	43.20
	Spd. coverage(%)	64.27	97.48	24.72	52.63	80.97	90.95	83.41	68.38	38.28

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
All	Static PIs	2003	839	8190	8621	3372	4203	15950	966	4064
	PIIs executed (M)	1743.78	431.14	519.92	1491.00	1168.17	471.80	758.38	114.31	456.48
	PDIs covered (M)	64.45	99.15	23.78	43.57	24.12	43.15	5.61	6.85	21.37
	Dyn. eff. (%)	3.70	23.00	4.57	2.92	2.06	9.14	0.74	5.99	4.68
	IPC	**3.81**	**3.53**	**3.35**	**4.19**	**4.24**	**3.49**	**5.54**	**4.24**	**3.29**
	Speedup (%)	**38.85**	**462.68**	**83.50**	**61.13**	**52.69**	**58.03**	**22.16**	**56.61**	**95.48**
PIs	Static PIs	31	40	124	87	38	98	61	45	36
	PIIs executed (M)	144.60	140.29	53.74	85.76	51.49	101.02	9.97	17.20	44.34
	PDIs covered (M)	33.58	89.67	15.95	35.26	21.20	35.65	3.26	5.47	14.79
	Dyn. eff. (%)	23.22	63.91	29.67	41.11	41.18	35.29	32.70	31.77	33.36
	IPC	**3.81**	**3.53**	**3.35**	**4.19**	**4.24**	**3.49**	**5.54**	**4.24**	**3.29**
	Speedup (%)	**16.32**	**341.58**	**50.10**	**46.34**	**44.10**	**43.53**	**14.17**	**45.36**	**48.66**
	PI coverage (%)	1.55	4.77	1.51	1.01	1.13	2.33	0.38	4.66	0.89
	PII coverage (%)	8.29	32.54	10.34	5.75	4.41	21.41	1.31	15.05	9.71
	PDI coverage (%)	52.10	90.43	67.06	80.92	87.91	82.63	58.13	79.77	69.23
	Spd. coverage(%)	42.00	73.83	60.00	75.79	83.71	75.01	63.95	80.11	50.96

PDI and speedup coverages and low PI and PII coverages. In other words, a good PI definition will cover the largest fraction of PDIs and achieve the largest fraction of the possible speedup while doing the least amount of work both statically and dynamically.

This data shows that a narrow PI definition has a higher inherent dynamic efficiency than an unconstrained definition, while achieving a large fraction of the speedup associated with the latter. The PI definition we have chosen "performs" well (in an ideal sense) as judged by these metrics, achieving greater than 60% PDI and speedup coverage with less than 15% PII coverage and less than 3% PI coverage on most programs. Section 5.3.2.1 measures DDMT's sensitivity to the PI definition.

5.2.4 DDT Characterization

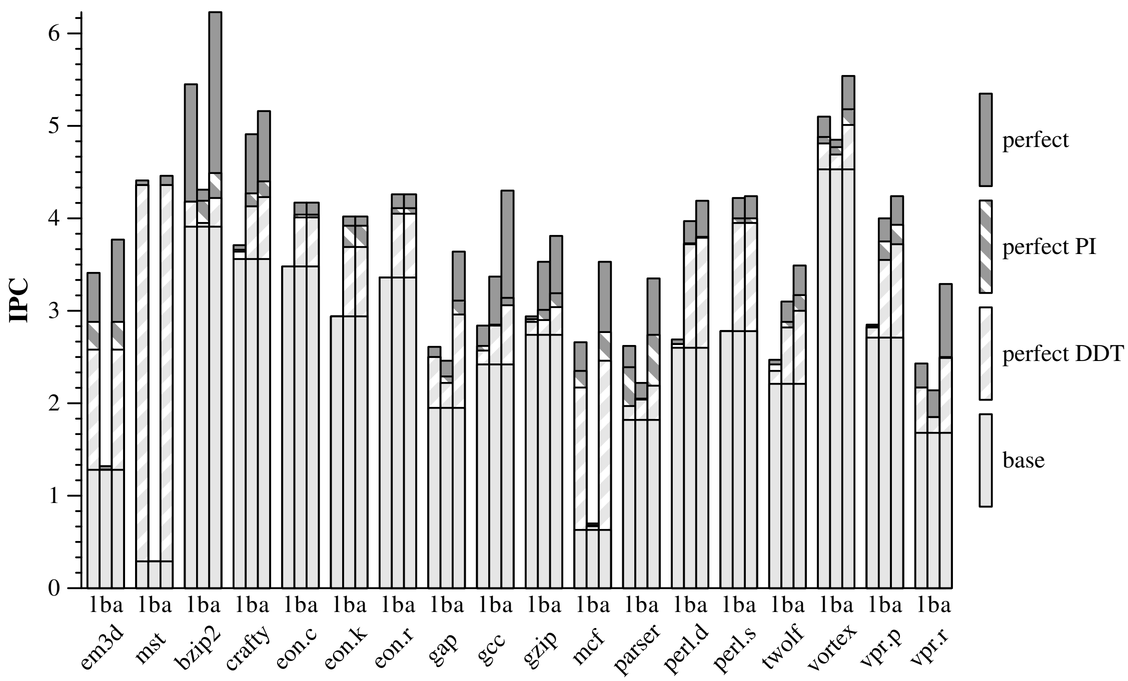
We select DDTs to cover PDIs of instructions that fit the PI definition. In this section, we "evaluate" these DDTs against the PI definition in the same way we evaluated the PI definition against the unconstrained definition. Specifically, we measure DDT "coverage" of the PI definition using the same metrics.

We quickly review the DDT selection parameters. We set DDT selection *scope*—the number of instructions older than the PDI the slice generator can look at—at 1024 instructions. *Maximum DDT length* is 32 instructions. The *minimum latency coverage acceptability factor (LCAF)* is 25%, meaning that any acceptable DDT must achieve at least 25% of the PDI's desired latency tolerance, LT_{des}. The *maximum unrolling degree* is 4. This is a concession to a realistic IT configuration which would be at most 4-way set-associative, meaning that the maximum number of un-integrated data-driven instructions with matching PCs is 4.

There are two ways to characterize DDTs. First, we may characterize them using measures observed over the DDT selection data set. Since in the *limit* scenario we use the same data set to select DDTs as we do to pre-execute them, these measures also represent fairly accurate projections for pre-execution. A second way of characterizing DDTs is to repeat our perfect PI experiment, perfecting only PIs for which DDTs were selected. Neither of these methods is ideal. The first provides accurate projections of PII and PDI coverage, but is not associated with a speedup. The second measures a speedup, but does not accurately reflect the number of PDIs that will actually be covered or the number of PIIs that will actually be pre-executed to cover them. Not all PDIs of a given PI will be covered by the chosen DDTs, and of those that are covered not all will be covered in full—i.e., their latency will only be partially tolerated. These shortfalls are reflected in the statistics collected during DDT selection, but not in the perfect DDT experiment. In the absence of a clear winner, we use both methods. Figure 5.3 is a copy of the previous graph on which we overlay the performance potential of perfecting *all* PDIs of those PIs for which *any* DDTs were found. This experiment is called *perfect DDT*.

A characterization of the DDTs is shown in three tables. Table 5.11 characterizes a set of DDTs that pre-executes only loads, Table 5.12 characterizes DDTs that pre-execute only branches and Table 5.13 characterizes a set of DDTs that

Figure 5.3 Performance potential of perfecting PIs for which DDTs were found.



pre-executes both loads and branches. The first experiment in each table reprises our raw data from the evaluation of the PI definition in Table 5.8. We include the static PI count, the counts of dynamic PIIs executed and PDIs covered and the potential speedup. The first row group of the *DDTs* experiment parallels this data using the observed (projected) statistics from the DDT selection process. *Static PIs* is the number of static PIs for which DDTs were found. *PIIs executed* is the number of projected PII executions. Note, the projected number of PIIs executed may in fact be *greater* than the corresponding number of PIIs found in the program sample, as DDTs can be triggered when no corresponding master thread PII is forthcoming. *PDIs covered* is the number of observed (projected) PDIs covered by the DDTs. *PDIs fully covered* is the number of PDIs whose latency we

Table 5.11 Characterizing load DDTs.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Perfect PIs	Static PIs	10	6	11	32	4	6	4	31	83
	PIIs executed (M)	18.28	12.67	18.06	14.95	0.11	0.33	0.08	6.47	3.42
	PDIs covered (M)	11.60	11.74	9.05	3.03	0.05	0.13	0.05	2.64	0.85
	Speedup (%)	125.16	1385.92	6.75	2.77	0.05	0.03	0.04	28.24	8.41
DDTs	Static PIs	10	6	9	12	2	5	3	22	52
	PIIs executed (M)	19.06	12.34	16.67	7.76	0.15	0.94	0.19	4.75	3.81
	PDIs covered (M)	9.65	11.03	9.02	1.31	0.05	0.13	0.05	0.98	0.40
	PDIs fully cov'd (M)	5.11	8.59	1.78	1.05	0.03	0.07	0.03	0.20	0.16
	PI cov'g (%)	**100.00**	**83.33**	**81.82**	**37.50**	**50.00**	**83.33**	**75.00**	**70.97**	**62.65**
	PII cov'g (%)	**104.23**	**97.39**	**92.30**	**51.92**	**144.98**	**282.50**	**232.79**	**73.45**	**111.28**
	PDI cov'g (%)	**83.20**	**93.95**	**99.57**	**43.25**	**95.81**	**98.99**	**97.89**	**37.27**	**46.36**
	PDI full cov'g (%)	**44.03**	**73.12**	**19.64**	**34.69**	**47.91**	**55.94**	**51.66**	**7.56**	**18.86**
Perfect DDTs	PIIs executed (M)	15.70	12.67	16.66	10.57	0.05	0.31	0.08	6.11	3.04
	PDIs covered (M)	9.43	11.74	8.90	2.29	0.05	0.13	0.05	2.57	0.70
	Speedup (%)	101.69	1385.92	6.73	2.23	0.05	0.03	0.05	28.17	6.48
	PII cov'g (%)	**85.87**	**100.00**	**92.26**	**70.75**	**47.61**	**93.65**	**99.35**	**94.42**	**88.94**
	PDI cov'g (%)	**81.31**	**100.00**	**98.30**	**75.68**	**86.71**	**97.38**	**99.56**	**97.12**	**82.28**
	Speedup cov'g (%)	**81.25**	**100.00**	**99.66**	**80.43**	**94.12**	**100.00**	**106.67**	**99.75**	**77.03**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Perfect PIs	Static PIs	3	21	46	56	10	39	28	18	20
	PIIs executed (M)	79.36	113.70	23.70	20.82	0.78	35.71	5.80	6.40	26.69
	PDIs covered (M)	20.89	81.02	9.84	4.35	0.20	19.66	1.72	2.43	9.06
	Speedup (%)	6.01	275.78	30.96	1.56	0.05	9.59	7.64	4.49	29.27
DDTs	Static PIs	1	14	38	37	4	25	16	14	15
	PIIs executed (M)	16.03	89.61	28.35	13.76	0.37	48.01	1.65	8.70	23.28
	PDIs covered (M)	6.77	60.41	5.66	2.28	0.12	15.94	1.03	1.94	6.03
	PDIs fully cov'd (M)	0.00	0.66	1.52	1.24	0.12	9.76	0.47	1.65	5.61
	PI cov'g (%)	**33.33**	**66.67**	**82.61**	**66.07**	**40.00**	**64.10**	**57.14**	**77.78**	**75.00**
	PII cov'g (%)	**20.20**	**78.81**	**119.60**	**66.10**	**48.04**	**134.46**	**28.51**	**136.02**	**87.21**
	PDI cov'g (%)	**32.39**	**74.56**	**57.48**	**52.47**	**57.66**	**81.10**	**60.24**	**79.98**	**66.49**
	PDI full cov'g (%)	**0.00**	**0.82**	**15.49**	**28.48**	**57.66**	**49.65**	**27.51**	**67.74**	**61.86**
Perfect DDTs	PIIs executed (M)	74.58	95.53	19.45	20.71	0.61	31.35	5.32	5.37	26.69
	PDIs covered (M)	17.74	66.53	6.61	4.28	0.17	16.86	1.48	2.20	9.06
	Speedup (%)	4.95	246.07	8.06	1.50	0.03	6.65	6.01	4.21	29.27
	PII cov'g (%)	**93.97**	**84.02**	**82.06**	**99.49**	**79.08**	**87.79**	**91.67**	**83.86**	**100.00**
	PDI cov'g (%)	**84.90**	**82.11**	**67.23**	**98.38**	**84.10**	**85.76**	**86.43**	**90.79**	**100.00**
	Speedup cov'g (%)	**82.39**	**89.23**	**26.02**	**96.30**	**61.54**	**69.41**	**78.74**	**93.74**	**100.00**

Table 5.12 Characterizing branch DDTs.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Perfect PIs	Static PIs	0	0	24	68	38	60	53	78	81
	PIIs executed (M)	0.00	0.00	35.08	56.69	2.94	21.79	6.33	21.32	7.64
	PDIs covered (M)	0.00	0.00	6.79	10.83	0.64	6.79	1.45	4.98	2.36
	Speedup (%)	0.00	0.00	6.93	19.88	16.04	33.43	22.46	17.43	18.03
DDTs	Static PIs	0	0	3	41	29	43	40	50	65
	PIIs executed (M)	0.00	0.00	3.89	27.47	1.82	15.89	4.77	14.78	6.13
	PDIs covered (M)	0.00	0.00	0.85	4.88	0.32	3.56	0.79	2.24	1.40
	PDIs fully cov'd (M)	0.00	0.00	0.52	4.17	0.23	2.77	0.64	1.47	0.66
	PI cov'g (%)	**nan**	**nan**	**12.50**	**60.29**	**76.32**	**71.67**	**75.47**	**64.10**	**80.25**
	PII cov'g (%)	**nan**	**nan**	**11.08**	**48.46**	**61.87**	**72.93**	**75.38**	**69.34**	**80.22**
	PDI cov'g (%)	**nan**	**nan**	**12.58**	**45.05**	**50.87**	**52.37**	**54.58**	**45.04**	**59.20**
	PDI full cov'g (%)	**nan**	**nan**	**7.68**	**38.47**	**36.92**	**40.71**	**44.01**	**29.41**	**27.70**
Perfect DDTs	PIIs executed (M)	0.00	0.00	5.00	46.81	2.76	17.07	5.87	18.43	6.54
	PDIs covered (M)	0.00	0.00	1.31	9.34	0.59	5.71	1.37	4.23	2.27
	Speedup (%)	0.00	0.00	0.90	16.04	15.25	25.57	20.56	13.78	17.51
	PII cov'g (%)	**nan**	**nan**	**14.25**	**82.57**	**93.74**	**78.37**	**92.70**	**86.41**	**85.60**
	PDI cov'g (%)	**nan**	**nan**	**19.29**	**86.23**	**93.41**	**84.07**	**94.81**	**84.85**	**96.07**
	Speedup cov'g (%)	**nan**	**nan**	**13.01**	**80.68**	**95.07**	**76.50**	**91.56**	**79.06**	**97.11**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Perfect PIs	Static PIs	28	19	78	31	28	59	33	27	16
	PIIs executed (M)	65.23	26.59	30.04	64.94	50.71	65.31	4.16	10.81	17.64
	PDIs covered (M)	12.81	5.89	5.81	30.93	21.00	14.03	1.54	2.93	4.98
	Speedup (%)	9.66	7.75	12.56	43.48	44.01	30.32	5.18	38.54	10.10
DDTs	Static PIs	12	13	41	24	17	43	18	15	11
	PIIs executed (M)	38.95	19.25	22.43	56.77	28.55	71.53	1.09	10.72	14.64
	PDIs covered (M)	5.82	2.37	2.01	24.29	11.77	9.60	0.59	2.06	2.50
	PDIs fully cov'd (M)	1.54	0.40	0.67	10.89	4.44	4.79	0.55	1.33	2.26
	PI cov'g (%)	**42.86**	**68.42**	**52.56**	**77.42**	**60.71**	**72.88**	**54.55**	**55.56**	**68.75**
	PII cov'g (%)	**59.70**	**72.39**	**74.68**	**87.42**	**56.29**	**109.52**	**26.24**	**99.16**	**82.99**
	PDI cov'g (%)	**45.45**	**40.28**	**34.58**	**78.51**	**56.03**	**68.39**	**38.35**	**70.26**	**50.21**
	PDI full cov'g (%)	**11.99**	**6.87**	**11.58**	**35.20**	**21.14**	**34.13**	**35.92**	**45.34**	**45.41**
Perfect DDTs	PIIs executed (M)	44.73	22.82	28.10	60.88	45.15	63.36	2.93	9.98	17.42
	PDIs covered (M)	9.00	4.87	5.54	30.71	20.42	13.71	1.19	2.61	4.92
	Speedup (%)	5.80	6.88	11.88	43.29	42.13	27.80	3.57	31.07	9.97
	PII cov'g (%)	**68.57**	**85.81**	**93.54**	**93.75**	**89.02**	**97.01**	**70.30**	**92.34**	**98.76**
	PDI cov'g (%)	**70.27**	**82.71**	**95.28**	**99.28**	**97.22**	**97.73**	**77.45**	**89.09**	**98.74**
	Speedup cov'g (%)	**60.03**	**88.68**	**94.63**	**99.55**	**95.73**	**91.69**	**68.80**	**80.62**	**98.70**

Table 5.13 Characterizing "combination" load and branch DDTs.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Perfect PIs	Static PIs	10	6	35	100	42	66	57	109	164
	PIIs executed (M)	18.28	12.67	53.14	71.64	3.05	22.12	6.42	27.79	11.06
	PDIs covered (M)	11.60	11.74	16.42	13.83	0.69	6.93	1.50	7.69	3.27
	Speedup (%)	125.16	1385.92	14.61	23.71	16.04	33.43	22.47	59.36	29.86
DDTs	Static PIs	10	6	12	53	31	48	43	72	117
	PIIs executed (M)	19.06	12.34	20.56	35.23	1.97	16.83	4.96	19.53	10.27
	PDIs covered (M)	9.65	11.03	9.87	6.19	0.37	3.69	0.84	3.23	1.79
	PDIs fully cov'd (M)	5.11	8.59	2.30	5.22	0.26	2.84	0.66	1.67	0.82
	PI cov'g (%)	**100.00**	**83.33**	**34.29**	**53.00**	**73.81**	**72.73**	**75.44**	**66.06**	**71.34**
	PII cov'g (%)	**104.23**	**97.39**	**38.68**	**49.18**	**64.75**	**76.09**	**77.39**	**70.29**	**92.83**
	PDI cov'g (%)	**83.20**	**93.95**	**60.12**	**44.76**	**54.52**	**53.28**	**56.16**	**41.97**	**54.87**
	PDI full cov'g (%)	**44.03**	**73.12**	**14.01**	**37.74**	**37.92**	**40.95**	**44.24**	**21.64**	**25.02**
Perfect DDTs	PIIs executed (M)	15.70	12.67	21.66	57.38	2.81	17.39	5.95	24.53	9.58
	PDIs covered (M)	9.43	11.74	10.71	11.60	0.64	5.85	1.42	6.83	3.02
	Speedup (%)	101.69	1385.92	7.83	18.85	15.26	25.56	20.57	51.94	26.53
	PII cov'g (%)	**85.87**	**100.00**	**40.76**	**80.10**	**92.15**	**78.60**	**92.78**	**88.27**	**86.63**
	PDI cov'g (%)	**81.31**	**100.00**	**65.25**	**83.88**	**93.89**	**84.36**	**94.98**	**88.72**	**92.41**
	Speedup cov'g (%)	**81.25**	**100.00**	**53.57**	**79.49**	**95.09**	**76.48**	**91.56**	**87.50**	**88.86**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Perfect PIs	Static PIs	31	40	124	87	38	98	61	45	36
	PIIs executed (M)	144.60	140.29	53.74	85.76	51.49	101.02	9.97	17.20	44.34
	PDIs covered (M)	33.58	89.67	15.95	35.26	21.20	35.65	3.26	5.47	14.79
	Speedup (%)	16.32	341.58	50.10	46.34	44.10	43.53	14.17	45.36	48.66
DDTs	Static PIs	13	25	79	61	21	68	34	29	25
	PIIs executed (M)	54.98	102.52	50.23	70.05	28.92	119.55	2.75	19.42	37.88
	PDIs covered (M)	12.59	62.06	7.61	26.50	11.88	26.20	1.62	4.00	8.51
	PDIs fully cov'd (M)	1.54	1.07	2.14	12.06	4.56	11.66	1.02	2.97	7.85
	PI cov'g (%)	**41.94**	**62.50**	**63.71**	**70.11**	**55.26**	**69.39**	**55.74**	**64.44**	**69.44**
	PII cov'g (%)	**38.02**	**73.08**	**93.45**	**81.68**	**56.16**	**118.34**	**27.56**	**112.86**	**85.44**
	PDI cov'g (%)	**37.50**	**69.21**	**47.72**	**75.18**	**56.05**	**73.49**	**49.84**	**73.25**	**57.54**
	PDI full cov'g (%)	**4.57**	**1.19**	**13.43**	**34.22**	**21.49**	**32.70**	**31.44**	**54.43**	**53.09**
Perfect DDTs	PIIs executed (M)	119.32	118.35	47.55	81.59	45.76	94.71	8.25	15.34	44.12
	PDIs covered (M)	26.65	74.06	12.45	34.96	20.61	32.36	2.68	4.91	14.73
	Speedup (%)	11.05	292.32	20.04	46.00	42.19	35.86	10.51	37.30	48.41
	PII cov'g (%)	**82.51**	**84.36**	**88.48**	**95.14**	**88.87**	**93.75**	**82.74**	**89.19**	**99.51**
	PDI cov'g (%)	**79.38**	**82.59**	**78.07**	**99.16**	**97.19**	**90.77**	**82.15**	**89.84**	**99.58**
	Speedup cov'g (%)	**67.70**	**85.58**	**40.01**	**99.29**	**95.66**	**82.38**	**74.16**	**82.23**	**99.50**

expect to fully tolerate using the selected DDTs. Recall, the DDT selection algorithm is given a desired level of latency tolerance, LT_{des}, for the instances of every PI and a latency coverage acceptability factor (LCAF) which—if set below 100%—allows it to accept DDTs that will tolerate less than 100% of the desired latency. As a result, the DDT selection algorithm may consciously produce DDTs that will only partially cover their target PDI latencies. Again, these are just approximations. As we will see in the following section, the DDT selection algorithm may both over-estimate and under-estimate the latency tolerance of individual DDTs leading to either a lower-than-expected or higher-than-expected full coverage rate at DDMT runtime. The second row group of the *DDT* experiment shows metric coverages which are computed by dividing the *DDT* observed/projected metric by the corresponding *Perfect PIs* metric. Here, we would like high PI and PDI coverages, and a low PII coverage. A good DDT selection implementation will successfully find DDTs for most PIs, project these DDTs to cover most PDIs—preferably in full—and place triggers along paths that will not result in many pre-executions for either NPDIs or no corresponding PIIs. Of course, failure to achieve proper coverage levels—both high and low for the respective metrics—does not imply that the DDT selection algorithm is faulty. It may simply be that the PDI dependence graphs are such that sufficient latency tolerance may not be possible within the length and scope bounds of the search, or that if such latency tolerance is possible that its PII to PDI ratio is unacceptably high. These failures are the "fault" of the program itself, not of the selection algorithm.

The final experiment shown in each table is *Perfect DDTs*. In this experiment, we perfect the behavior of all PI instances for all PIs for which any DDTs were found. We do this to obtain a tighter performance upper bound for DDMT, although as we explained this upper bound is rather loose.

We briefly discuss "combination" load and branch DDTs. Statistics for load and branch DDTs separately are also provided, but are not discussed as they have the same general character as the combined DDTs. Our two micro-benchmarks have few mis-predicted branches and, hence, only load PIs. Per projected coverage metrics, our DDT selection algorithm performs well on the two micro-benchmarks—*em3d* and *mst*—finding DDTs to cover all static PIs and high percentages—83% and 93%, respectively—of dynamic PDIs. Projected full coverage for PDIs is lower—44% and 73%, respectively. Both benchmarks have high L2 miss rates, whose long latencies require high degrees of unrolling. *Em3d*'s longer inner loops restrict the unrolling degree that can be observed within the slicing scope. As mentioned earlier, to achieve both high PDI coverage and sufficient latency tolerance, it may be necessary to choose DDTs that will be pre-executed even when the likelihood is high that no corresponding master thread PIIs exist. An extreme form of this situation—which is not a positive one—arises in *em3d* and is reflected by a PII coverage that is greater than 100%.

As expected, DDT selection "performs" well, but less so, on the SPEC2000 benchmarks. DDTs are found for between 40% and 80% of PIs, and these typically project to cover between 20% and 75% of PDIs. Projected full coverage is lower still, ranging from 1% to 54%. *Mcf* is an interesting case study. *Mcf* contains tight loops, many with serial (pointer-chasing) latencies. High degrees of unroll-

Table 5.14 Additional characterization of "combination" load and branch DDTs.

	em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Static multi-PDI DDTs	2	2	10	54	29	43	39	96	144
Static single-PDI DDTs include	16	10	14	95	47	78	64	139	259
Avg. static length	15.00	20.50	11.40	10.81	12.83	10.35	9.21	15.08	14.97
DDT executions (M)	3.14	2.47	15.98	22.76	1.38	10.13	2.69	15.87	4.98
Avg. dynamic length	12.42	20.50	13.86	9.70	8.52	8.51	9.14	9.60	11.74
Avg. dynamic unrolling	1.00	1.00	2.26	0.23	0.00	0.00	0.00	0.59	0.36
Avg. dynamic PII/DDT	6.07	5.00	1.29	1.55	1.43	1.66	1.85	1.23	2.06
Avg. dynamic PDI/DDT	**3.07**	**4.47**	**0.62**	**0.27**	**0.27**	**0.36**	**0.31**	**0.20**	**0.36**
Avg. dynamic advantage/DDT	**62.18**	**316.00**	**7.87**	**2.10**	**1.82**	**2.37**	**1.84**	**2.58**	**3.59**
Avg. dynamic advantage/PDI	20.24	70.70	12.74	7.73	6.75	6.51	5.89	12.66	9.98

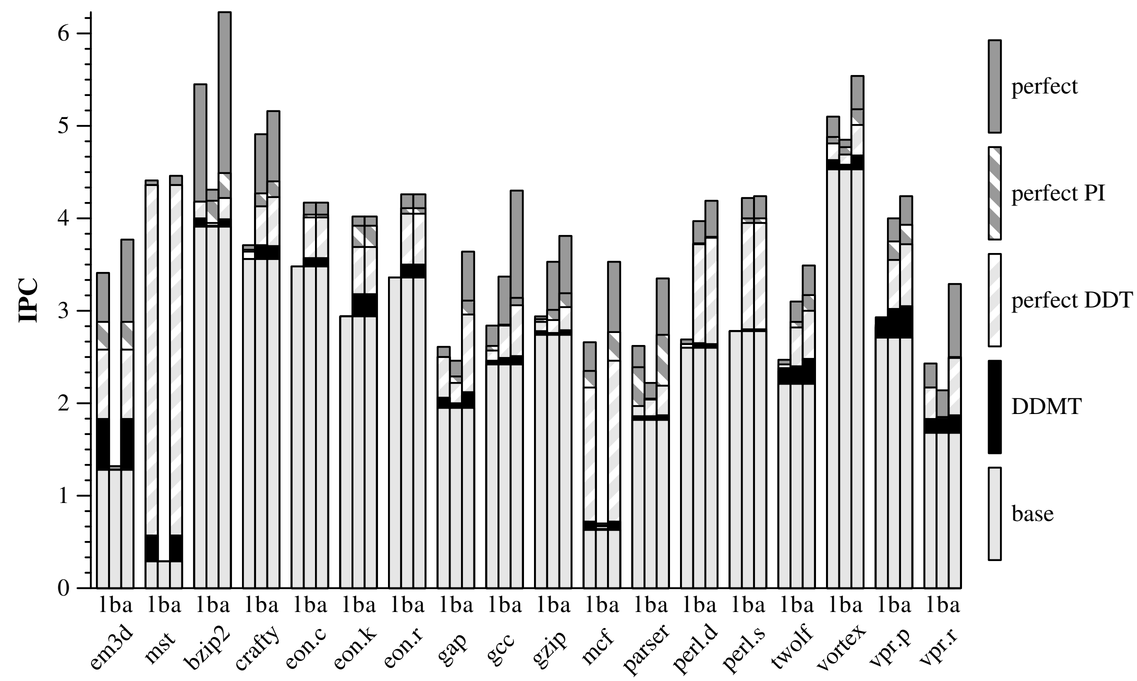
	gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Static multi-PDI DDTs	17	18	83	116	43	84	52	30	32
Static single-PDI DDTs include	23	31	156	163	56	198	69	64	48
Avg. static length	8.41	13.00	7.47	14.70	17.65	11.38	17.04	15.27	11.03
DDT executions (M)	45.38	51.75	29.85	46.94	21.46	49.45	2.19	9.17	12.52
Avg. dynamic length	7.58	11.71	5.91	13.64	17.37	9.91	19.10	12.27	18.96
Avg. dynamic unrolling	0.03	1.70	0.75	0.01	0.05	1.06	0.00	0.53	0.78
Avg. dynamic PII/DDT	1.21	1.98	1.68	1.49	1.35	2.42	1.26	2.12	3.03
Avg. dynamic PDI/DDT	**0.28**	**1.20**	**0.25**	**0.56**	**0.55**	**0.53**	**0.74**	**0.44**	**0.68**
Avg. dynamic advantage/DDT	**1.22**	**23.92**	**2.14**	**3.00**	**4.90**	**3.43**	**8.82**	**2.85**	**18.10**
Avg. dynamic advantage/PDI	4.41	19.95	8.39	5.31	8.84	6.47	11.87	6.54	26.62

ing are needed to tolerate these latencies fully. For the parallel latencies, induction unrolling suffices. However, our selection algorithm limits the unrolling degree to 4 and the maximum DDT length to 32. For the serial latencies, unoverlapped full unrolling is needed. As we explained in Chapter 2 and again in Chapter 3, our DDT selection algorithm has a difficult time recognizing instances of unoverlapped induction unrolling due to its limited slicing scope and statistical nature. Consequently, while the selection algorithm find DDTs to cover nearly 70% of *mcf*'s PDIs, full latency tolerance is projected for only 2%.

Table 5.14 provides some additional data about the automatically selected "combination" (i.e., both load and branch) DDTs. For static DDT measures, we list the total number of final (merged) DDTs and the number of single-PDI DDTs included in the merged total. As these diagnostics show, DDT overlaps are commonplace. In the *em3d* micro-benchmark, for instance, the 16 single-PDI DDTs found by the second phase of DDT selection are merged into 2 DDTs by the final phase. We also show the average static DDT size; the maximum DDT size is 32 instructions.

For dynamic measures, we present the number of *projected* DDT executions and the average dynamic length and degree of unrolling. The average number of PIIs pre-executed and PDIs covered per dynamic DDT is also shown. The projected number of PDIs covered per DDT pre-executed is an important predictive measure of DDMT performance. While the Olden micro-benchmarks average more than 3 PDIs covered per DDT, the SPEC2000 benchmarks average less than 1. We do not expect significant speedups for *gap* and *gzip*, which cover 1 PDI per 5 DDTs pre-executed. However, we do expect performance improvement on *mcf*, whose DDTs project to cover a PDI per pre-execution. The final dynamic measures shown in the table are the average projected advantage in cycles per DDT pre-executed and per PDI covered. Advantage, recall, is latency tolerance minus overhead. Average advantage per DDT is also a performance predictor. Again, the Olden micro-benchmarks project better DDMT performance than the SPEC2000 benchmarks. Of the SPEC2000 benchmarks, *mcf* again promises to benefit the most with a 22 cycle projected reduction in execution time per DDT executed.

Figure 5.4 DDMT performance on central configuration.



5.2.5 DDMT Performance

In this section we examine the performance of the runtime component of DDMT using the DDTs selected in the previous section. This portion of the evaluation is certainly not a limit study, at least if one disregards the *limit* relationship between DDMT setup and runtime.

Figure 5.4 shows DDMT's performance overlaid on our accumulating graph which earlier emphasized the performance potential of perfecting all loads and branches, perfecting PIs and perfecting DDTs. Table 5.15 and Table 5.16 provide more detailed data—Table 5.15 details combination load-branch DDTs, while Table 5.16 deals with load DDTs and branch DDTs separately. Each table shows data from two experiments, *RI* and *DDMT*. *RI* is the baseline register integration

Table 5.15 DDMT performance for "combination" load and branch DDTs.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI	TIs renamed (M)	117.89	101.21	6930.62	3894.45	244.14	1292.70	392.01	1233.65	877.51
	TIs executed (M)	103.04	100.48	6282.10	2840.59	213.68	1013.90	325.73	975.69	602.48
	BMR lat. (c)	3.86	404.81	12.78	9.41	10.51	9.50	9.76	21.07	13.86
	Load lat. (c)	21.17	54.73	5.64	2.84	2.96	2.87	2.92	4.59	3.56
	IPC	**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
DDMT	TIs renamed (M)	162.03	148.35	7062.58	3838.57	251.62	1278.75	399.73	1378.86	893.35
	TIs executed (M)	117.01	101.70	6283.74	2852.15	217.96	1025.10	335.15	1046.42	608.62
	BMR lat. (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load lat. (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI	TIs renamed (M)	7993.90	1625.76	2012.33	6399.18	4454.68	2293.74	1847.96	529.65	1662.21
	TIs executed (M)	6052.98	1126.67	1498.72	4056.57	2904.26	1507.00	1744.24	341.40	1240.72
	BMR lat. (c)	17.25	53.94	20.12	11.81	14.25	11.19	12.37	10.34	34.82
	Load lat. (c)	3.10	21.82	4.04	3.45	2.83	3.62	3.20	2.86	4.20
	IPC	**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
DDMT	TIs renamed (M)	8300.92	2198.87	2136.36	6913.57	5023.78	2450.58	1867.48	559.91	1851.15
	TIs executed (M)	6184.76	1256.79	1560.48	4319.80	3048.72	1653.96	1752.55	377.10	1290.43
	BMR lat. (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load lat. (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**

enabled superscalar configuration. In the *DDMT* experiment, we pre-execute the DDTs characterized in the previous section. Data in the row groups can be compared directly. We follow our conventions for reporting average latencies by including only retired instructions in those figures. The total counts of instructions renamed and executed in the DDMT experiments include instructions renamed and executed by DDTs. We do not report the count of instructions fetched. As DDTs do not consume fetch bandwidth, this measure is not informative.

Table 5.16 DDMT performance load DDTs and branch DDTs separately.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI	TIs renamed (M)	117.89	101.21	6930.62	3894.45	244.14	1292.70	392.01	1233.65	877.51
	TIs executed (M)	103.04	100.48	6282.10	2840.59	213.68	1013.90	325.73	975.69	602.48
	BMR. lat. (c)	3.86	404.81	12.78	9.41	10.51	9.50	9.76	21.07	13.86
	Load lat. (c)	21.17	54.73	5.64	2.84	2.96	2.87	2.92	4.59	3.56
	IPC	**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
Load DDMT	TIs renamed (M)	162.03	148.35	7043.58	3945.79	244.96	1297.02	392.91	1288.84	882.78
	TIs executed (M)	117.01	101.70	6271.40	2849.33	213.66	1013.91	325.61	994.28	595.16
	BMR lat. (c)	3.54	180.54	11.81	9.24	10.44	9.46	9.74	19.71	13.43
	Load lat. (c)	12.30	23.40	5.52	2.81	2.95	2.86	2.91	4.33	3.42
	IPC	**1.83**	**0.57**	**4.00**	**3.56**	**3.48**	**2.94**	**3.36**	**2.06**	**2.46**
	Speedup (%)	**42.99**	**93.83**	**2.14**	**0.03**	**0.07**	**0.03**	**-0.02**	**5.81**	**1.72**
Branch DDMT	TIs renamed (M)	117.89	101.21	6958.21	3783.19	251.01	1274.92	398.89	1336.17	888.15
	TIs executed (M)	103.04	100.48	6269.38	2820.84	217.71	1023.55	334.88	1029.95	605.99
	BMR lat. (c)	3.86	404.81	12.43	7.53	8.38	6.52	7.34	19.58	12.51
	Load lat. (c)	21.17	54.73	5.63	2.82	2.87	2.70	2.79	4.47	3.48
	IPC	**1.28**	**0.29**	**3.92**	**3.71**	**3.57**	**3.18**	**3.50**	**2.00**	**2.49**
	Speedup (%)	**0.00**	**0.00**	**0.25**	**4.22**	**2.48**	**8.19**	**4.34**	**2.44**	**3.06**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI	TIs renamed (M)	7993.90	1625.76	2012.33	6399.18	4454.68	2293.74	1847.96	529.65	1662.21
	TIs executed (M)	6052.98	1126.67	1498.72	4056.57	2904.26	1507.00	1744.24	341.40	1240.72
	BMR. lat. (c)	17.25	53.94	20.12	11.81	14.25	11.19	12.37	10.34	34.82
	Load lat. (c)	3.10	21.82	4.04	3.45	2.83	3.62	3.20	2.86	4.20
	IPC	**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Load DDMT	TIs renamed (M)	8003.97	2160.95	2087.72	6633.64	4604.80	2399.67	1880.83	553.61	1849.87
	TIs executed (M)	5961.85	1228.66	1518.78	4103.27	2907.05	1522.23	1754.18	360.94	1287.72
	BMR lat. (c)	16.60	40.53	18.92	11.66	14.22	9.98	12.31	7.75	35.91
	Load lat. (c)	3.09	17.19	3.85	3.42	2.82	3.09	3.15	2.65	3.78
	IPC	**2.78**	**0.72**	**1.86**	**2.60**	**2.78**	**2.38**	**4.63**	**2.93**	**1.83**
	Speedup (%)	**1.54**	**15.33**	**2.19**	**0.08**	**0.17**	**7.99**	**2.20**	**8.37**	**9.18**
Branch DDMT	TIs renamed (M)	8321.94	1797.69	2076.10	6845.82	5021.70	2392.94	1840.07	539.60	1820.38
	TIs executed (M)	6274.07	1123.63	1525.65	4299.09	3046.86	1644.78	1742.73	363.91	1282.52
	BMR lat. (c)	16.61	50.42	18.88	10.47	13.32	8.33	10.10	6.62	34.68
	Load lat. (c)	3.08	19.52	3.93	3.39	2.77	3.28	3.18	2.64	3.89
	IPC	**2.76**	**0.64**	**1.86**	**2.65**	**2.80**	**2.40**	**4.58**	**3.02**	**1.83**
	Speedup (%)	**0.80**	**2.51**	**2.14**	**2.05**	**0.93**	**8.86**	**0.97**	**11.58**	**8.82**

We discuss the Olden micro-benchmarks before turning to the SPEC2000 programs. Both *em3d* and *mst* are bound by memory latency as their initial average load latency figures of 21 and 54 cycles suggest. DDMT attacks this memory latency directly by pre-executing the problem loads achieving a prefetching effect. In both cases, DDMT is successful in reducing the average load latency by almost 50%—to 12 and 23 cycles, respectively. In *mst*, where these latencies are taken in series, this nearly 50% reduction in average load latency translates directly into a nearly 50% reduction in execution time—i.e., a 93% speedup. In *em3d*, many of the L2 cache misses are parallel, meaning that some level of latency overlapping is naturally present in the program. In this case, a nearly 50% reduction in average load latency translates "only" into a 43% speedup (a 31% reduction in execution time).

DDMT's effectiveness on these two micro-benchmarks is in part due to their low initial performance: IPCs of 1.28 and 0.29 respectively. Such low baseline levels of processor utilization justify and can tolerate the siphoning of large amounts of sequencing and execution bandwidth to DDTs. In both cases, DDTs account for at least a 50% increase in sequencing bandwidth consumption, with that figure at close to 80% for *em3d*. However, the increase in execution bandwidth consumption is much less—only 25% and 13%, respectively. This, of course, is due to pre-execution reuse via register integration. We will take a closer look at pre-execution and pre-execution reuse in the following section.

One factor not explicitly shown in the table that contributes to DDMT's excellent performance on both micro-benchmarks is that their dominant loops perform relatively few stores. This translates into a small number of load mis-integrations. We will investigate the performance impact of mis-integrations in Section 5.3.1.2.

DDMT's performance impact on the SPEC2000 benchmarks is both less pronounced and more varied. Performance improves by 8% to 15% over the RI case for six benchmarks: *eon.k, gap, mcf, twolf*, *vpr.p*, and *vpr.r.* The remaining 10 benchmarks experience speedups of between 1% and 4%. It is probably not coincidental that four of the runs that benefit the most are *mcf, twolf*, *vpr.p* and *vpr.r*. *Mcf, twolf* and *vpr* make heavy use of pointer-based data structures, and use execution idioms similar to those used in the two Olden micro-benchmarks.

Of the six high-performing benchmarks, *eon.k* achieves its speedup almost exclusively via the pre-execution of branches while *mcf* benefits primarily from the pre-execution of loads. Like *mst*, most of *mcf*'s cache misses are serial, accounting both for its low initial performance and the fact that a 20% reduction in average load latency produced by DDMT translates into a 15% performance improvement. The rest of the benchmarks achieve some benefit from both. Notice, speedups generated by pre-executing branch DDTs and speedups generated by pre-executing load DDTs *are not additive* (or multiplicative). In *vpr.r*, for instance, pre-executing only loads achieves a speedup of 9% while pre-executing only branches achieves an 8% speedup. However, pre-executing both achieves only an 11% speedup. The reason for this phenomenon is the overlap of load and branch PDI computations, specifically the frequent idiom of a mis-predicted branch depending on a cache miss. Notice, pre-executing branch computations in *vpr.r* primarily lowers the branch mis-prediction resolution (BMR), but also reduces the load latency as many of the mis-predicted branches have cache missing loads in their computations. Similarly, pre-executing load computations in *vpr.r* primarily lowers the average load latency, but also effects the BMR latency as mis-predicted branches that depend on pre-executed loads are resolved earlier.

The lower speedups observed on the remaining benchmarks are due to several factors, most of which will be discussed in 5.2.6. Here, we are concerned with factors that produce a DDMT performance shortfall with respect to the performance predicted by our DDT selection algorithm and by the perfect DDT experiments.

Ignoring performance for a moment, DDMT does what it is meant to do, reducing *both* the average load latency and the average mis-prediction resolution (BMR) latency in every benchmark. Reduction in average load latency varies from a fraction of 1% to over 20%. Branch mis-prediction resolution latency is reduced by up to 40%. In Section 5.2.5.3, we use register integration to break down performance improvements into prefetching effects, branch resolution effects and reuse effects.

DDMT achieves high speedups on the two Olden micro-benchmarks because those programs are composed almost entirely of PDIs and their computations. As we have seen, DDMT sequences almost a full copy of the master thread as the sum of many DDTs. Performance improves because this copy is effectively time shifted with respect to the master thread. In the SPEC2000 benchmarks, PDIs and their computations account for a much smaller fraction of the dynamic instruction stream. DDMT typically results in a 3% to 10% increase in the total number of instructions renamed. However, due to pre-execution reuse via register integration, the consumption of execution bandwidth increases only by 1% to 6%.

One apparent anomaly we should explain is the case of pre-executing load DDTs for *vpr.p*, where performance achieved is higher than perfect performance. This is effect is due to register integration. In our model, a perfect load completes in 3 cycles. However, from the point of view of the master thread, a load integrated after having completed execution effectively completes in zero cycles. With low overhead and a sufficiently high integration rate, it is conceivable that the master thread's average load latency could drop to well below its perfect value of 3. In this experiment, it drops to 2.65.

5.2.5.1 Measuring Pre-Execution Activity and Pre-Execution Reuse

The previous section presented *total-system diagnostics* that can be directly compared with non-DDMT configurations. In this section, we present DDMT-specific metrics that allow us to measure the "level" of pre-execution activity going on in the processor and the degree to which pre-executed results are integrated by the master thread.

Table 5.17 presents pre-execution specific metrics for the three DDMT experiments— *loads*, *branches* and *all* (both loads and branches together). We discuss the measures of the *all* experiment. The other two result sets have the same characteristics. We report the number of *DDTs forked* by the master thread, and the number of *DDTs squashed*. DDT squashes occur due to path differences between the DDT and the master thread and resulting invalidation of physical registers or due to exceptions in DDT instructions. A DDT squash does not mean that all instructions in the DDT are lost; only instructions that occur after the disastrous event are aborted. The bottom row group in each experiment shows per-*data-driven instructions (DDI)* metrics. We show the numbers of *DDIs renamed*, *DDIs executed*, and *DDI results integrated*. Again, we count an integration event when the integrating master thread instruction retires to avoid double counting and counting integrations by mis-speculated instructions. The *DDI integration rate* is computed by dividing the number of DDT results integrated by the number of DDT instructions renamed. We comment briefly about a few of these measures.

DDT squashes are relatively rare. In general, about 10% of DDTs are squashed—or rather partially squashed—with the bulk of the squashes caused by physical register invalidations. It may be possible to reduce the overhead of squashed DDTs by dynamically learning which DDTs are susceptible to squashes and selectively suppressing the fork of these DDTs. We have not experimented with such a mechanism.

The number of DDT instructions renamed is often much larger than the number of DDT instructions executed, often exceeding the latter by a factor of 2. There are two effects at work here. First, un-executed DDT instructions may be aborted via cascaded invalidations even after the DDT has completed sequencing and vacated its register context. Second, a DDT instruction which is integrated before it has had a chance to execute and which is subsequently executed by the master thread is not counted as a DDT executed instruction.

Table 5.17 Pre-execution and pre-execution reuse diagnostics.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Load	DDTs forked (M)	3.55	2.38	13.08	6.21	0.16	0.98	0.19	4.33	1.81
	DDTs squashed (M)	0.09	0.00	2.08	1.09	0.01	0.04	0.00	0.15	0.19
	DDIs renamed (M)	42.73	47.22	150.57	73.16	0.83	4.74	0.87	47.41	12.36
	DDIs executed (M)	35.75	31.36	84.27	53.25	0.64	3.20	0.58	23.55	6.62
	DDIs integrated (M)	25.05	34.29	50.96	26.58	0.51	2.66	0.42	9.78	1.91
	DDI integ. rate (%)	**58.62**	**72.62**	**33.85**	**36.34**	**60.65**	**55.99**	**48.32**	**20.62**	**15.41**
Branch	DDTs forked (M)	0.00	0.00	4.11	24.39	1.73	12.87	3.66	16.53	7.29
	DDTs squashed (M)	0.00	0.00	0.30	3.44	0.12	1.19	0.31	1.54	1.53
	DDIs renamed (M)	0.00	0.00	50.52	187.48	14.57	104.49	30.78	141.59	68.83
	DDIs executed (M)	0.00	0.00	23.45	132.41	10.37	80.94	23.00	91.38	39.43
	DDIs integrated (M)	0.00	0.00	11.07	86.05	5.07	41.73	10.38	48.30	13.35
	DDI integ. rate (%)	**nan**	**nan**	**21.91**	**45.90**	**34.84**	**39.93**	**33.73**	**34.11**	**19.40**
All	DDTs forked (M)	3.55	2.38	15.58	29.46	1.86	13.82	3.85	19.84	8.20
	DDTs squashed (M)	0.09	0.00	2.64	4.26	0.12	1.19	0.32	1.68	1.64
	DDIs renamed (M)	42.73	47.22	184.74	249.73	15.29	109.06	31.66	186.31	76.47
	DDIs executed (M)	35.75	31.36	103.44	180.69	10.95	84.52	23.60	111.28	43.27
	DDIs integrated (M)	25.05	34.29	57.59	109.18	5.48	44.28	10.83	58.20	14.45
	DDI integ. rate (%)	**58.62**	**72.62**	**31.17**	**43.72**	**35.82**	**40.60**	**34.20**	**31.24**	**18.90**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Load	DDTs forked (M)	8.93	57.17	24.87	25.51	0.59	29.62	1.96	3.92	6.77
	DDTs squashed (M)	0.78	2.62	1.41	1.77	0.10	2.77	0.07	0.63	0.39
	DDIs renamed (M)	83.96	646.72	122.65	177.90	9.13	170.74	35.47	57.40	150.42
	DDIs executed (M)	20.04	239.66	67.57	81.38	7.10	84.04	27.02	46.16	82.27
	DDIs integrated (M)	38.32	107.84	24.66	17.93	4.12	50.15	20.91	22.12	42.31
	DDI integ. rate (%)	**45.64**	**16.68**	**20.10**	**10.08**	**45.12**	**29.37**	**58.96**	**38.54**	**28.13**
Branch	DDTs forked (M)	51.00	27.33	24.53	62.38	37.05	55.44	1.06	8.02	12.22
	DDTs squashed (M)	5.63	2.10	2.52	10.16	5.70	5.19	0.08	0.60	0.35
	DDIs renamed (M)	423.16	286.93	134.89	818.45	579.69	429.95	18.38	84.97	198.27
	DDIs executed (M)	329.95	151.01	82.36	415.44	248.52	303.32	15.27	67.08	115.79
	DDIs integrated (M)	71.56	83.14	31.13	140.05	101.83	94.36	13.14	32.24	65.21
	DDI integ. rate (%)	**16.91**	**28.97**	**23.08**	**17.11**	**17.57**	**21.95**	**71.51**	**37.94**	**32.89**
All	DDTs forked (M)	60.15	66.46	41.53	74.75	37.51	56.09	2.57	9.95	12.97
	DDTs squashed (M)	7.25	4.18	3.31	10.98	5.78	7.28	0.15	0.75	0.33
	DDIs renamed (M)	493.08	740.37	223.97	894.94	586.71	499.00	49.51	115.33	239.12
	DDIs executed (M)	305.73	294.97	135.34	444.81	253.76	333.52	38.26	89.17	141.38
	DDIs integrated (M)	135.39	131.26	48.37	147.48	104.95	128.93	29.74	42.65	82.24
	DDI integ. rate (%)	**27.46**	**17.73**	**21.60**	**16.48**	**17.89**	**25.84**	**60.06**	**36.98**	**34.39**

The number of instructions integrated and the *integration rate*—the ratio of DDT instructions integrated to DDT instructions renamed—are measures of two things. First, they are a crude measure of DDT "accuracy", the fraction of DDT instructions that pre-executed the right thing. The measure is crude because integration may fail for any number of reasons. Admittedly, artificial integration failures—i.e., ones that could have, and perhaps should have, been avoided—are rare. Second, they are a more accurate measure of the DDMT's execution "non-overhead". Integrated instructions are executed only once by the out-of-order execution engine. Integrated loads are re-executed, but this functionality belongs to the retirement stage. DDT instruction integration rates are about 58% and 70% for the Olden micro-benchmarks *em3d* and *mst*, respectively. They vary between 15% and 40% for most of the SPEC2000 benchmarks. The fact that integration rates are not close to 100% is due to a combination of three factors. The first is a pathological interaction with the integration mechanism itself. We will investigate this effect in more detail in our sensitivity analysis (Section 5.3.1.1). The second factor is the presence of induction unrolling which involves sequencing what is essentially the same induction sequence instance multiple times. This is likely a small effect. The third factor is the fact that the DDT selection algorithm chooses DDTs whose triggers are not control-equivalent with the rest of the computation. In other words, it selects a DDT with the *a priori* "understanding" that some fraction of its forked instances will execute along an implicit control path that the master thread will not follow. The DDT selection algorithm knows the expected ratio of DDTs that will be forked along non-master thread paths, and takes that ratio into account when calculating aggregate overhead (OH_{agg}) and hence aggregate advantage (ADV_{agg}). There is not much we can do to explicitly characterize this last factor which is likely responsible for the integration rate differences observed between the micro-benchmarks and the more realistic SPEC2000 benchmarks. *Em3d*'s and *mst*'s DDTs are both induction unrolled on an outer loop; they are dynamically control-equivalent with their trigger instructions a large fraction of the time. The integration shortfall observed in these benchmarks is due to individual inner loop instructions that are not executed by the master thread. Rarely is an entire DDT not integrated due to path divergence. In contrast, the SPEC2000 benchmarks contain more varied control flow, and the DDTs chosen for them are dynamically control equivalent with their trigger instructions a lower fraction of the time.

5.2.5.2 PDI Coverage

Ultimately, the function of DDTs is to pre-execute the PDIs at the ends of their respective computations, not the intermediate results. In this section, we present PDI specific metrics. We then use our measurements to evaluate DDMT's PII and PDI "coverages" in terms of the upper bounds set by the selected DDTs. In other words, just as we evaluated the PI definition against an ideal processor and the DDT selection algorithm against an ideal "implementation" of the PI definition, we now evaluate DDMT against an ideal "implementation" of the selected DDTs. Again, we show the results of the three experiments— *loads*, *branches* and *all*. These are shown in three tables, Table 5.18, Table 5.19 and Table 5.20,

Table 5.18 Load pre-execution and pre-execution reuse diagnostics.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
DDT	PIIs executed (M)	19.06	12.34	16.67	7.76	0.15	0.94	0.19	4.75	3.81
	PDIs covered (M)	9.65	11.03	9.02	1.31	0.05	0.14	0.05	0.98	0.40
	PDIs fully covered (M)	5.11	8.59	1.78	1.05	0.03	0.07	0.03	0.20	0.16
DDMT	PIIs executed (M)	19.00	9.57	14.02	5.45	0.15	0.83	0.16	3.06	2.10
	PDIs executed (M)	8.76	7.52	6.09	1.52	0.05	0.12	0.03	0.61	0.28
	PDIs integrated (M)	9.08	9.32	4.89	0.78	0.03	0.06	0.01	0.99	0.17
	PDIs covered (M)	7.45	6.95	4.05	0.44	0.03	0.05	0.00	0.26	0.11
	PDIs fully covered (M)	4.55	5.85	2.78	0.27	0.03	0.03	0.00	0.05	0.07
	PII coverage (%)	**99.70**	**77.51**	**84.11**	**70.26**	**96.15**	**88.52**	**83.14**	**64.43**	**55.15**
	PDI coverage (%)	**77.18**	**63.03**	**44.95**	**33.63**	**49.80**	**36.36**	**9.08**	**26.68**	**27.77**
	PDI full coverage (%)	**47.13**	**53.01**	**30.85**	**20.56**	**49.80**	**24.46**	**1.75**	**4.69**	**17.33**
	Full coverage realized (%)	**89.07**	**68.11**	**156.34**	**25.63**	**99.60**	**45.46**	**3.46**	**23.10**	**42.60**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
DDT	PIIs executed (M)	16.03	89.61	28.35	13.76	0.37	48.01	1.65	8.70	23.28
	PDIs covered (M)	6.77	60.41	5.66	2.28	0.12	15.94	1.03	1.94	6.03
	PDIs fully covered (M)	0.00	0.66	1.52	1.24	0.12	9.76	0.47	1.65	5.61
DDMT	PIIs executed (M)	3.94	78.26	24.65	9.31	0.37	30.23	1.34	8.34	11.98
	PDIs executed (M)	1.33	24.88	3.51	1.36	0.11	9.12	0.88	1.96	4.42
	PDIs integrated (M)	3.69	33.46	3.38	0.29	0.04	8.77	0.65	1.12	2.69
	PDIs covered (M)	1.21	13.82	1.77	0.16	0.04	5.39	0.64	0.95	2.57
	PDIs fully covered (M)	0.53	9.19	0.98	0.06	0.04	5.01	0.43	0.82	2.35
	PII coverage (%)	**24.57**	**87.33**	**86.95**	**67.67**	**99.01**	**62.96**	**80.92**	**95.83**	**51.48**
	PDI coverage (%)	**17.90**	**22.88**	**31.28**	**6.91**	**34.92**	**33.84**	**61.85**	**48.72**	**42.71**
	PDI full coverage (%)	**7.82**	**15.21**	**17.27**	**2.69**	**33.36**	**31.40**	**41.57**	**42.31**	**39.01**
	Full coverage realized (%)	**nan**	**1385.4**	**64.12**	**4.96**	**33.36**	**51.28**	**91.03**	**49.95**	**41.93**

respectively.

The first experiment in each table—*DDT*—shows the numbers of *PIIs executed*, *PDIs covered* and *PDIs fully covered* as projected by the DDT selection algorithm. The second experiment shown in the table—*DDMT*—measures the corresponding quantities from the DDMT performance simulation. Coverages are shown in bold as the bottom row group of each DDMT experiment. We clarify how each coverage is computed. *PDI coverage* is computed by dividing DDMT's *PDIs covered* by DDT's *PDIs covered*. *PDI coverage* measures how many of the targeted PDIs were actually covered by DDMT. *PDI full coverage* is computed by dividing DDMT's *PDIs fully covered* by DDT's *PDIs covered*. *PDI full coverage* measures the fraction of the targeted PDIs that were fully covered in practice. Finally, *full coverage realized* is computed by dividing DDMT's *PDIs fully covered* by DDT's *PDIs fully covered*. *Full coverage realized* measures the ratio of PDIs for which full coverage was achieved to that for which full coverage was projected. It is distinctly possible for the DDT selection algorithm to underestimate latency tolerance of a certain DDT, resulting in a full coverage realization of greater than 100%.

Our DDMT measure of *PIIs executed* is straightforward. However, our definition of *PDIs covered* and *PDIs fully covered* in the DDMT experiment, as well as our reason for listing *PDIs executed* and *PDIs integrated* measures, merit further explanation. This is true especially because these measures have slightly different interpretations when applied to load and branch PDIs.

In the load case, it is not immediately clear which DDMT quantity properly corresponds to the *PDIs covered* metric of the DDT upper bound. Our definition for covering a PDI load is to pre-execute it such that its master thread execution will appear to hit in the cache. However, if the load is integrated—as we hope it will be—there will be no master thread execution whose latency we can measure! On the other hand, if the load is *not* integrated then we have no way of associating the pre-executed instance with the master thread instance for which it is intended. There are similar problems in calculating full vs. partial latency coverage for pre-executed loads. We cannot know how much execution latency of the master thread instance is untolerated if the master thread instance is not re-executed. To deal with this dilemma, we list four metrics, none of which measures precisely the information we want, but which collectively convey the general idea. *PDIs executed* is the number of cache misses triggered by load pre-executed PIIs. This count excludes cache misses due to pre-executed loads whose execution took place after integration as no latency tolerance is achieved for these loads. However, PDIs executed may be an over-count as—if the PDI is not subsequently integrated—the prefetched block may never be used by any master thread instruction. *PDIs integrated* is a measure of all retired loads that were initially allocated in DDTs and triggered a cache miss. *PDIs integrated* is an overcount. Although it cannot include "wrong prefetches", it does include cache misses issued post integration, by the master thread. We show *PDIs integrated* because our strict definition of does not require that the PDI was issued prior to integration, only that its latency would have been equivalent to that of a cache hit had the load been executed by master thread. If the load's computation was integrated in the completed state, this level of latency tolerance may be achieved. The final two measures correlate with PDI coverage more intuitively, if not more accurately in the strict sense. Load *PDIs covered* is the number of load PDIs integrated after having been issued by the DDT. Load *PDIs fully covered* is the number of load PDIs integrated after having been issued by the DDT and completed. These, too, are both over- and under-counts. They are over-counts because

Table 5.19 Branch pre-execution and pre-execution reuse diagnostics.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
DDT	PIIs executed (M)	0.00	0.00	3.89	27.47	1.82	15.89	4.77	14.78	6.13
	PDIs covered (M)	0.00	0.00	0.85	4.88	0.32	3.56	0.79	2.24	1.40
	PDIs fully covered (M)	0.00	0.00	0.52	4.17	0.23	2.77	0.64	1.47	0.66
DDMT	PIIs executed (M)	0.00	0.00	0.60	17.43	1.44	12.27	3.89	6.96	2.92
	PDIs executed (M)	?	?	?	?	?	?	?	?	?
	PDIs integrated (M)	0.00	0.00	0.22	3.62	0.19	2.78	0.47	1.35	0.60
	PDIs covered (M)	0.00	0.00	0.16	2.63	0.12	2.23	0.34	0.87	0.42
	PDIs fully covered (M)	0.00	0.00	0.15	2.40	0.12	2.17	0.32	0.84	0.42
	PII coverage (%)	**nan**	**nan**	**15.40**	**63.44**	**79.02**	**77.25**	**81.53**	**47.09**	**47.69**
	PDI coverage (%)	**nan**	**nan**	**18.32**	**53.80**	**36.53**	**62.58**	**43.24**	**38.83**	**30.00**
	PDI full coverage (%)	**nan**	**nan**	**17.95**	**49.28**	**36.38**	**60.86**	**41.08**	**37.40**	**29.65**
	Full coverage realized (%)	**nan**	**nan**	**29.39**	**57.70**	**50.12**	**78.29**	**50.94**	**57.28**	**63.37**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
DDT	PIIs executed (M)	38.95	19.25	22.43	56.77	28.55	71.53	1.09	10.72	14.64
	PDIs covered (M)	5.82	2.37	2.01	24.29	11.77	9.60	0.59	2.06	2.50
	PDIs fully covered (M)	1.54	0.40	0.67	10.89	4.44	4.79	0.55	1.33	2.26
DDMT	PIIs executed (M)	19.76	6.72	13.25	27.33	9.60	38.25	0.62	7.14	11.49
	PDIs executed (M)	?	?	?	?	?	?	?	?	?
	PDIs integrated (M)	3.05	1.37	0.98	7.06	2.15	5.16	0.50	1.40	1.54
	PDIs covered (M)	1.03	0.60	0.53	2.64	1.27	4.31	0.27	0.97	1.32
	PDIs fully covered (M)	0.97	0.60	0.51	2.63	1.27	4.28	0.27	0.95	1.32
	PII coverage (%)	**50.72**	**34.91**	**59.08**	**48.15**	**33.63**	**53.47**	**56.41**	**66.60**	**78.46**
	PDI coverage (%)	**17.66**	**25.42**	**26.13**	**10.86**	**10.80**	**44.91**	**46.58**	**47.26**	**52.79**
	PDI full coverage (%)	**16.73**	**25.28**	**25.33**	**10.83**	**10.78**	**44.60**	**46.39**	**46.34**	**52.76**
	Full coverage realized (%)	**63.43**	**148.23**	**75.63**	**24.16**	**28.58**	**89.38**	**49.54**	**71.81**	**58.34**

integration may fail even though the proper prefetching effect was achieved. They are undercounts because pre-integration completion was not an initial requirement for load latency coverage, nor do we select DDTs with the express purpose of providing the appearance of pre-integration completion.

The definitions of these measures for branches (Table 5.19) are somewhat less muddled. First, note that in this experiment the values of *PDIs executed* are replaced with question marks. The reason for this is that there is no way to know

Table 5.20 Load and branch pre-execution and pre-execution reuse diagnostics.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
DDT	PIIs executed (M)	19.06	12.34	20.56	35.23	1.97	16.83	4.96	19.53	10.27
	PDIs covered (M)	9.65	11.03	9.87	6.19	0.37	3.69	0.84	3.23	1.79
	PDIs fully covered (M)	5.11	8.59	2.30	5.22	0.26	2.84	0.66	1.67	0.82
DDMT	PIIs executed (M)	19.00	9.57	15.14	23.53	1.61	13.14	4.06	10.22	5.29
	PDIs executed (M)	8.76	7.52	6.08	2.14	0.05	0.12	0.04	0.62	0.30
	PDIs integrated (M)	9.08	9.32	5.23	4.41	0.21	2.85	0.50	2.36	0.75
	PDIs covered (M)	7.45	6.95	4.33	3.16	0.14	2.29	0.36	1.14	0.53
	PDIs fully covered (M)	4.55	5.85	2.84	0.30	0.02	0.03	0.01	0.05	0.07
	PII coverage (%)	**99.70**	**77.51**	**73.67**	**66.78**	**81.49**	**78.05**	**81.85**	**52.34**	**51.49**
	PDI coverage (%)	**77.18**	**63.03**	**43.84**	**51.02**	**38.34**	**61.93**	**43.04**	**35.38**	**29.44**
	PDI full coverage (%)	**47.13**	**53.01**	**30.32**	**44.33**	**38.19**	**59.80**	**39.46**	**27.51**	**26.80**
	Full coverage realized (%)	**89.07**	**68.11**	**130.07**	**52.58**	**54.91**	**77.80**	**50.09**	**53.34**	**58.79**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
DDT	PIIs executed (M)	54.98	102.52	50.23	70.05	28.92	119.55	2.75	19.42	37.88
	PDIs covered (M)	12.59	62.06	7.61	26.50	11.88	26.20	1.62	4.00	8.51
	PDIs fully covered (M)	1.54	1.07	2.14	12.06	4.56	11.66	1.02	2.97	7.85
DDMT	PIIs executed (M)	29.29	84.74	39.89	37.12	9.97	72.14	1.96	15.46	24.28
	PDIs executed (M)	3.97	24.20	3.65	1.38	0.11	9.68	0.88	1.91	4.62
	PDIs integrated (M)	6.89	33.12	3.83	6.98	2.19	14.15	1.15	2.01	4.57
	PDIs covered (M)	2.25	13.19	2.01	2.71	1.31	9.98	0.92	1.46	4.13
	PDIs fully covered (M)	0.54	8.41	0.72	0.05	0.04	4.95	0.43	0.35	2.32
	PII coverage (%)	**53.26**	**82.66**	**79.42**	**52.99**	**34.48**	**60.34**	**71.48**	**79.60**	**64.08**
	PDI coverage (%)	**17.88**	**21.26**	**26.39**	**10.24**	**11.04**	**38.11**	**56.42**	**36.52**	**48.47**
	PDI full coverage (%)	**11.90**	**14.17**	**16.04**	**9.81**	**11.00**	**35.01**	**43.24**	**32.95**	**45.11**
	Full coverage realized (%)	**97.59**	**823.20**	**56.98**	**21.56**	**28.70**	**78.68**	**68.56**	**44.34**	**48.89**

whether a pre-executed branch PII is a PDI or not until the result is integrated and correlated with a master thread branch. On the other hand, the *PDIs fully covered* metric—again defined as the number of pre-executed branches integrated after having been executed and completed by a DDT—measures exactly what we want. Our intent in selecting DDTs for problem branches is that those DDTs will allow the branch to complete prior to integration, thus taking full advantage of instantaneous mis-prediction resolution.

For consistency, we include *PDIs integrated* and *PDIs covered* metrics. These use the same definitions as were used in the load case. *PDIs integrated* is the count of mis-predicted branches integrated at any point during their lifetime. PDIs integrated has the same significance for branches as it does for loads. Integrating an un-executed branch will effectively lower its resolution latency if, by the time it is integrated, the majority of its computation has completed. *PDIs covered* is the count of mis-predicted branches integrated after having issued in the DDT. Notice the almost precise match between *PDIs covered* and *PDIs fully covered* for branches. Since branches execute in one cycle, this difference measures those branches that are executed by the out-of-order engine in the precise cycle in which they are integrated.

As the tables show, load and branch *PII coverages* are high while *PDI coverages* and *PDI full coverages* are significantly lower. However, this does not mean that DDMT is a low-efficiency implementation of pre-execution. Recall, *PDI full coverage* is artificially low for loads and *PDI coverage* is artificially low for both loads and branches, as it defines coverage too tightly. If we redefine *PDI coverage* to use *PDIs integrated* as the numerator rather than *PDIs covered*, coverages will double in many cases. Of course, this is not an ideal solution either as *PDIs integrated* defines coverage too loosely. For loads especially, any definition of coverage that includes integration as a criterion does not fully measure coverage. Performance benefit for load PDIs can be conveyed via the prefetching effect and without integration at all. If we mentally correct for these inaccuracies, load and overall coverage rise into the moderate range.

One coverage measure we can use directly with relatively high confidence is *PDI full coverage* for branches, since it matches our definition of latency coverage precisely. Happily, branch *PDI full coverage* is relatively high, in the 20% to 50% range for many of the benchmarks.

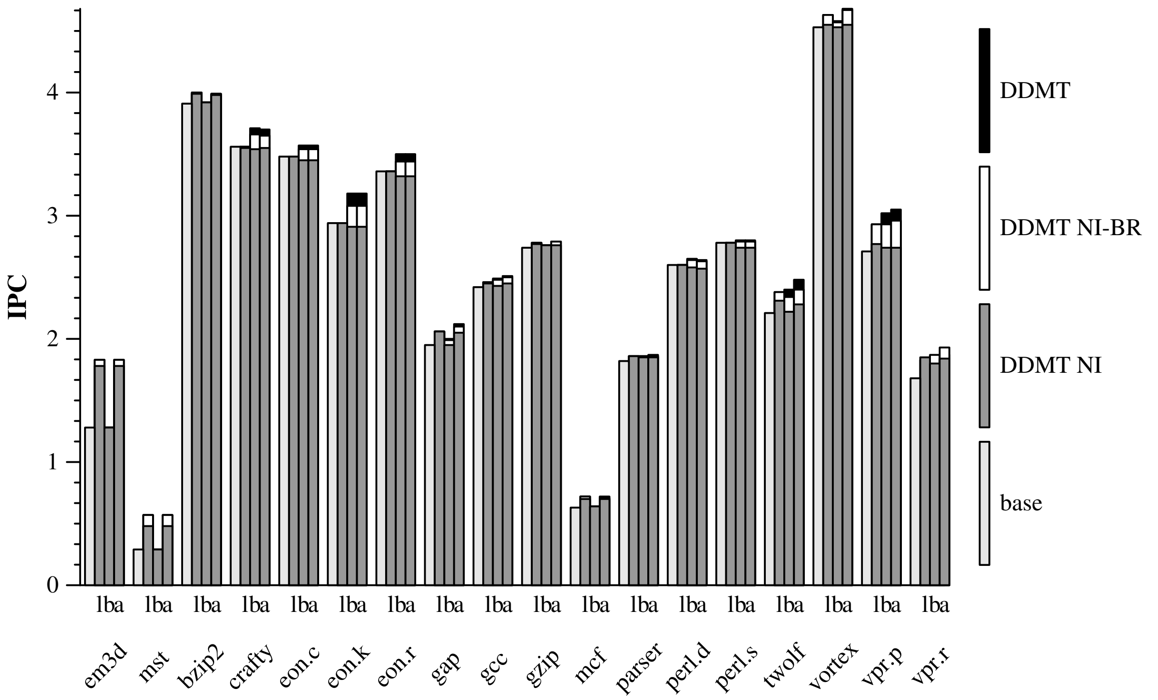
As we initially hinted, the possibility exists for the DDT selection algorithm to underestimate the latency tolerance of a DDT—this is usually due to an under-estimation in the execution time of the master thread—and project a lower incidence of full coverage than is achieved in practice. We observe this phenomenon in *bzip2* and *mcf* in the form of full coverage realization rates of greater than 100%.

5.2.5.3 Impact of Pre-Execution Reuse

In Section 5.1.1, we measured the performance impact of squash reuse. In this section, we attempt to isolate the performance impact of pre-execution reuse on DDMT. Recall, pre-execution reuse provides DDMT with two functions. First, *it* enables the implementation of instantaneous resolution of mis-predicted pre-executed branches. Second, it passes DDT results to the master thread, reducing execution bandwidth consumption and compressing the master thread's critical path. In this section, we measure the performance impact of each of these functions. We then use this breakdown to attribute performance improvement to instantaneous branch resolution, prefetching effects and reuse.

We measure a function's performance impact by eliminating it from our configuration. We begin by measuring the performance of DDMT with all integration

Figure 5.5 Performance impact of pre-execution reuse.



transitions (as described in Section 4.4.3) enabled. We eliminate the instantaneous resolution effect by disallowing master thread instructions from integrating *D* state branches. Doing this has little effect besides eliminating instantaneous resolution, as branches have no downstream dataflow successors whose integration will be forfeited, and the cost of operationally re-executing the branch is a single cycle. We remove pre-execution reuse by disallowing the *D* to *C* state transition entirely.

The results of these experiments are shown in Figure 5.5 and Table 5.21. The format of the figure is somewhat different than the one we have been use to. Our IPC stack consists of three experiments. *DDMT* is our initial experiment with integration allowed in all directions. *DDMT NI-BR* is DDMT with branch inte-

Table 5.21 Performance impact of pre-execution reuse.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
DDMT	DDIs executed (M)	35.75	31.36	103.44	180.69	10.95	84.52	23.60	111.28	43.27
	DDIs integrated (M)	25.05	34.29	57.59	109.18	5.48	44.28	10.83	58.20	14.45
	BMR lat. (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load lat. (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
DDMT NI-BR	DDIs executed (M)	29.70	31.28	93.88	182.00	10.51	81.74	23.62	104.77	35.69
	DDIs integrated (M)	25.04	34.29	56.53	95.94	4.79	36.16	9.27	49.60	12.83
	BMR lat. (c)	3.54	181.50	11.73	7.92	8.66	7.29	7.85	18.45	12.48
	Load lat. (c)	12.31	23.40	5.52	2.81	2.86	2.69	2.78	4.22	3.39
	IPC	**1.83**	**0.57**	**3.99**	**3.65**	**3.54**	**3.08**	**3.44**	**2.10**	**2.50**
	Speedup (%)	**42.97**	**93.83**	**2.05**	**2.59**	**1.71**	**4.76**	**2.38**	**7.48**	**3.36**
DDMT NI	DDIs executed (M)	34.92	42.34	126.64	218.35	12.11	93.64	26.40	125.86	43.62
	DDIs integrated (M)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	BMR lat. (c)	3.62	181.52	12.03	9.27	10.27	9.29	9.60	19.54	13.17
	Load lat. (c)	13.73	23.17	5.54	2.87	2.96	2.88	2.92	4.33	3.43
	IPC	**1.78**	**0.48**	**3.98**	**3.55**	**3.45**	**2.91**	**3.32**	**2.05**	**2.45**
	Speedup (%)	**39.29**	**64.25**	**1.74**	**-0.24**	**-0.75**	**-0.86**	**-1.28**	**5.29**	**1.42**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
DDMT	DDIs executed (M)	305.73	294.97	135.34	444.81	253.76	333.52	38.26	89.17	141.38
	DDIs integrated (M)	135.39	131.26	48.37	147.48	104.95	128.93	29.74	42.65	82.24
	BMR lat. (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load lat. (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
DDMT NI-BR	DDIs executed (M)	324.36	251.57	127.20	435.86	220.74	312.10	36.85	90.66	176.26
	DDIs integrated (M)	113.41	127.97	42.50	135.71	87.22	104.78	28.88	38.03	113.51
	BMR lat. (c)	16.02	41.71	18.38	10.59	13.38	8.64	10.32	6.88	32.87
	Load lat. (c)	3.07	17.83	3.82	3.39	2.77	2.93	3.13	2.49	3.53
	IPC	**2.79**	**0.71**	**1.86**	**2.63**	**2.79**	**2.40**	**4.67**	**2.96**	**1.93**
	Speedup (%)	**1.69**	**12.92**	**2.17**	**1.15**	**0.61**	**8.84**	**3.00**	**9.32**	**14.54**
DDMT NI	DDIs executed (M)	448.52	403.65	152.83	464.20	238.21	369.08	32.19	113.10	138.34
	DDIs integrated (M)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	BMR lat. (c)	16.56	42.67	18.90	11.35	14.04	10.42	12.39	9.15	34.99
	Load lat. (c)	3.11	18.07	3.90	3.43	2.83	3.30	3.19	2.75	3.86
	IPC	**2.76**	**0.70**	**1.85**	**2.57**	**2.74**	**2.28**	**4.55**	**2.74**	**1.84**
	Speedup (%)	**0.74**	**12.10**	**1.50**	**-0.94**	**-1.15**	**3.23**	**0.28**	**1.19**	**9.76**

gration disabled. In the *DDMT NI* experiment, all pre-execution reuse is disabled. We show base performance alongside this IPC stack rather than underneath it to emphasize the fact that excluding pre-execution reuse can result in sub-baseline performance. As usual, the graph contains results for *load*, *branch* and *all* experiments. The table, however, only shows the results of the *all* experiment.

The results show that instantaneous branch resolution has a noticeable performance effect for those programs that pre-execute branches—*crafty*, the *eon* benchmarks, *twolf* and *vpr.p*. In *twolf*, for instance, performance improvement drops from 12% to 8% when branch integration is not included. Notice, even for benchmarks that benefit only from branch pre-execution—like *crafty* and *eon*—performance improvement is possible even without instant resolution. This speedup is conveyed by integrating the completed computation of a branch, ensuring that the branch is ready to execute as soon as it enters the window. By the same token, disallowing all integration for these programs eliminates this "computation compression" effect and eliminates the benefit of DDMT entirely, leaving only overhead and resulting is sub-baseline performance. Notice, however, this overhead is remarkably low, less than 1% in most cases. We will discuss overhead in greater detail in the next section.

Naturally, DDMT performance degrades for all programs when pre-execution reuse is removed entirely (*DDMT NI*), as resource contention in the execution engine is increased and the effects of dataflow-graph compression for integrated completed results are lost. Notice, the increase in DDT instructions executed as the result of the removal of integration is simply an artifact of our accounting method. Those additional instructions *are* executed even when pre-execution reuse is active. However, they are executed after having been integrated by the master thread and are counted as control-driven executions.

Without pre-execution reuse, the remaining performance effect is prefetching. The elimination of pre-execution reuse allows us to validate our statements of the previous section concerning the under-estimation of *PDI coverage*. Take *mcf* as an example. Using our definition of coverage, DDMT on *mcf* covers 13 million PDIs (Table 5.20). 33 million PDIs are integrated but execute post-integration and hence are not counted as being covered. However, when integration is removed, performance improvement drops from 15% to 12%. If our coverage metric was accurate and instructions executed post-integration did not cover any latency, then we would have observed performance improvement dropping by a factor of 3, to 5%. We conclude that tolerating the latency of a PDIs computation does constitute partial coverage of the PDI itself, even if the PDI is not executed before being integrated by the master thread.

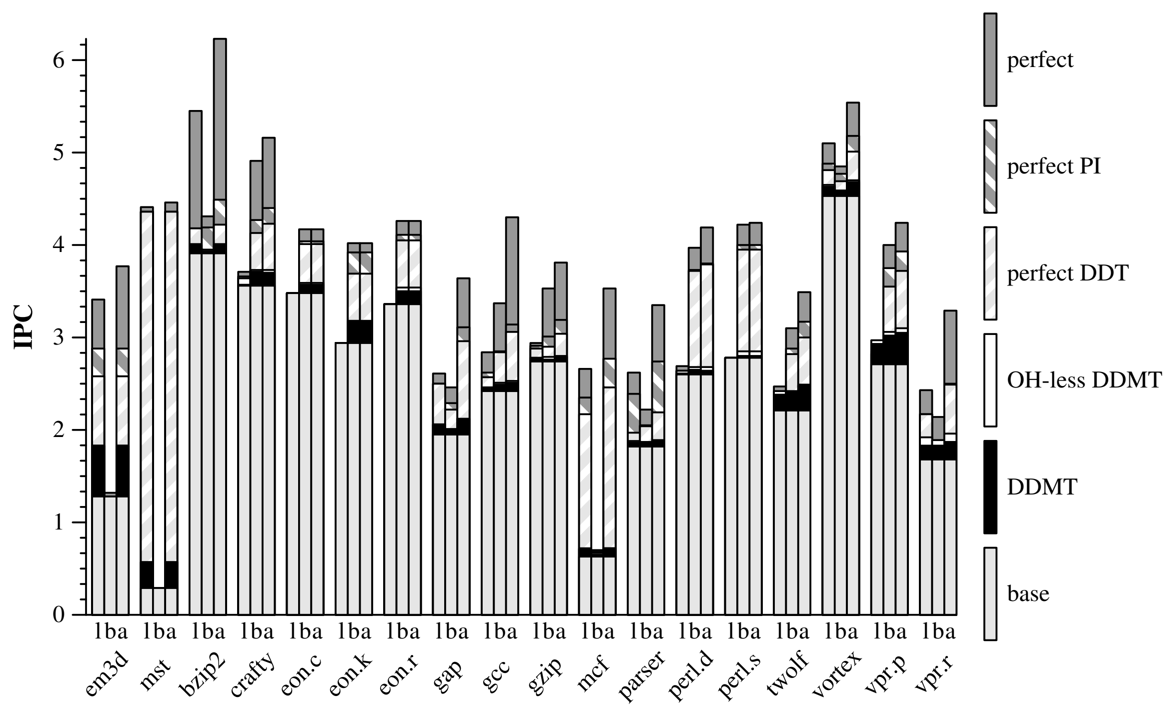
Before we leave this section, we must explain an anomaly—in *vpr.r*, performance improves dramatically when branch reuse is eliminated. After some searching, we have discovered the extremely subtle source of this anomaly. Eliminating branch re-use lengthens branch resolution latency. In the case of one particular branch, the resolution latency is lengthened just enough to allow a certain wrong path trigger instruction to be renamed and to fork a DDT. Now, this trigger instruction lies in a reconvergent code region, so that the DDT it forks is a useful one. However, the correct path conditional arm is longer than the wrong path such that waiting to fork the DDT from the correct path actually delays it significantly. In this case, allowing a DDT to be forked along the wrong path actually lets it execute sooner and achieve greater latency tolerance. This is reflected in the reduction in average load latency.

5.2.5.4 Attempting to Account for Overhead

In this penultimate experiment of our detailed analysis, we attempt to measure DDMT's overhead. We do this by removing as many sources of DDT contention as possible from the system. We remove injection and re-execution overhead by not charging DDT instructions for renaming and re-execution bandwidth, respectively. We remove reservation station contention by simulating a large number of reservation stations and enforcing a restriction only on the number allocated to master thread instructions. We also remove scheduling contention by not charging DDT instructions for scheduling slots, although this may be going a little overboard as with integration we effectively gain scheduling bandwidth for the master thread. Note, we do not have the same problem in modeling renaming bandwidth since the master thread *must* re-rename all DDT instructions.

Our evaluation is shown in Figure 5.6 and in Table 5.22. Again, the graph shows data for the *load*, *branch* and *all* experiments separately, while the table shows data only for the *all* experiment. In the graph, we add another bar to our IPC stack which represents "overhead-less" DDMT. In the table, we present the now familiar system diagnostics for DDMT with and without modeled overhead.

Figure 5.6 Performance potential of DDMT without overhead.



In the previous section, we noted that removing pre-execution reuse entirely in programs that benefit solely from branch pre-execution removes the benefit of DDMT entirely, leaving only overhead. Subsequently, we remarked at the low level of this overhead, less than 1% in most programs, even though DDT instructions account for up to a 10% increase in instructions sequenced and executed. The data shown here gives us a different look at the same phenomenon. Just as disallowing the integration of branch DDTs leaving only their overhead results in little slowdown (Table 5.2.5.3), so does removing overhead result in negligible speedup. Only in *vpr.r* does removing DDMT overhead result in even a 5% speedup. In most cases, DDMT overhead is virtually zero. In two cases—*eon.k* and *mcf*—removing overhead actually induces slowdowns!

Table 5.22 Performance potential of DDMT without overhead.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
DDMT	TIs renamed (M)	162.03	148.35	7062.58	3838.57	251.62	1278.75	399.73	1378.86	893.35
	TIs executed (M)	117.01	101.70	6283.74	2852.15	217.96	1025.10	335.15	1046.42	608.62
	BMR lat. (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load lat. (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
OH-less DDMT	TIs renamed (M)	158.17	148.86	7073.24	3946.45	252.16	1312.96	405.24	1389.43	905.46
	TIs executed (M)	111.25	101.65	6284.94	2859.58	217.76	1030.68	335.63	1041.64	605.80
	BMR lat. (c)	3.55	183.39	11.85	7.87	8.58	7.05	7.56	18.45	12.56
	Load lat. (c)	12.51	23.65	5.53	2.81	2.86	2.70	2.80	4.22	3.38
	IPC	**1.83**	**0.57**	**4.01**	**3.73**	**3.59**	**3.17**	**3.54**	**2.12**	**2.53**
	Speedup (%)	**43.25**	**94.96**	**2.36**	**4.92**	**3.18**	**7.82**	**5.50**	**8.82**	**4.73**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
DDMT	TIs renamed (M)	8300.92	2198.87	2136.36	6913.57	5023.78	2450.58	1867.48	559.91	1851.15
	TIs executed (M)	6184.76	1256.79	1560.48	4319.80	3048.72	1653.96	1752.55	377.10	1290.43
	BMR lat. (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load lat. (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
OH-less DDMT	TIs renamed (M)	8426.59	2200.78	2182.75	7044.81	4887.04	2504.74	1868.37	587.09	1831.72
	TIs executed (M)	6232.68	1213.41	1570.88	4355.76	3023.69	1662.60	1750.94	379.85	1265.19
	BMR lat. (c)	16.35	41.75	18.56	10.90	13.39	8.62	9.98	6.71	32.14
	Load lat. (c)	3.06	17.73	3.80	3.36	2.77	2.95	3.14	2.50	3.52
	IPC	**2.80**	**0.72**	**1.89**	**2.68**	**2.85**	**2.49**	**4.70**	**3.10**	**1.96**
	Speedup (%)	**2.17**	**14.19**	**3.78**	**3.16**	**2.72**	**13.02**	**3.71**	**14.69**	**16.48**

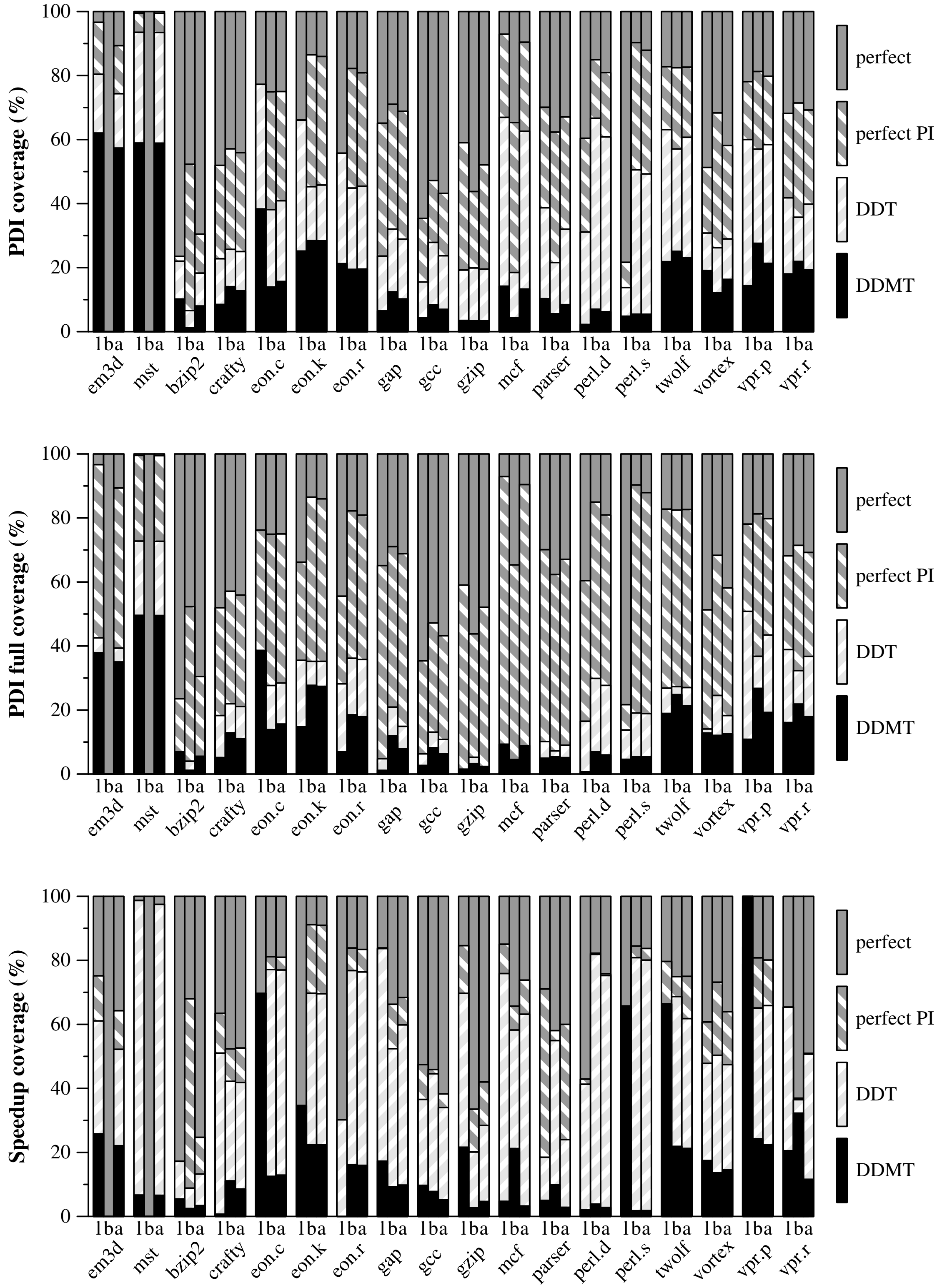
The initial result—that DDMT overhead is low—may initially seem surprising, but on further thought is actually intuitive. Our base processor has bandwidths that none of our benchmark programs is capable of fully, or even mostly, exploiting. The cost of DDTs, which typically account for a 10% to 15% increase total instructions sequenced, can easily be hidden in this slack bandwidth. Our micro-benchmarks, for which DDT pre-execution effects a 50% increase in total instruction count, have such low bandwidth utilization that even this overhead is easily hidden. We expect that DDT overhead will be expressed more strongly on narrower machines. Our early experience with these machines suggests that this is indeed so, but not to a high degree.

The phenomenon of producing speedups by the addition of overhead, seen in *eon.k* and *mcf*, is an artifact of the fact that we did not eliminate *all* possible sources of contention. Specifically, we did not add functional units, cache ports, or bus bandwidth. Ignoring all direct sources of DDT bandwidth consumption allows DDTs to swamp the machine and consume those resources we did not unconstrain. It is possible that removing constraints on these resources, perhaps via modeling the execution of DDTs as taking place on an unlimited second pipeline, would eliminate this artifact. It is also possible that these same resources we did not unconstrain are leading us to underestimate overhead. Increased contention for these resources may be actively dampening the performance of "overhead-less" DDMT.

5.2.6 Evaluating the Implementation "Stack"

Up to this point, we have evaluated each step in our implementation stack—PI definition, DDT selection and DDMT—in terms of the previous step. In this final section of our detailed analysis, we evaluate all of them against our absolute upper bound, a machine with perfect caches and perfect branch resolution. This is a slightly different way of looking at data we have already seen, one which hopefully makes it easier to assign performance shortfall—i.e., blame—to each imple-

Figure 5.7 Cumulative PDI coverage, PDI full coverage and speedup coverage.



mentation stage.

The three graphs in Figure 5.7 measure cumulative *PDI coverage*, *PDI full coverage* and *speedup coverage*. These coverages are computed with the usual numerators. However, the denominators in each case are the metrics of the *Perfect* experiment. Obviously, the Perfect model achieves 100% coverage of its own results. For the *DDT* experiments, we use projections to measure *PDI coverage* and *PDI full coverage*, and the admittedly overly aggressive *Perfect DDT* experiment to measure *speedup coverage*.

Again, the graphs are divided into bar groups that show results for *load*, *branch*, and *all* experiments separately. However, in contrast with previous breakdown graphs which showed IPCs, these graphs show coverages. The relationship of the *load* bars and the *branch* bars to the corresponding *all* bars is not approximately additive as it is for IPC. Rather, it is an averaging relationship that is weighted by the incidence of PDI loads and branches in the program. In other words, unless a program has either no load or no branch PIs whatsoever, this relationship is inscrutable from the graph.

What the graph *does* is show potential performance improvement is forfeited at each implementation stage. For instance, an average of approximately 35%, but as much as 80%, in performance improvement is lost due to the restricted PI definition. Now, this is not to say that an expanded definition would recover this lost performance. In fact, it is likely *not* to. An expanded definition will admit static instructions with poorer pre-execution characteristics—i.e., lower problem ratio and contribution rates—than the ones currently included. In this sense, this portion of performance improvement is not forfeited, it was never really attainable in the first place. In Section 5.3.2, we will explore the performance impact of expanding and narrowing the PI definition.

Approximately another 30% is lost to the inability of the DDT selection algorithm to find DDTs for the PIs that were identified. This 30% value was constructed as an ad hoc average of the *PDI coverages*, which are artificially low, and the *speedup coverage*, which is unreasonably high. Again, this 30% may not be practically recoverable. It may be the computations of those PIs for which no covering or fully covering DDTs were found are simply too dense with respect to the complete program to allow pre-execution to achieve any latency advantage. In some sense, these PIs are "super" PIs; they present problems not only for branch and address predictors but for pre-execution as well. Section 5.3.2 will also explore variations in the DDT selection algorithm parameters to try and gauge the extent of its performance impact.

Finally, about 20% in potential performance improvement is lost to the DDMT implementation itself, leaving realized performance improvement at 15%. In Section 5.3.1, we explore microarchitectural parameter variations in an attempt to see if any of this 20% can be recovered. We find that imperfect load integration suppression—i.e., a high incidence of mis-integrations or over-zealous integration avoidance—is partially responsible.

As a final note, we should mention that simply because the efficiencies of each of DDMT's implementation stages are seemingly low, it does not mean that DDMT is relegated to 15% performance improvements. Most significantly, our baseline machine has large caches and branch predictor tables that produce relatively few cache misses and branch mis-predictions for DDMT to attack. In Section 5.3.4, we examine processors with smaller caches and predictors that generate more PDIs that may be targeted by DDMT. While DDMT's performance relative to the perfect machine may stay the same in these configurations, its absolute performance gain relative to the base processor rises.

5.3 Sensitivity Analysis

In this section, we measure DDMT's sensitivity to several relevant parameters of its specific microarchitecture components, the DDT selection algorithm, the relationship between DDT selection and DDMT pre-execution and the configuration of the base processor.

In the interest of space, we summarize some of our more intuitive, or less interesting results in prose rather than provide full data and discussion. At the same time, some interactions that we would like to study are so complex that to do them justice would require a full paper. For these, we summarize our initial results and point to some of the interactions we have seen. Also in the interest of space, our sensitivity analysis does not contain breakdowns into load and branch contributions.

5.3.1 Sensitivity to DDMT Specific Microarchitecture Configuration

We have performed an array of sensitivity analyses on several aspects of the DDMT specific microarchitecture components. We have investigated the role of bandwidth contention at different stages, experimented with different DDT injection scheduling policies and altered the configuration of register integration. DDMT appears to be insensitive to most changes in its microarchitectural configuration but sensitive to two parameters in particular—the associativity of the IT and the use or lack thereof of load integration suppression. We also experiment with adding chaining—allowing one DDT to trigger the fork of another DDT—to our system. We present our results for these three parameters and briefly summarize our other findings.

5.3.1.1 Integration Associativity

Our central DDMT configuration uses a fully associative IT as a way of modeling—somewhat imprecisely, as we described—a new formulation of the physical register semantics of DDMT. This new formulation separates the two roles of the IT to avoid destructive interference that would otherwise limit the practically realizable unrolling degree to the associativity of the IT. In this section, we evaluate DDMT performance with ITs of realistic associativity—1, 2 and 4. However, since our simulator implements the old formulation, these do not model the new interference-free formulation closely. Instead, they serve to highlight its shortcomings.

Table 5.23 shows the results of four experiments. We include the results for the fully associative IT from the previous sections, and complement them with measurements for a 4-way set-associative, a 2-way set-associative and a direct-mapped IT. For each configuration we present the number of DDIs renamed, exe-

Table 5.23 Performance impact of IT associativity.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Full	DDIs renamed (M)	42.73	47.22	184.74	249.73	15.29	109.06	31.66	186.31	76.47
	DDIs executed (M)	35.75	31.36	103.44	180.69	10.95	84.52	23.60	111.28	43.27
	DDIs integrated (M)	25.05	34.29	57.59	109.18	5.48	44.28	10.83	58.20	14.45
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
4-way	DDIs renamed (M)	42.80	51.08	184.09	252.25	15.29	109.04	31.67	177.93	76.64
	DDIs executed (M)	34.88	27.12	92.35	179.98	10.95	84.46	23.60	102.72	40.88
	DDIs integrated (M)	20.64	22.93	31.72	106.24	5.48	44.27	10.82	55.01	13.35
	IPC	**1.82**	**0.36**	**3.98**	**3.69**	**3.57**	**3.18**	**3.50**	**2.04**	**2.49**
	Speedup (%)	**42.48**	**22.56**	**1.64**	**3.68**	**2.56**	**8.20**	**4.30**	**4.65**	**3.02**
2-way	DDIs renamed (M)	36.12	48.52	147.67	260.09	13.32	100.49	29.33	171.50	64.86
	DDIs executed (M)	24.00	11.00	34.96	166.16	9.95	78.69	22.53	82.59	28.20
	DDIs integrated (M)	19.01	0.09	9.97	93.98	5.44	44.29	10.81	47.72	11.35
	IPC	**1.76**	**0.30**	**3.94**	**3.67**	**3.57**	**3.18**	**3.50**	**2.02**	**2.45**
	Speedup (%)	**37.42**	**2.97**	**0.77**	**3.19**	**2.55**	**8.17**	**4.27**	**3.50**	**1.55**
Direct Map	DDIs renamed (M)	35.34	13.53	115.56	272.02	13.00	100.79	28.85	169.44	62.70
	DDIs executed (M)	20.02	1.27	28.07	151.01	9.16	69.13	20.00	59.76	19.48
	DDIs integrated (M)	5.46	0.00	7.91	81.30	4.88	39.19	10.05	34.22	8.11
	IPC	**1.71**	**0.29**	**3.94**	**3.66**	**3.56**	**3.14**	**3.49**	**2.00**	**2.42**
	Speedup (%)	**33.54**	**0.03**	**0.64**	**2.75**	**2.35**	**6.75**	**3.79**	**2.42**	**0.26**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Full	DDIs renamed (M)	493.08	740.37	223.97	894.94	586.71	499.00	49.51	115.33	239.12
	DDIs executed (M)	305.73	294.97	135.34	444.81	253.76	333.52	38.26	89.17	141.38
	DDIs integrated (M)	135.39	131.26	48.37	147.48	104.95	128.93	29.74	42.65	82.24
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
4-way	DDIs renamed (M)	491.30	678.86	229.32	877.95	571.52	482.89	49.54	117.52	239.73
	DDIs executed (M)	301.05	170.20	101.76	430.29	252.09	253.74	38.33	89.12	141.29
	DDIs integrated (M)	132.99	75.43	38.71	158.91	105.19	115.45	29.73	40.29	86.83
	IPC	**2.76**	**0.63**	**1.84**	**2.64**	**2.80**	**2.43**	**4.68**	**3.02**	**1.85**
	Speedup (%)	**0.86**	**1.16**	**0.93**	**1.45**	**0.80**	**9.98**	**3.22**	**11.54**	**9.92**
2-way	DDIs renamed (M)	492.62	573.14	203.59	779.09	386.15	423.92	48.02	118.60	300.28
	DDIs executed (M)	291.57	72.76	81.09	397.89	205.61	171.40	35.11	79.66	171.48
	DDIs integrated (M)	131.43	16.78	31.24	146.72	91.87	74.25	28.83	40.53	86.61
	IPC	**2.75**	**0.62**	**1.83**	**2.61**	**2.80**	**2.32**	**4.67**	**3.02**	**1.89**
	Speedup (%)	**0.40**	**-1.15**	**0.38**	**0.63**	**1.05**	**5.36**	**3.05**	**11.71**	**12.32**
Direct Map	DDIs renamed (M)	500.92	532.20	202.45	710.30	389.19	414.19	46.87	132.09	155.35
	DDIs executed (M)	274.43	33.01	62.40	266.66	170.67	130.19	23.39	73.42	108.25
	DDIs integrated (M)	86.30	7.45	24.43	111.24	82.73	46.15	17.83	23.25	58.54
	IPC	**2.73**	**0.63**	**1.82**	**2.56**	**2.77**	**2.27**	**4.60**	**2.79**	**1.84**
	Speedup (%)	**-0.51**	**0.54**	**-0.37**	**-1.38**	**-0.29**	**2.71**	**1.38**	**3.10**	**9.31**

cuted and integrated. In contrast with our detailed single design-point analysis, where to judge performance we compared total system metrics for the *RI* and *DDMT* configurations, here we are comparing multiple *DDMT* configurations with each other. As a result, we can choose to show whatever metrics we feel will provide the most insight, as all metrics are directly comparable. We still report IPCs *and speedups* over the *RI* configuration even though changes in the IT associativity change the performance of the baseline system as well. We note here that, in our experience, squash reuse is far less sensitive to IT associativity than pre-execution reuse. This is due primarily to unrolled idioms which are frequently pre-executed and reused but rarely squashed and reused.

The results support our assertion that—at least under the current physical register discipline—IT associativity plays an important role in DDMT performance, especially when unrolling is used. An extreme example is our serial memory latency-bound micro-benchmark—*mst*. *Mst* can benefit from high degrees of DDT unrolling and will exploit unrolling to the maximum allowed degree. The maximum unrolling degree used in all of these experiments is 4 to match our default IT configuration. Notice that the number of DDT instructions renamed and executed—and hence *mst*'s performance—drops sharply for IT associativities below 4. The low-associativity ITs simply will not allow highly unrolled DDTs to execute, initiating a cascade of invalidations that aborts the portion of a DDT that depends on a DDT physical register that is evicted from the IT. Although its DDTs are unrolled only 4 times, *mst* experiences a sharp drop in performance when IT associativity drops from full to 4-way. The reason for this is that *mst*'s

DDTs are both induction unrolled and unoverlapped fully unrolled. *Mst* executes a nested loop with a hash table lookup constituting the inner loop. To tolerate the long latencies of the hash bucket traversal, the DDT selection algorithm induction unrolls the outer loop 4 times, jumping ahead 4 outer loop iterations in each DDT, and unoverlapped fully unrolls several iterations of the inner loop. Although *mst*'s outer unrolling degree—and the only one visible to the merging algorithm—is only 4, its unrolling degree for integration purposes is 16 (16 *inner* loop iteration results may be active at any one time). When IT associativity drops below 16, destructive interference among the inner loop iterations is observed. Destructive interference among outer loop iterations takes place when IT associativity drops below 4, at which point *all* performance benefit is lost.

Mst is an extreme case due to its heavy use of induction unrolling, but all benchmarks suffer from finite associativity under the current formulation. Even without unrolling, IT conflicts between different PCs—from within the same DDT or across DDTs—can trigger destructive interference. One solution to reducing different-PC set conflicts while still maintaining reasonable integration complexity is to use an IT with a large number of sets. Our experiments with such ITs show that they are indeed useful in eliminating most conflicts that are not born of unrolling. However, unrolling is an important enough idiom to justify adopting the new formulation.

5.3.1.2 Load Integration Suppression

The ability to suppress the integration of loads that are likely to cause mis-integrations is important. The cost of a single load mis-integration—a full squash equivalent in latency to a branch mis-prediction or a memory ordering violation—far outweighs the benefit of a single load integration. However, over-zealous suppression can result in numerous lost opportunities to integrate good loads and their downstream pre-executed computations. This especially hurts if the end of a suppressed computation is a pre-executed branch that was mis-predicted by the master thread. In this section, we measure the performance impact of mis-integrations and of load integration suppression.

Table 5.24 shows the results for three experiments. Base—the second experiment—is our central DDMT configuration using an aggressive load integration suppression predictor (LISP)—a 256-entry structure that distinguishes loads based on path information. In the *None* experiment, no LISP is used and suppression is not implemented; all loads that can be integrated are integrated. Our final experiment, *Oracle*, uses a perfect LISP that can predict mis-integrations and suppress them with 100% coverage and with no false positives. For each configuration, we report the numbers of control-driven (CD) and data-driven (DD) instructions integrated. We distinguish between control-driven and data-driven—i.e., between squash reuse and pre-execution reuse—integrations by the state in which the physical register was *allocated*. We also report the number of mis-integrations, and similarly distinguish between the two mis-integration sources—control-driven and data-driven. Again, we only count successful integrations on retirement, to avoid double counting. Obviously, mis-integrations are

Table 5.24 Performance impact of load integration suppression predictor.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
None	CD integ (M)	0.60	0.40	123.63	173.71	8.34	50.93	14.80	42.69	46.99
	CD mis-integ (K)	0.00	0.01	3791.40	1126.57	83.60	548.41	125.76	466.15	141.15
	DD integ (M)	25.04	34.29	62.80	114.40	6.89	49.62	14.45	61.18	16.30
	DD mis-integ (K)	0.00	1.47	699.71	374.34	107.71	412.12	299.52	559.26	106.70
	IPC	**1.83**	**0.57**	**3.84**	**3.65**	**3.43**	**3.05**	**3.32**	**2.06**	**2.50**
	Speedup (%)	**42.97**	**93.83**	**-1.79**	**2.57**	**-1.47**	**3.75**	**-1.19**	**5.59**	**3.34**
Base	CD integ (M)	0.60	0.39	59.00	161.29	4.56	35.16	8.74	29.25	44.68
	CD mis-integ (K)	0.00	0.01	12.49	588.32	0.17	0.47	0.20	1.36	25.56
	DD integ (M)	25.05	34.29	57.59	109.18	5.48	44.28	10.83	58.20	14.45
	DD mis-integ (K)	0.00	1.22	3.71	150.57	3.94	15.33	3.33	3.11	7.26
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
Oracle	CD integ (M)	0.60	0.36	67.25	154.75	4.17	35.33	8.34	30.18	44.82
	CD mis-integ (K)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03
	DD integ (M)	25.04	34.30	60.48	112.58	6.75	48.52	13.61	61.09	16.02
	DD mis-integ (K)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	IPC	**1.83**	**0.57**	**4.00**	**3.76**	**3.62**	**3.20**	**3.58**	**2.12**	**2.53**
	Speedup (%)	**42.99**	**93.87**	**2.15**	**5.61**	**4.02**	**8.98**	**6.45**	**8.63**	**4.91**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Base	CD integ (M)	758.28	51.02	109.91	286.46	177.28	89.75	28.33	22.58	65.83
	CD mis-integ (K)	5596.96	297.68	1412.02	1798.57	160.00	87.03	46.66	196.56	388.41
	DD integ (M)	174.19	132.71	52.79	188.85	146.96	131.54	31.74	52.08	124.36
	DD mis-integ (K)	1141.77	56.00	240.67	2287.80	1283.86	865.14	118.76	61.96	56.57
	IPC	**2.69**	**0.71**	**1.82**	**2.58**	**2.86**	**2.38**	**4.64**	**3.09**	**1.93**
	Speedup (%)	**-1.73**	**12.61**	**-0.15**	**-0.83**	**2.91**	**8.05**	**2.46**	**14.02**	**14.71**
Simple	CD integ (M)	529.53	46.20	90.19	245.22	165.43	69.59	22.62	23.09	42.47
	CD mis-integ (K)	39.16	4.98	19.27	3.50	0.02	5.74	0.78	1.06	1.49
	DD integ (M)	135.39	131.26	48.37	147.48	104.95	128.93	29.74	42.65	82.24
	DD mis-integ (K)	32.16	2.22	11.81	11.34	19.81	32.56	0.06	7.93	0.45
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
None	CD integ (M)	672.54	48.56	96.41	250.09	165.54	69.91	22.78	16.98	58.13
	CD mis-integ (K)	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	00.00
	DD integ (M)	168.59	132.46	53.57	187.98	135.77	130.63	31.39	51.67	124.31
	DD mis-integ (K)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	IPC	**2.83**	**0.73**	**1.88**	**2.69**	**2.91**	**2.48**	**4.68**	**3.22**	**1.95**
	Speedup (%)	**3.41**	**16.96**	**2.84**	**3.49**	**4.94**	**12.49**	**3.29**	**19.12**	**15.90**

counted whenever they occur as mis-integrated loads are not retired. Note that successful integrations are measures in millions (M) while mis-integrations are measured in thousands (K). Again, we report speedups over the *RI* configuration even though changes to the LISP configuration change the performance of the *RI* baseline as well. However, in contrast with the previous section in which we argued that squash reuse is largely insensitive to associativity, the table shows that squash reuse suffers from mis-integrations at roughly the same rate as does pre-execution reuse.

The table shows that a suppression predictor can significantly boost performance in *many* cases by eliminating many mis-integrations. In fact, mis-integrations are so common in some programs— *bzip2*, *eon.c*, *eon.r*, *gzip*, *parser*, and *perl.d*—that without a suppression mechanism their cost swamps the positive performance impact of DDMT and results in an overall slowdown. In contrast, however, in other cases—*perl.s*, *vpr.p* and *vpr.r*—the suppression predictor can become too aggressive and falsely suppress many loads whose integration would not have caused a problem, losing the opportunity to exploit even more performance. We quantify this opportunity loss using a simulated oracle suppression mechanism.

We have experimented with a few LISP configurations, including ones that don't distinguish loads based on paths. Our 256-entry, path-based LISP gives the best performance of the few predictors we have examined, and this performance usually falls within 2% of oracle performance. However, in those three cases where we identified the LISP as being overly aggressive, its performance shortfall is about 5%. Further refinements of the LISP may recover some of this shortfall.

5.3.1.3 Trigger Chaining

In Chapter 2, we mentioned that our simulated DDMT implementation does not include *trigger chaining*, preferring to use induction unrolling instead. However, implementing chaining is trivial in our model, so an evaluation of this design choice is simple.

To implement chaining at runtime, we allow DDT instructions to check the trigger table and trigger the forking of other DDTs. However, the DDT selection algorithm must also be notified that chaining is used so that it does not produce induction-unrolled DDTs, as chaining induction-unrolled DDTs creates an extra level of redundancy. To do this, we set the maximum unrolling degree to 1. All other DDT selection parameters remain the same. Our implementation of chaining has an advantage over other implementations. Specifically, it uses the register integration mechanism to suppress redundant DDT forks. An integrating instruction is prevented from forking a DDT with the argument that the integrated instance has already forked an identical DDT.

Our evaluation of chaining is shown in Table 5.25. *DDTs forked* counts all DDTs forked by the master thread, as well as by other DDTs. *DDTs forked chain* only counts those DDTs forked by other DDTs. Obviously, when chaining is disabled, *DDTs forked chain* is zero. The remaining metrics have been explained in previous sections.

Even with register integration to suppress redundant forks, the addition of

Table 5.25 Performance impact of trigger chaining.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
DDMT	DDTs forked (M)	3.55	2.38	15.58	29.46	1.86	13.82	3.85	19.84	8.20
	DDTs forked chain (M)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	DDIs renamed (M)	42.73	47.22	184.74	249.73	15.29	109.06	31.66	186.31	76.47
	DDIs integrated (M)	25.05	34.29	57.59	109.18	5.48	44.28	10.83	58.20	14.45
	PDIs covered (M)	7.45	6.95	4.33	3.16	0.14	2.29	0.36	1.14	0.53
	BMR latency (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load latency (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
DDMT +chain	DDTs forked (M)	6.04	4.05	103.60	35.38	2.28	14.34	4.51	40.60	20.51
	DDTs forked chain (M)	4.62	3.34	99.86	11.73	0.53	2.19	0.85	28.33	13.52
	DDIs renamed (M)	49.12	50.84	410.27	267.03	16.30	101.83	32.91	308.43	95.11
	DDIs integrated (M)	24.30	36.35	22.31	101.69	4.89	42.05	10.17	60.07	14.01
	PDIs covered (M)	6.96	7.27	1.22	2.90	0.11	2.23	0.33	1.57	0.63
	BMR latency (c)	3.52	192.96	11.89	7.63	8.87	6.59	7.68	17.98	11.79
	Load latency (c)	11.64	25.54	5.52	2.82	2.88	2.70	2.80	4.11	3.30
	IPC	**1.87**	**0.54**	**4.01**	**3.69**	**3.53**	**3.17**	**3.48**	**2.12**	**2.55**
	Speedup (%)	**45.90**	**84.36**	**2.34**	**3.61**	**1.57**	**7.87**	**3.53**	**8.65**	**5.74**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
DDMT	DDTs forked (M)	60.15	66.46	41.53	74.75	37.51	56.09	2.57	9.95	12.97
	DDTs forked chain (M)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	DDIs renamed (M)	493.08	740.37	223.97	894.94	586.71	499.00	49.51	115.33	239.12
	DDIs integrated (M)	135.39	131.26	48.37	147.48	104.95	128.93	29.74	42.65	82.24
	PDIs covered (M)	2.25	13.19	2.01	2.71	1.31	9.98	0.92	1.46	4.13
	BMR latency (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load latency (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
DDMT +chain	DDTs forked (M)	62.75	406.02	201.26	92.23	45.46	112.11	3.88	13.10	32.35
	DDTs forked chain (M)	5.28	374.37	173.58	31.63	11.30	84.14	1.82	7.26	25.29
	DDIs renamed (M)	509.25	1246.6	393.28	872.27	445.50	414.72	50.21	127.70	231.89
	DDIs integrated (M)	134.59	158.58	53.27	151.25	86.67	109.18	29.44	33.50	79.00
	PDIs covered (M)	2.27	14.12	2.47	3.26	1.43	8.57	0.90	1.03	3.76
	BMR latency (c)	15.95	44.31	18.32	10.29	13.37	8.20	9.86	7.37	33.20
	Load latency (c)	3.06	18.88	3.76	3.39	2.77	2.91	3.13	2.54	3.73
	IPC	**2.79**	**0.67**	**1.86**	**2.65**	**2.80**	**2.47**	**4.67**	**2.92**	**1.89**
	Speedup (%)	**1.78**	**6.67**	**2.06**	**2.06**	**0.92**	**11.96**	**3.13**	**7.99**	**12.50**

chaining increases the total number of DDTs forked by the processor. Register integration can suppress redundant forks. However, register integration cannot detect spurious chained forks such as repeated forks of loop iterations after the loop has exited. Spurious forks damage performance. Not only do they unnecessarily steal bandwidth and resources from the master thread, but they also unnecessarily occupy thread contexts which may prevent more useful DDTs from being forked by the master thread. Substituting induction unrolling for chaining eliminates this problem. Once the master thread exits a loop, no further pre-executed iterations will be forked.

Since they are not unrolled, chained DDTs are shorter than the ones used in our central configuration. However, the increase in the number of DDTs forked usually translates into an increase in DDT instructions renamed and increased overhead. In certain programs—e.g. *mcf*—this overhead may be significant.

Ultimately, via a combination of overhead and thread context contention with master-thread forked DDTs, the addition of DDT chaining usually under-performs our unrolled implementation. Occasionally, chaining's more aggressive approach can result in higher levels of latency tolerance—e.g., in *em3d*—or increased PDI coverage—e.g., in *mst*. However, its high overhead usually means that end performance gain as a result of these improvements is low.

5.3.1.4 A Summary of Other Results

In addition to the studies presented above, we have performed studies that measure the effects of parameter variations in other DDMT-specific components.

Below is a brief summary of our findings.

DDSQ size is not an important performance factor. We have found that a 16-entry DDSQ performs as well as a 256-entry DDSQ. DDT-internal memory communication is quite frequent, but having more than 4 simultaneously active communicating pairs is not. Even when a DDT-internal communication happens as part of the induction step of an unrolled DDT, inter-DDT integration limits executed communication to a single instance per DDT. All shared instances of the communication are actually shared within the DDSQ.

We supply our DDTs with 128 additional physical registers for result buffering. However, our experiments show that, using our default DDT selection parameters of a maximum of 32 instructions per DDT and an unrolling degree of 4, only two programs—*mst* and *twolf*—benefit from having more than 64 additional physical registers, and none benefit from having more than 128 additional registers. In fact, *vpr.p* performs better with only 64 additional physical registers. Pre-execution in *vpr.p* does use more than 64 physical registers. Limiting the physical registers to 64, reduces the number of pre-executed results that are integrated, but in this particular case greatly reduces the number of loads that are mis-integrated.

DDMT's insensitivity to the number of physical registers available for storing pre-executed results is a function of the fixed size of DDTs and our choice to exclude trigger chaining. The combination of tying DDT fork to master thread progress and limiting the work performed by each DDT fixes the number of outstanding pre-executed results. With a maximum DDT length of 32 instructions, a maximum degree of unrolling of 4, and up to 3 active DDTs, the number of DDT results that must be buffered can exceed the product of these three factors, but rarely approaches it in practice.

With a fully associative IT and a proper replacement policy—i.e., *F* and *M* state registers are the primary candidates for eviction—the IT need not be larger than the number of physical registers minus the number of architected registers. Of course, a larger IT with lower associativity may be used to reduce different-PC set-conflicts as described in Section 5.3.1.1.

Just as it is insensitive to the number of additional physical registers, DDMT also does not respond to an increase in the number of reservation station entries. We did not experiment with a number of RS entries smaller than our default 80, since doing so handicaps the master thread. Since we stall DDT renaming when RS entries are not available, it would appear that additional RS entries would prevent DDT stalls and allow DDTs to be sequenced more quickly. Indeed, adding RS entries does reduce the average DDT *thread context occupancy*—the number of cycles a DDT actively occupies a thread context—implying that DDTs are sequenced faster. However, the sequenced DDT instructions are not executed any earlier than they would have been. Because DDTs are dependence chains and because many contain internal cache misses, DDT instructions are often not ready to execute immediately after being sequenced. Allowing DDTs to be sequenced more quickly does not effectively change their execution schedule or resulting latency tolerance.

Insensitivity to DDT sequencing within a certain range also manifests as insensitivity to the DDT injection policy. On our 8-wide processor, a DDT-1 policy is modeled by a frequency of 8. We have investigated policies that inject DDTs at frequencies ranging from 2—i.e., a round-robin policy—to 32 cycles. High scheduling frequencies theoretically increase effective DDT overhead, but in practice are throttled by the fixed number of RS entries. Lower frequencies compromise a DDTs latency tolerance capabilities by undermining data-driven sequencing. However, many of our DDTs have serial—i.e, pointer-chasing—cache misses which make DDT IPC far lower than 1 and dampen this effect. Reducing DDT sequencing frequency too much—we have only experimented with reductions by a factor of 4—*will* eventually negate the benefit of data-driven sequencing and render DDMT useless. For a DDT of length 16 and a *sequential span* of 512 instructions, a factor of 16 frequency reduction in scheduling frequency is needed to offset data-driven sequencing.

In addition to physical registers and reservation station slots, DDTs contend with the master thread for register renaming, scheduling and store retirement bandwidths, where the latter is used for load re-execution. DDMT performance is almost completely insensitive to increases in these bandwidths. We have already seen a preview of this insensitivity—and effectively discussed it—in Section 5.2.5.4 where we measured the performance of “overhead-less DDMT”.

Finally, we have found that DDMT performance on most programs suffers slightly when only one extra register context—rather than the default three—is available for pre-execution. Theoretically, a program pre-executing an induction-unrolled DDT of degree *N* requires *N-1* additional register contexts since, in the steady state, *N-1* copies of the unrolled DDT will be active. However, with our default unrolling and DDT size parameters, no program takes advantage of the availability of more than three additional contexts.

Of course, the number of thread contexts, the injection scheduling policy, and the numbers of physical registers and reservation station entries all independently act as throttling mechanisms. It may be that in order to improve DDMT performance, several or all of them need to be simultaneously unconstrained. We have not explored the "cross-product" sensitivity of these factors.

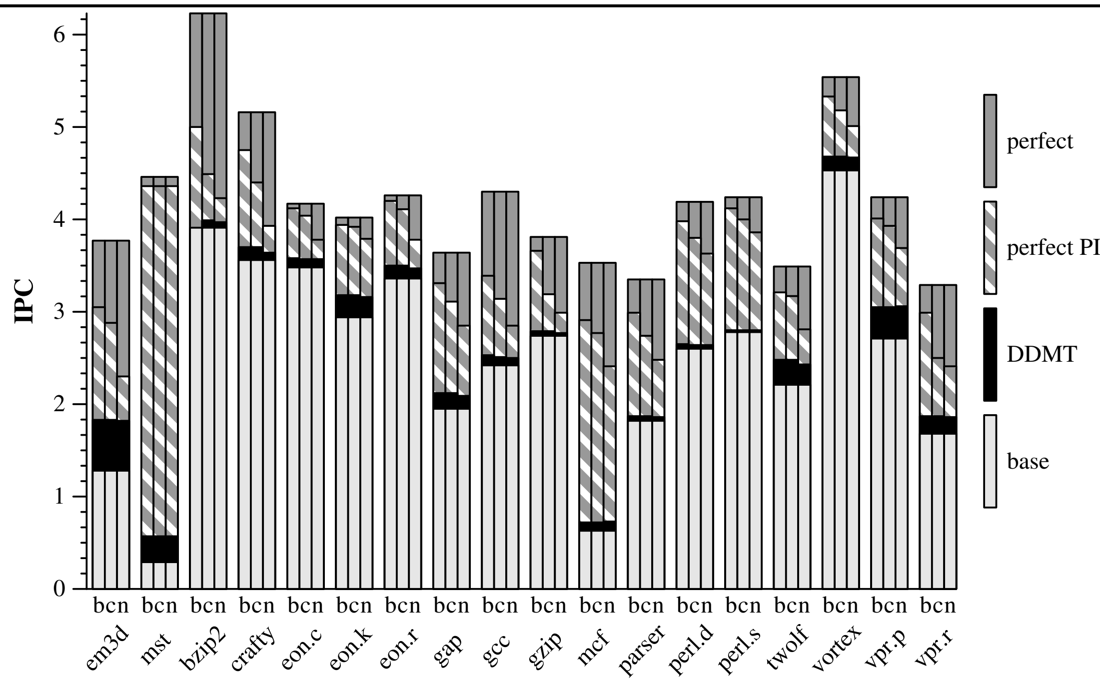
5.3.2 Sensitivity to DDT Selection Parameters

Due to the constraints of register integration, there is not much we *can* parametrize about DDTs structurally. However, we can control their general character by expanding and narrowing the PI definition, restricting DDT size and allowed degree of unrolling, and raising and lowering the latency tolerance acceptability threshold (LCAF). In this section, we investigate DDMT's sensitivity to these parameters.

5.3.2.1 Problem Instruction Definition

A good PI definition is one that, when applied to a given execution, combines the smallest number of static PIs with the highest number of dynamic PDIs and the smallest number of dynamic NPDIs. In this section, we compare three PI definitions. *Central (c)* is our central definition, with problem and contribution ratios set at 10% and 0.2%, respectively. For a *broad (b)* definition, we set the problem and contribution ratios at 5% and 0.1% respectively. The broad definition admits

Figure 5.8 DDMT sensitivity to PI definition.



more PIs. Finally, we use a *narrow (n)* definition with problem and contribution ratios of 20% and 0.5%, respectively, in an attempt to find a more focused PI pool. In all three definitions, we use minimum problem latency thresholds for both branches and loads. In particular, we do not raise the load problem latency threshold in an attempt to prune loads that only incur L1 misses.

Since the PI definition is the top implementation level, we can evaluate its cascaded impact on DDT selection and DDMT. This "stack" evaluation, similar to the one performed in detail for our central design point, is shown in Figure 5.8 and in three tables. The figure shows the familiar IPC stack of *base*, *DDMT*, *perfect PIs*, and *perfect* performance. The tables evaluate the PI definitions (Table 5.26), DDTs selected (Table 5.27) and DDMT (Table 5.28) in more detail.

Table 5.26 Comparative evaluation of PI definitions.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Broad	Static PIs	11	6	52	188	61	77	85	213	343
	PIIs executed (M)	19.12	12.67	376.99	125.78	4.24	24.94	8.38	41.64	18.52
	PDIs covered (M)	11.67	11.74	37.06	18.33	0.77	7.13	1.62	9.05	4.26
	IPC	**3.05**	**4.36**	**5.00**	**4.75**	**4.12**	**3.94**	**4.20**	**3.31**	**3.39**
	Speedup (%)	**138.66**	**1385.92**	**27.83**	**33.48**	**18.32**	**34.23**	**24.90**	**69.45**	**40.32**
Central	Static PIs	10	6	35	100	42	66	57	109	164
	PIIs executed (M)	18.28	12.67	53.14	71.64	3.05	22.12	6.42	27.79	11.06
	PDIs covered (M)	11.60	11.74	16.42	13.83	0.69	6.93	1.50	7.69	3.27
	IPC	**2.88**	**4.36**	**4.49**	**4.40**	**4.04**	**3.92**	**4.11**	**3.11**	**3.14**
	Speedup (%)	**125.16**	**1385.92**	**14.61**	**23.71**	**16.04**	**33.43**	**22.47**	**59.36**	**29.86**
Narrow	Static PIs	8	6	15	28	19	51	31	47	60
	PIIs executed (M)	14.87	12.67	21.52	16.27	1.10	18.38	2.82	12.23	4.40
	PDIs covered (M)	11.15	11.74	11.88	5.85	0.40	6.46	0.98	5.06	2.09
	IPC	**2.30**	**4.36**	**4.23**	**3.93**	**3.78**	**3.79**	**3.78**	**2.85**	**2.85**
	Speedup (%)	**79.49**	**1385.92**	**7.93**	**10.30**	**8.73**	**29.02**	**12.62**	**45.90**	**17.81**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Broad	Static PIs	46	51	239	164	52	119	130	61	67
	PIIs executed (M)	295.04	162.93	88.77	122.46	69.50	105.94	19.63	20.19	90.38
	PDIs covered (M)	47.33	94.66	19.58	38.39	22.04	36.35	4.06	5.70	18.90
	IPC	**3.66**	**2.91**	**2.99**	**3.98**	**4.12**	**3.21**	**5.33**	**4.01**	**2.99**
	Speedup (%)	**33.62**	**364.70**	**64.21**	**53.23**	**48.52**	**45.61**	**17.51**	**48.09**	**78.17**
Central	Static PIs	31	40	124	87	38	98	61	45	36
	PIIs executed (M)	144.60	140.29	53.74	85.76	51.49	101.02	9.97	17.20	44.34
	PDIs covered (M)	33.58	89.67	15.95	35.26	21.20	35.65	3.26	5.47	14.79
	IPC	**3.19**	**2.77**	**2.74**	**3.80**	**4.00**	**3.17**	**5.18**	**3.93**	**2.50**
	Speedup (%)	**16.32**	**341.58**	**50.10**	**46.34**	**44.10**	**43.53**	**14.17**	**45.36**	**48.66**
Narrow	Static PIs	12	27	56	29	22	73	23	30	19
	PIIs executed (M)	32.03	113.51	30.57	59.80	41.66	68.39	5.42	11.28	30.92
	PDIs covered (M)	14.65	82.30	11.78	31.37	20.02	30.24	2.39	4.46	12.66
	IPC	**2.99**	**2.41**	**2.48**	**3.63**	**3.86**	**2.81**	**5.01**	**3.69**	**2.41**
	Speedup (%)	**8.91**	**284.91**	**35.91**	**39.56**	**39.20**	**27.51**	**10.45**	**36.38**	**43.28**

This evaluation differs from our central "stack" evaluation in three respects. First, we do not break down our data into load and branch effects. Second, we do not show the Perfect DDT experiment, as we have already argued (and hopefully demonstrated) that its performance bound is inaccurate and uninformative.

Finally, since we are comparing different implementations to each other rather than to a single base case, we can conduct our evaluation by comparing any metrics we wish, as all metrics are comparable across implementations.

In Section 5.2.6, we reasoned that broadening or narrowing the PI definition is unlikely to have a large effect on DDMT. Broadening the definition admits instructions that are less suitable for pre-execution—judging by their problem ratio and contribution metrics—than those included by the current definition. Similarly, narrowing the definition will exclude those instructions that are least suitable for pre-execution—judging by the same criteria—from the current list. The instructions most likely to benefit from pre-execution will survive all but the narrowest of definitions.

The graph and supporting tables corroborate this hypothesis. As shown in Table 5.26, broadening the PI definition admits a large number of additional PIs. However, the performance impact of these added PIs—the number of associated PDIs covered and the performance improvement potential of "perfecting" them—is low in relation to their static representation. Take *vortex* as an example. Broadening the PI definition more than doubles the number of static PIs from 61 to 130. However, the number of PDIs covered and relative performance improvement each increase by only 20%. Broadening the definition includes more PIs, but these are "lower quality" PIs. Narrowing the definition has a similar effect in the opposite direction. The total number of PIs may halve, but since the remaining PIs are of the "highest quality", PDI coverage and performance improvement potential

Table 5.27 Characterizing DDTs selected using different PI definitions.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Broad	Static PIs	11	5	24	76	41	54	53	118	210
	PIIs executed (M)	21.64	12.34	339.83	42.32	2.47	17.25	5.32	23.87	13.95
	PDIs covered (M)	**9.75**	**11.03**	**29.74**	**6.98**	**0.40**	**3.75**	**0.86**	**3.72**	**2.18**
	PDIs fully covered (M)	5.11	8.59	14.40	5.75	0.29	2.88	0.68	1.85	1.00
Central	Static PIs	10	5	12	53	31	48	43	72	117
	PIIs executed (M)	19.06	12.34	20.56	35.23	1.97	16.83	4.96	19.53	10.27
	PDIs covered (M)	**9.65**	**11.03**	**9.87**	**6.19**	**0.37**	**3.69**	**0.84**	**3.23**	**1.79**
	PDIs fully covered (M)	5.11	8.59	2.30	5.22	0.26	2.84	0.66	1.67	0.82
Narrow	Static PIs	8	5	8	17	16	41	24	33	49
	PIIs executed (M)	16.82	12.34	15.78	15.56	1.13	16.39	2.81	8.57	5.59
	PDIs covered (M)	**9.17**	**11.03**	**9.39**	**3.67**	**0.26**	**3.54**	**0.56**	**1.67**	**1.25**
	PDIs fully covered (M)	4.63	8.59	1.99	3.04	0.17	2.75	0.43	0.30	0.54

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Broad	Static PIs	16	32	128	115	27	82	73	31	48
	PIIs executed (M)	55.70	110.65	65.49	81.08	31.83	122.23	4.06	20.41	40.21
	PDIs covered (M)	**12.70**	**63.40**	**8.79**	**27.91**	**12.35**	**26.61**	**1.85**	**4.08**	**8.77**
	PDIs fully covered (M)	1.60	1.18	2.63	12.71	4.65	11.91	1.18	3.05	8.01
Central	Static PIs	13	25	79	61	21	68	34	29	25
	PIIs executed (M)	54.98	102.52	50.23	70.05	28.92	119.55	2.75	19.42	37.88
	PDIs covered (M)	**12.59**	**62.06**	**7.61**	**26.50**	**11.88**	**26.20**	**1.62**	**4.00**	**8.51**
	PDIs fully covered (M)	1.54	1.07	2.14	12.06	4.56	11.66	1.02	2.97	7.85
Narrow	Static PIs	4	16	39	24	13	53	12	22	11
	PIIs executed (M)	7.54	84.01	30.31	54.97	26.65	87.81	1.85	15.62	27.90
	PDIs covered (M)	**2.14**	**60.85**	**5.88**	**24.18**	**11.23**	**22.60**	**1.32**	**3.65**	**7.31**
	PDIs fully covered (M)	0.02	0.81	1.58	10.33	4.19	9.93	0.88	2.85	6.85

decrease less dramatically .

Table 5.27 shows the cascading effect on DDT selection. Although speedups are not shown, the *PDIs covered* and *PDIs fully covered* measures show that the "self-dampening" of the PI definition is dampened further by the DDT selection algorithm. Although the numbers of PIs for which DDTs were successfully found maintains a roughly constant relationship with the number of PIs in the defini-

Table 5.28 DDMT performance sensitivity to different PI definitions.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
Broad	DDIs renamed (M)	42.73	47.22	1822.34	276.82	17.36	111.34	33.77	211.13	92.49
	PDIs covered (M)	11.35	8.97	96.16	13.94	0.79	6.08	1.24	5.29	2.24
	BMR latency (c)	3.54	180.54	11.25	7.48	8.05	6.48	7.31	17.93	11.94
	Load latency (c)	12.30	23.40	4.54	2.81	2.83	2.69	2.76	4.19	3.34
	IPC	**1.83**	**0.57**	**3.85**	**3.70**	**3.58**	**3.18**	**3.50**	**2.12**	**2.53**
	Speedup (%)	**42.99**	**93.83**	**-1.57**	**3.99**	**2.79**	**8.11**	**4.08**	**8.55**	**4.72**
Central	DDIs renamed (M)	42.73	47.22	184.74	249.73	15.29	109.06	31.66	186.31	76.47
	PDIs covered (M)	10.94	8.97	8.43	11.14	0.48	5.95	1.09	4.40	1.59
	BMR latency (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load latency (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
Narrow	DDIs renamed (M)	39.94	47.22	175.13	138.19	10.43	106.27	21.95	136.55	53.82
	PDIs covered (M)	8.61	8.97	5.01	3.49	0.32	5.24	0.62	0.74	0.64
	BMR latency (c)	3.52	180.54	11.73	8.26	8.57	6.58	7.85	18.74	12.59
	Load latency (c)	12.55	23.40	5.58	2.83	2.88	2.71	2.83	4.28	3.44
	IPC	**1.82**	**0.57**	**3.97**	**3.64**	**3.57**	**3.16**	**3.47**	**2.09**	**2.50**
	Speedup (%)	**42.15**	**93.83**	**1.41**	**2.22**	**2.62**	**7.61**	**3.36**	**7.15**	**3.52**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Broad	DDIs renamed (M)	497.09	749.21	259.04	987.82	597.12	509.09	60.97	117.45	246.95
	PDIs covered (M)	10.45	27.13	12.92	6.70	3.35	30.48	1.27	5.02	14.30
	BMR latency (c)	15.94	40.25	18.03	10.33	13.29	8.01	9.78	6.18	34.30
	Load latency (c)	3.06	17.26	3.79	3.35	2.76	2.93	3.12	2.49	3.74
	IPC	**2.79**	**0.72**	**1.87**	**2.65**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.17**	**2.56**	**1.91**	**0.83**	**12.53**	**3.24**	**12.65**	**11.48**
Central	DDIs renamed (M)	493.08	740.37	223.97	894.94	586.71	499.00	49.51	115.33	239.12
	PDIs covered (M)	10.24	20.82	8.54	4.45	2.79	29.87	1.06	4.58	13.19
	BMR latency (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load latency (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
Narrow	DDIs renamed (M)	307.62	633.07	145.32	788.00	564.91	361.12	29.85	101.29	213.74
	PDIs covered (M)	0.07	18.99	5.22	2.82	1.48	21.11	0.88	3.89	7.38
	BMR latency (c)	16.79	39.90	18.87	10.59	13.38	9.12	10.13	6.44	34.45
	Load latency (c)	3.07	17.08	3.88	3.39	2.77	3.02	3.14	2.52	3.77
	IPC	**2.77**	**0.73**	**1.86**	**2.64**	**2.80**	**2.43**	**4.67**	**3.06**	**1.86**
	Speedup (%)	**0.98**	**15.84**	**1.97**	**1.60**	**0.78**	**10.23**	**2.93**	**12.94**	**10.84**

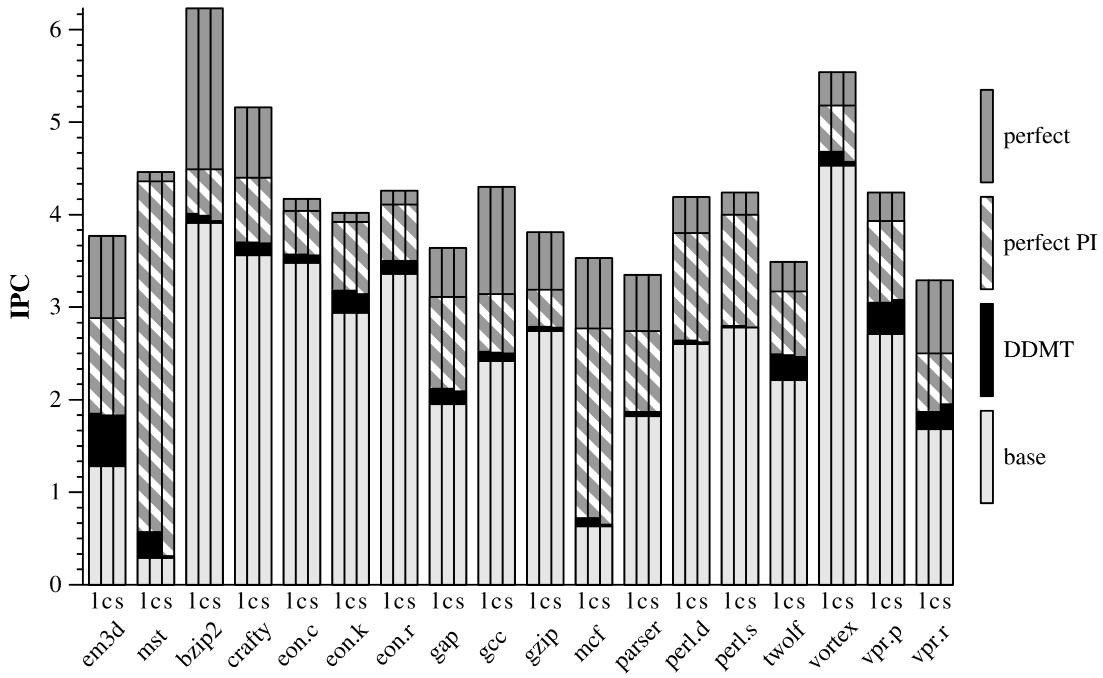
tion, the number of PDIs covered by the chosen DDTs varies by even less than the 20% relative variation of PDIs perfected. *Perl.d* is an extreme example. For the broad central and narrow definitions, DDTs are chosen for 115, 61 and 24 PIs, respectively. However, the number of PDIs covered by these DDTs varies only from 27 million to 24 million across this entire range.

Finally, Table 5.28 shows the effect of the twice-dampened PI definition on DDMT performance. It shows that, at least across the range we have studied, the PI definition has *almost no effect* on end DDMT performance. A factor of 4 difference in the number of PIs selected effects less than a 2% change in end performance. In fact, in certain cases—e.g., *bzip2*—a narrower PI definition can lead to higher (albeit slightly) overall performance. This is not a surprise. A narrower definition contains only “higher quality” PIs. Restricting pre-execution to these PIs will result in fewer PDIs actually covered. However, the number of DDT instructions required to achieve this reduced coverage may be far reduced in proportion. While DDMT’s benefit would decrease slightly, its overhead would decrease significantly. This argument makes the most sense when we take it to its broad limit. At its broadest definition, we would attempt to find DDTs for and pre-execute every PDI of every PI. We have already seen that such an approach is inherently inefficient.

5.3.2.2 DDT Size, Slicing Scope and Unrolling Degree

DDT size, slicing window size and unrolling degree constraints limit DDT length. In general, longer DDTs tolerate more latency per PDI covered. However,

Figure 5.9 DDMT sensitivity to DDT size, selection scope and unrolling degree.



longer DDTs also consume more sequencing bandwidth and cover fewer PDIs. In this section, we study the effects of three DDT size, slicing window size and unrolling degree combinations. Our *central (c)* configuration, used consistently to this point, has a slicing window of 1024 instructions, a maximum DDT length of 32 instructions and a maximum unrolling degree of 4. We add a *long (l)* configuration with maximum values of 2048, 64 and 8, respectively, and a *short (s)* configuration with maximum values of 512, 16 and 2, respectively.

Results are shown in Figure 5.9 and in two tables. Since the top level of the implementation hierarchy—the PI definition—is fixed, we evaluate only DDT selection (Table 5.29) and DDMT (Table 5.30). From the graph in Figure 5.9, our only way of gauging the impact of our parameter variations is by examining the results of the DDMT experiment. From the graph, it appears that permitting longer DDTs with higher unrolling degrees does not appreciably boost performance for any program, while restricting DDT size and unrolling degree adversely affects only a few programs, most notably the micro-benchmark *mst*. We appeal to the tables for clarification.

In Table 5.29, we have replaced our metrics of static PIs and PIIs executed with projected DDT metrics having to do with length and unrolling degree. We list the projected average dynamic DDT length and the projected average dynamic unrolling degree. The unrolling degree is averaged over all projected DDT executions, such that if a DDT of unrolling degree 3 is projected to execute half as many times as a DDT that does not use unrolling at all, the reported average unrolling degree is 1. This precise situation occurs in the two DDTs chosen for *mst*.

Table 5.29 shows that while restricting DDT length and unrolling degree to levels below those of our central configuration has an effect on the character of DDTs chosen—the DDTs are shorter and exploit unrolling to a lesser degree—loosening these restrictions does not have a pronounced opposite effect. Desired latency tolerance, LT_{des}, and the selection algorithm's implicit edict not to tolerate more latency than necessary tends to give each DDT a "natural" length which is not exceeded. The only exception to this rule is the situation in which memory latencies must be tolerated. Increased unrolling may need to be used in these cases. However, although several of our benchmarks—*mst* and *mcf*—suffer from

Table 5.29 Evaluating DDTs of different sizes, slicing scopes and unrolling degrees.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
Long	Avg. dyn. length	13.13	20.50	15.75	9.73	8.74	8.51	9.16	9.66	11.95
	Avg. dyn. unrolling	1.00	1.00	2.26	0.23	0.00	0.00	0.00	0.59	0.43
	PDIs covered (M)	**9.71**	**11.03**	**9.90**	**6.24**	**0.38**	**3.69**	**0.84**	**3.18**	**1.80**
	PDIs fully covered (M)	5.16	8.59	4.82	5.26	0.26	2.84	0.66	1.73	0.83
Central	Avg. dyn. length	12.42	20.50	13.86	9.70	8.52	8.51	9.14	9.60	11.74
	Avg. dyn. unrolling	1.00	1.00	2.26	0.23	0.00	0.00	0.00	0.59	0.36
	PDIs covered (M)	**9.65**	**11.03**	**9.87**	**6.19**	**0.37**	**3.69**	**0.84**	**3.23**	**1.79**
	PDIs fully covered (M)	5.11	8.59	2.30	5.22	0.26	2.84	0.66	1.67	0.82
Short	Avg. dyn. length	12.42	7.00	9.97	9.43	7.05	7.81	8.47	6.29	10.09
	Avg. dyn. unrolling	1.00	1.00	1.62	0.23	0.00	0.00	0.00	0.59	0.28
	PDIs covered (M)	**9.65**	**2.82**	**3.32**	**6.17**	**0.34**	**3.38**	**0.79**	**2.56**	**1.61**
	PDIs fully covered (M)	5.11	1.59	0.52	4.84	0.23	2.55	0.61	1.36	0.63

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
Long	Avg. dyn. length	7.58	12.54	6.49	13.64	17.37	9.99	18.75	12.26	19.08
	Avg. dyn. unrolling	0.03	1.70	0.80	0.01	0.05	0.46	0.00	0.53	0.78
	PDIs covered (M)	**12.59**	**62.58**	**7.65**	**26.51**	**11.88**	**25.70**	**1.63**	**4.01**	**8.51**
	PDIs fully covered (M)	1.54	10.38	2.89	12.07	4.56	12.93	1.03	2.98	7.95
Central	Avg. dyn. length	7.58	11.71	5.91	13.64	17.37	9.91	19.10	12.27	18.96
	Avg. dyn. unrolling	0.03	1.70	0.75	0.01	0.05	1.06	0.00	0.53	0.78
	PDIs covered (M)	**12.59**	**62.06**	**7.61**	**26.50**	**11.88**	**26.20**	**1.62**	**4.00**	**8.51**
	PDIs fully covered (M)	1.54	1.07	2.14	12.06	4.56	11.66	1.02	2.97	7.85
Short	Avg. dyn. length	4.88	6.57	5.30	9.65	11.10	8.02	11.64	10.76	12.69
	Avg. dyn. unrolling	0.03	1.30	0.63	0.01	0.00	0.49	0.00	0.54	0.67
	PDIs covered (M)	**10.94**	**63.97**	**7.01**	**22.38**	**9.36**	**24.16**	**1.60**	**3.55**	**8.11**
	PDIs fully covered (M)	1.54	0.31	2.00	5.27	2.79	10.61	0.28	2.38	3.52

high L2 miss rates in this large-cache configuration, it appears that our central maximum degree of unrolling, 4, is sufficient to create DDTs that will tolerate these latencies. Even if this is not the case in practice, the DDT selection algorithm "believes" that no advantage is to be gained by exploiting further unrolling.

Unlike variations in the PI definition, varying DDT size and unrolling constraints are not "damped" by the DDT selection algorithm. As constraints are tightened, the impact is shown directly in the *PDIs fully covered* metric, as

Table 5.30 DDMT performance sensitivity to DDTs of different sizes.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
Long	DDIs renamed (M)	45.06	47.22	205.43	251.60	15.66	109.06	31.71	187.13	77.17
	PDIs covered (M)	10.97	8.97	9.00	11.20	0.48	5.95	1.09	4.39	1.57
	BMR latency (c)	3.57	180.54	11.64	7.55	8.27	6.48	7.32	18.08	12.23
	Load latency (c)	11.84	23.40	5.48	2.81	2.85	2.69	2.78	4.20	3.38
	IPC	**1.85**	**0.57**	**4.01**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.52**
	Speedup (%)	**44.83**	**93.83**	**2.35**	**3.86**	**2.60**	**8.21**	**4.31**	**8.58**	**4.16**
Central	DDIs renamed (M)	42.73	47.22	184.74	249.73	15.29	109.06	31.66	186.31	76.47
	PDIs covered (M)	10.94	8.97	8.43	11.14	0.48	5.95	1.09	4.40	1.59
	BMR latency (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load latency (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
Short	DDIs renamed (M)	42.73	17.28	63.43	247.59	12.27	100.84	27.99	119.54	66.42
	PDIs covered (M)	10.94	3.69	2.60	10.83	0.44	5.33	1.08	3.81	1.50
	BMR latency (c)	3.54	378.72	12.66	7.58	8.49	6.90	7.36	18.58	12.40
	Load latency (c)	12.30	48.52	5.60	2.81	2.87	2.72	2.78	4.25	3.41
	IPC	**1.83**	**0.31**	**3.93**	**3.69**	**3.56**	**3.14**	**3.50**	**2.09**	**2.50**
	Speedup (%)	**42.99**	**6.03**	**0.47**	**3.81**	**2.22**	**6.95**	**4.23**	**7.37**	**3.65**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Long	DDIs renamed (M)	493.08	780.74	218.45	894.98	586.71	408.03	49.73	115.29	240.52
	PDIs covered (M)	10.24	21.20	6.86	4.45	2.79	29.03	1.08	4.61	13.18
	BMR latency (c)	15.95	40.00	18.22	10.45	13.29	8.25	9.95	6.21	34.27
	Load latency (c)	3.07	17.26	3.84	3.38	2.76	2.99	3.13	2.49	3.74
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.49**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**14.68**	**2.62**	**1.72**	**0.97**	**12.64**	**3.24**	**12.65**	**11.12**
Central	DDIs renamed (M)	493.08	740.37	223.97	894.94	586.71	499.00	49.51	115.33	239.12
	PDIs covered (M)	10.24	20.82	8.54	4.45	2.79	29.87	1.06	4.58	13.19
	BMR latency (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load latency (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
Short	DDIs renamed (M)	235.58	413.57	196.92	583.58	411.20	311.77	43.58	95.93	176.35
	PDIs covered (M)	10.76	17.55	6.55	3.13	2.85	28.40	0.76	5.38	18.12
	BMR latency (c)	16.39	50.87	18.39	10.88	13.67	8.37	11.78	5.91	32.03
	Load latency (c)	3.08	20.15	3.87	3.42	2.79	2.97	3.17	2.47	3.52
	IPC	**2.78**	**0.65**	**1.87**	**2.62**	**2.78**	**2.46**	**4.57**	**3.08**	**1.95**
	Speedup (%)	**1.27**	**3.26**	**2.31**	**0.75**	**0.23**	**11.51**	**0.81**	**13.71**	**16.29**

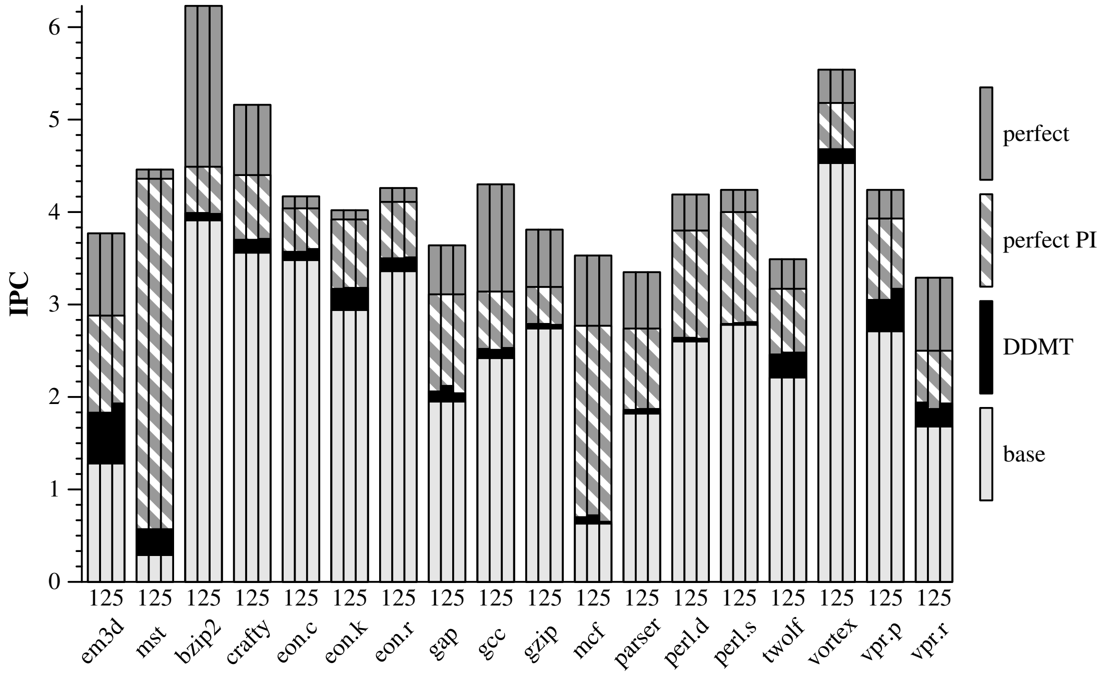
shorter DDTs are unable to tolerate the full latency of their target PDIs. If the constraint is tightened sufficiently, the number of *PDIs covered* may drop as well, as the shorter DDT may not be able to achieve threshold (LCAF) latency tolerance.

However, shorter DDTs do not always imply lower performance as shown in Table 5.30. It is possible for the DDT selection algorithm to underestimate latency tolerance, and choose long DDTs when shorter ones would have sufficed. Restricting DDT selection to short DDTs prevents the DDT selection algorithm from using its faulty estimates. We observe this phenomenon in both *vpr* benchmarks. In general, however, shorter DDTs tolerate less latency per PDI and result in lower performance. As we initially observed in the graph, *mst*'s and *mcf*'s performance improvement disappears when unrolling is limited to 2 iterations. These benchmarks have a high incidence of serial memory accesses whose latencies can only be hidden using high unrolling degrees.

5.3.2.3 Latency Tolerance Acceptability Factor (LCAF)

The LCAF determines what level of latency tolerance makes a DDT minimally acceptable. For instance, with an LCAF of 50%—i.e., a DDT must tolerate at least 50% the required latency—and an LT_{des} of 80, a DDT that tolerates 30 cycles of latency is deemed unacceptable. Our default LCAF is 25%. In this section, we measure the DDT selection algorithm's and DDMT's sensitivity to this parameter by experimenting with LCAFs of 50% and 10%. Again changes in the LCAF leave the upper level of the implementation stack unchanged, leaving us to evaluate

Figure 5.10 DDMT sensitivity to latency coverage acceptability threshold (LCAF).



DDT selection and DDMT. The results are shown in Figure 5.10, with the 10%, 20% and 50% experiments denoted by the numbers 1, 2 and 5, respectively, under each bar group. As in the previous section, Table 5.31 characterizes the DDTs chosen using each one of these LCAF values while Table 5.32 provides DDMT performance details.

As the graph and tables show, the ideal LCAF is different for different programs. Like most DDT selection parameters, tuning the LCAF is a balancing act. A high LCAF will cause the DDT selection algorithm to choose a DDT with a higher per-PDI latency tolerance over one with sub-LCAF latency tolerance but a higher aggregate advantage. As shown in Table 5.31, this is the case for *em3d*. However, the other effect of a high LCAF is to cause the DDT selection algorithm

Table 5.31 Evaluating DDTs chosen using different LCAF values.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
10%	Avg. dyn. length	2.82	11.00	8.68	6.93	7.60	5.31	5.79	6.69	7.37
	Avg. dyn. unrolling	1.00	1.00	1.37	0.22	0.00	0.00	0.00	0.48	0.35
	PDIs covered (M)	**9.65**	**11.03**	**10.70**	**6.58**	**0.38**	**3.99**	**0.85**	**3.89**	**2.02**
	PDIs fully covered (M)	5.11	8.59	2.30	5.22	0.26	2.81	0.66	1.65	0.79
25%	Avg. dyn. length	12.42	20.50	13.86	9.70	8.52	8.51	9.14	9.60	11.74
	Avg. dyn. unrolling	1.00	1.00	2.26	0.23	0.00	0.00	0.00	0.59	0.36
	PDIs covered (M)	**9.65**	**11.03**	**9.87**	**6.19**	**0.37**	**3.69**	**0.84**	**3.23**	**1.79**
	PDIs fully covered (M)	5.11	8.59	2.30	5.22	0.26	2.84	0.66	1.67	0.82
50%	Avg. dyn. length	3.35	12.50	14.32	7.31	8.85	6.58	6.24	8.97	9.40
	Avg. dyn. unrolling	1.90	1.50	2.42	0.27	0.00	0.00	0.00	0.66	0.50
	PDIs covered (M)	**10.68**	**11.03**	**5.57**	**6.38**	**0.36**	**3.72**	**0.78**	**2.50**	**1.43**
	PDIs fully covered (M)	6.06	8.59	2.30	5.29	0.26	2.97	0.68	1.73	0.89

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
10%	Avg. dyn. length	6.60	5.37	3.99	9.55	10.49	6.00	17.93	6.07	7.92
	Avg. dyn. unrolling	0.03	1.28	0.47	0.01	0.04	1.06	0.00	0.50	0.74
	PDIs covered (M)	**13.01**	**64.27**	**8.02**	**28.02**	**14.14**	**25.83**	**1.67**	**4.04**	**8.48**
	PDIs fully covered (M)	1.54	1.07	1.95	11.83	4.56	11.58	1.02	2.90	7.85
25%	Avg. dyn. length	7.58	11.71	5.91	13.64	17.37	9.91	19.10	12.27	18.96
	Avg. dyn. unrolling	0.03	1.70	0.75	0.01	0.05	1.06	0.00	0.53	0.78
	PDIs covered (M)	**12.59**	**62.06**	**7.61**	**26.50**	**11.88**	**26.20**	**1.62**	**4.00**	**8.51**
	PDIs fully covered (M)	1.54	1.07	2.14	12.06	4.56	11.66	1.02	2.97	7.85
50%	Avg. dyn. length	3.80	6.35	5.38	10.21	12.68	6.50	19.37	7.40	9.46
	Avg. dyn. unrolling	0.03	1.86	0.83	0.01	0.06	1.19	0.00	0.59	0.71
	PDIs covered (M)	**10.93**	**42.20**	**6.11**	**17.07**	**10.59**	**26.13**	**1.50**	**3.92**	**6.87**
	PDIs fully covered (M)	1.54	1.07	2.48	9.40	4.63	13.03	1.23	3.14	5.90

to reject a DDT with a positive but sub-LCAF latency tolerance in favor of no DDT at all. For, instance setting the LCAF at 100% will result in the DDT selection algorithm choosing only DDTs that can achieve full latency coverage on every PDI. Intuitively, a 100% LCAF is too restrictive, especially if long memory latencies are to be tolerated. It is not difficult to find DDTs that can tolerate 20 or 30 cycles of latency. Finding DDTs that can hide 100 cycles is more difficult. The combination of two effects of the LCAF produces the following result. In general,

Table 5.32 DDMT sensitivity to different LCAF values.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
10%	DDIs renamed (M)	36.60	47.13	185.19	255.79	13.53	105.03	29.45	169.58	64.93
	PDIs covered (M)	10.93	8.97	8.38	11.37	0.48	5.73	1.09	4.33	1.62
	BMR latency (c)	3.54	181.50	11.69	7.54	8.27	6.57	7.31	18.89	12.20
	Load latency (c)	12.31	23.40	5.52	2.81	2.86	2.69	2.78	4.30	3.37
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.17**	**3.50**	**2.06**	**2.52**
	Speedup (%)	**42.97**	**93.83**	**1.98**	**3.92**	**2.63**	**7.84**	**4.33**	**5.56**	**4.23**
25%	DDIs renamed (M)	42.73	47.22	184.74	249.73	15.29	109.06	31.66	186.31	76.47
	PDIs covered (M)	10.94	8.97	8.43	11.14	0.48	5.95	1.09	4.40	1.59
	BMR latency (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load latency (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
50%	DDIs renamed (M)	54.06	47.13	120.77	249.35	14.22	111.55	29.13	149.94	63.81
	PDIs covered (M)	11.37	8.97	4.80	11.32	0.47	5.72	1.06	4.01	1.52
	BMR latency (c)	3.59	181.50	11.97	7.48	7.99	6.47	7.37	19.44	12.17
	Load latency (c)	11.05	23.40	5.56	2.81	2.85	2.69	2.79	4.34	3.39
	IPC	**1.93**	**0.57**	**3.98**	**3.71**	**3.60**	**3.18**	**3.51**	**2.04**	**2.53**
	Speedup (%)	**51.00**	**93.83**	**1.78**	**4.19**	**3.33**	**8.18**	**4.54**	**4.55**	**4.52**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
10%	DDIs renamed (M)	499.84	584.75	188.78	789.92	412.70	426.58	47.83	111.13	233.78
	PDIs covered (M)	10.23	21.94	7.25	4.27	2.98	30.96	1.08	4.67	18.75
	BMR latency (c)	15.95	42.22	18.65	10.43	13.49	7.99	9.95	6.30	32.15
	Load latency (c)	3.05	17.76	3.85	3.39	2.77	2.92	3.13	2.50	3.53
	IPC	**2.79**	**0.70**	**1.86**	**2.64**	**2.79**	**2.46**	**4.68**	**3.05**	**1.94**
	Speedup (%)	**1.81**	**12.45**	**1.78**	**1.77**	**0.44**	**11.44**	**3.33**	**12.55**	**15.71**
25%	DDIs renamed (M)	493.08	740.37	223.97	894.94	586.71	499.00	49.51	115.33	239.12
	PDIs covered (M)	10.24	20.82	8.54	4.45	2.79	29.87	1.06	4.58	13.19
	BMR latency (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load latency (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
50%	DDIs renamed (M)	235.92	551.89	183.73	750.44	343.86	445.50	39.03	121.43	251.52
	PDIs covered (M)	10.77	19.67	6.79	4.13	2.84	33.08	0.86	6.01	16.83
	BMR latency (c)	16.38	50.33	18.37	10.54	13.37	7.84	9.76	5.04	32.65
	Load latency (c)	3.08	19.18	3.86	3.41	2.77	2.89	3.13	2.45	3.57
	IPC	**2.78**	**0.65**	**1.87**	**2.63**	**2.81**	**2.48**	**4.68**	**3.17**	**1.93**
	Speedup (%)	**1.27**	**4.37**	**2.29**	**1.36**	**1.17**	**12.17**	**3.24**	**17.20**	**14.70**

the high LCAF results in DDTs that cover fewer total PDIs, but *fully* cover more PDIs. These trends show clearly in Table 5.31.

How LCAF-driven changes in the character of DDTs impact performance is shown in Table 5.32. The performance effects are not consistent, even within a program. Certain programs, like *perl.s*, perform consistently better with a higher LCAF. Others, like *perl.d*, perform consistently worse. Certain programs, like *gap* and *mcf*, perform better with DDTs chosen using an LCAF of 25% than with DDTs chosen using an LCAF of either 10% or 50%. Other programs, like vpr.r, exhibit precisely the opposite behavior performing better with DDTs chosen using either a low or a high LCAF.

These different responses can be explained using the combination of LCAF's two effects: the "threshold" effect, in which latency tolerant DDTs are rejected because their latency tolerance levels do not meet LCAF requirements, and the "latency tolerance" effect in which more latency tolerant DDTs are preferred over ones with higher aggregate advantage. Suppose that by raising the LCAF, a certain DDT is eliminated from the chosen set. If this DDT was falsely preferred due to the "latency tolerance" effect, then eliminating it will improve performance. Otherwise, eliminating the DDT will result in performance degradation.

5.3.2.4 Summary of Other Results

Although currently not a parameter of the DDT selection algorithm, one variable that may benefit from being externally parametrizable is the formula used to compute the *master thread's effective sequencing bandwidth consumption* (BW_{seq-}

$_{CD}$) from its IPC and the sequencing width of the machine. $BW_{seq\text{-}CD}$ is an important parameter in DDT selection as it effectively conveys both the master thread's sequencing constraint which is used in latency tolerance calculations and its consumption of sequencing bandwidth which is used in overhead calculations. After a short run of "pre-tuning", we have chosen to define $BW_{seq\text{-}CD}$ as the average of the master thread's IPC and the sequencing width of the machine, weighing IPC 2-to-1. What our "pre-tuning" exercise showed is that changes in this formula—i.e., weighing IPC differently—can produce radically different DDTs and, consequently, different levels of DDMT performance. It also appears that different formulae work better for different programs and, furthermore, these changes do not appear correlated with the program's base IPC. The formula we chose works produces good DDTs for most benchmarks, and is the best formula—of the ones we have tried—for 6 of the benchmarks.

As of this writing, we do not have a clear understanding of how different formula weights determine the character of the chosen DDTs and the resulting DDMT performance impact. In general, a formula that produces a high $BW_{seq\text{-}CD}$—i.e., one that weighs sequencing width more heavily—reduces the master thread's sequencing constraint and leads the DDT selection algorithm to construct longer, more highly unrolled DDTs that tolerate more latency, but consume more overhead. However, a high $BW_{seq\text{-}CD}$ may also make it appear that a DDT is unable to achieve any sequencing advantage over the master thread and cause the DDT selection algorithm to reject certain DDTs which in practice would be useful. It is possible that some combination of formula weights, LCAF and unrolling degree could achieve better DDT selection uniformly across all benchmarks. Empirically, however, different programs respond differently to different combinations.

One shortcoming of the formula we are currently aware of is that its IPC measure is taken as a global constant over the execution of the program. This simplification ignores the real scenario of different program phases with different IPCs. In these cases, it is likely that the single IPC we use is too high for PDIs in certain phases and too low for PDIs in other phases. Correcting this shortcoming, and understanding the impact of this parameter are areas for future research.

5.3.3 Sensitivity to Setup/Runtime Relationship

Up until this point, our evaluation modeled a *limit* scenario in which DDT selection and DDMT pre-execution take place over the same program sample. In this section, we examine the effect of more realistic relationships between DDMT's setup and runtime components.

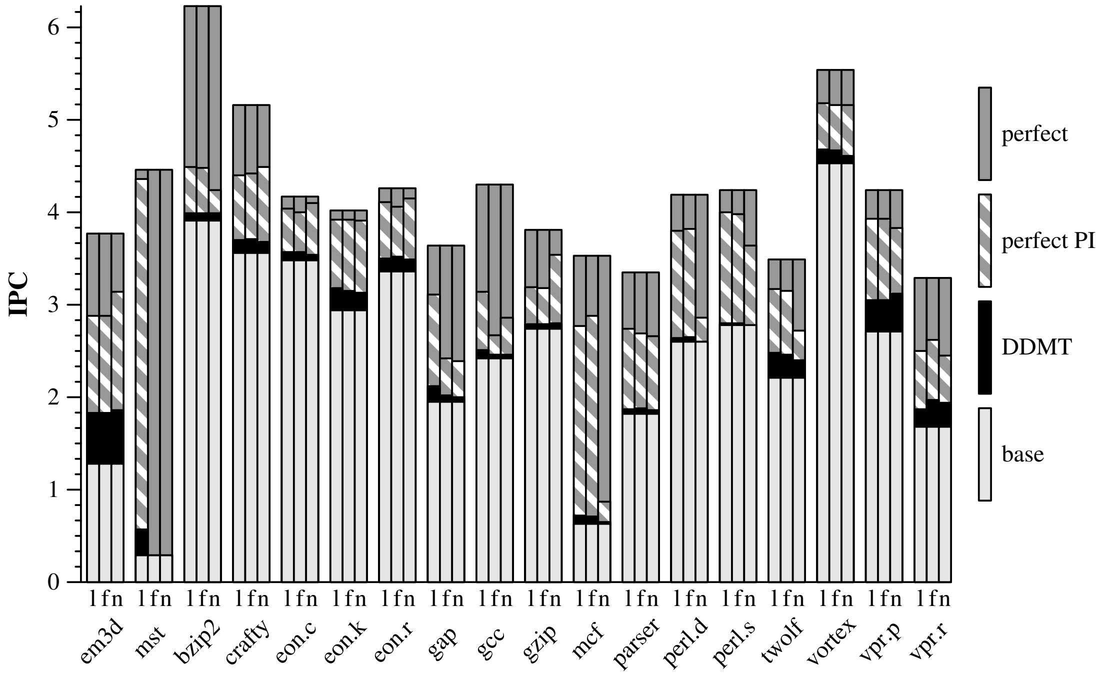
In this dissertation, we do not provide an actual implementation of the setup phase nor do we model its cost. Consequently, our investigation is *not* of the total system performance of the various implementations, but rather only of the effects of *cross-training* DDT selection. Still, such a study has value in that it speaks to the *cross-data-set stability* of both the PI definition and the DDTs themselves.

We are concerned with stability in two different scenarios. An *offline* scenario which models a software implementation of DDT selection, and an *online* scenario which models a dynamically optimizing VM implementation. In order to model these scenarios with some degree of accuracy we must choose the input program sample to the DDT selection algorithm with care. For the *offline* scenario, we substitute the benchmark *test* inputs for their *training* inputs. Since test inputs are much shorter than training inputs, sampling at 1B instruction intervals is too coarse and may not produce a representative execution cross-section. Test runs are sampled at 100M instruction intervals where the first 80M instructions are fast-forwarded, the next 10M are used for warming up, and the last 10M are used for DDT selection. If a program has multiple test inputs (e.g., *eon* and *vpr*), we select DDTs using all of them and then merge the resulting sets of DDTs together using our merging procedure. Each of a program's training runs then executes an identical set of DDTs constructed using all test runs.

For the *online* scenario, we keep the training input for DDT selection purposes but use a different sample within that run. Specifically, we use the 10M instructions that immediately precede each performance sample. This scenario "models" a VM in which a the program is observed for 10M instructions, DDTs are selected using information about that sample and then those DDTs are pre-executed over the next 100M instructions.

Cross-training can actually be done at two levels. We can extract PI definitions using one sample, select the appropriate DDTs for PDIs of those PIs using a second sample, and pre-execute the DDTs on a third sample. We perform only *single-level cross-training*, identifying PIs and selecting DDTs using the same sample. We use our central PI definition and DDT selection parameters.

Figure 5.11 DDMT sensitivity to setup/runtime relationship.



The results are shown in Figure 5.11 and in two supporting tables. For this particular sensitivity analysis, the selected DDTs cannot be evaluated directly as has been our method to this point. Because DDT selection no longer uses the performance evaluation input sample to select DDTs, its statistics and measures cannot be used as projections. We are left with evaluating the PI definition (Table 5.33) and the end performance effect of the generated DDTs (Table 5.34).

Table 5.33 presents *PI definition stability* data. The presented data was obtained by running the *perfect PI* experiment on the performance sample, using the PIs found in each of the three samples: *limit* (the performance sample itself), *online*, and *offline*. In the interest of space, we list only raw statistics. The results confirm our intuition that an instruction's PI status is a function of its computa-

Table 5.33 PI definition stability under different setup/runtime scenarios.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
Limit	Static PIs	10	6	35	100	42	66	57	109	164
	PIIs executed (M)	18.28	12.67	53.14	71.64	3.05	22.12	6.42	27.79	11.06
	PDIs covered (M)	11.60	11.74	16.42	13.83	0.69	6.93	1.50	7.69	3.27
	IPC	**2.88**	**4.36**	**4.49**	**4.40**	**4.04**	**3.92**	**4.11**	**3.11**	**3.14**
	Speedup (%)	**125.16**	**1385.92**	**14.61**	**23.71**	**16.04**	**33.43**	**22.47**	**59.36**	**29.86**
Offline	Static PIs	12	4	38	108	70	75	77	83	139
	PIIs executed (M)	21.70	0.00	23.91	95.56	4.57	22.29	7.31	14.97	6.84
	PDIs covered (M)	12.48	0.00	11.49	15.13	0.75	6.92	1.54	3.96	2.17
	IPC	**3.14**	**0.29**	**4.24**	**4.49**	**4.10**	**3.91**	**4.15**	**2.39**	**2.86**
	Speedup (%)	**145.57**	**0.00**	**8.42**	**26.30**	**17.92**	**33.13**	**23.52**	**22.54**	**18.58**
Online	Static PIs	10	0	35	102	40	67	59	102	168
	PIIs executed (M)	18.28	0.00	50.04	73.84	2.74	22.20	6.47	23.55	5.06
	PDIs covered (M)	11.60	0.00	13.78	14.06	0.64	6.94	1.48	5.63	1.49
	IPC	**2.88**	**0.29**	**4.48**	**4.42**	**4.00**	**3.92**	**4.06**	**2.42**	**2.67**
	Speedup (%)	**125.16**	**0.00**	**14.51**	**24.19**	**15.03**	**33.43**	**20.96**	**24.14**	**10.33**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Limit	Static PIs	31	40	124	87	38	98	61	45	36
	PIIs executed (M)	144.60	140.29	53.74	85.76	51.49	101.02	9.97	17.20	44.34
	PDIs covered (M)	33.58	89.67	15.95	35.26	21.20	35.65	3.26	5.47	14.79
	IPC	**3.19**	**2.77**	**2.74**	**3.80**	**4.00**	**3.17**	**5.18**	**3.93**	**2.50**
	Speedup (%)	**16.32**	**341.58**	**50.10**	**46.34**	**44.10**	**43.53**	**14.17**	**45.36**	**48.66**
Offline	Static PIs	14	30	133	59	24	66	80	53	24
	PIIs executed (M)	264.72	62.06	44.76	21.76	38.25	46.21	10.19	14.89	41.83
	PDIs covered (M)	42.88	26.47	11.59	10.04	16.88	10.62	2.86	4.10	14.11
	IPC	**3.54**	**0.87**	**2.66**	**2.86**	**3.64**	**2.72**	**5.16**	**3.83**	**2.45**
	Speedup (%)	**29.07**	**38.31**	**45.87**	**10.18**	**31.15**	**23.48**	**13.88**	**41.44**	**45.66**
Online	Static PIs	32	42	114	86	36	101	54	45	23
	PIIs executed (M)	141.73	148.83	46.17	94.01	51.13	100.44	10.43	17.20	71.17
	PDIs covered (M)	33.28	92.45	14.24	35.78	21.00	35.49	3.21	5.47	16.87
	IPC	**3.18**	**2.88**	**2.69**	**3.82**	**3.98**	**3.15**	**5.16**	**3.93**	**2.62**
	Speedup (%)	**16.13**	**359.25**	**47.40**	**46.99**	**43.48**	**42.98**	**13.91**	**45.36**	**55.94**

tion—i.e., program structure—and is largely independent of the input data set. In general, the cross-trained PI definition is more stable in the *online* scenario than in the *offline* scenario. This is not a surprise since, after all, in the *online* scenario we train the PI definition using the same input data set, only with a different sample from within that run. In the *offline* scenario, the input data set is different. In both cross-training scenarios it is possible to obtain both a broader PI definition that will cover more dynamic PDIs and a narrower PI definition that will cover fewer PDIs.

In the *offline* scenario, differences are largely due to different program working sets exercised by each individual data set. For instance, *vpr* has two input test data sets, *place* and *route*, and two matching training data sets. The *place* and *route* runs are not similar to each other from an execution standpoint, exercising largely independent sections of the program. Accordingly, their individual PI definitions will have some instructions in common and some different instructions. The merged PI definition will contain those instructions which both sets defined as PIs and then some but not necessarily all instructions that appear in only one set. The degree to which PI samples from each set overlap will determine whether the merged definition is closer to being the intersection of the two definitions or their union. An intersection will produce a narrower definition, a union will produce a wider definition. Sampling differences do play a role as well, since the *offline* sample is typically much smaller than its corresponding training sample.

Sampling differences are the only differences in online scenario. As discussed in Section 5.1.4, the 10% sampling rate used for our performance experiments yields little sampling error. However, our online training methodology uses a 1% sampling rate, which likely results in larger errors. An extreme case of such sampling error is *mst*. The *mst* benchmark has three distinct phases, of which only the last one has many PDIs. *Mst*'s training data accommodates only a single performance sample and, thus, a single online DDT selection sample as well. While the performance sample straddles the second and third phases, the DDT selection sample executes exclusively in the second phase. As a result, it captures none of the PDIs that dominate the third phase.

In Table 5.34, we measure DDMT's *performance stability* in each of the three scenarios. Rather than provide full system and pre-execution diagnostics, we present a few choice metrics. We show the number of DDIs renamed and integrated, average load and branch misprediction resolution (BMR) latency and IPC. DDMT performance stability tracks the stability of the PI definition. This is an intuitive result. We have already argued that PI stability is due to its basis in program structure, which is fixed. The structure of DDTs—which after all are computations extracted directly from execution traces—is also determined by program structure. Since the underlying processor modeled by the DDT selection algorithm is constant, PI problem penalties and subsequent levels of desired latency tolerance, LT_{des}, are also roughly constant. Finally, we also keep the DDT selection parameters fixed. With the combination of these factors, it stands to reason that for a given PDI the same DDT will be selected independently of the DDT selection sample. Obviously, a stable set of DDTs leads to stable performance. Both the performance numbers and the diagnostics bear this out.

Table 5.34 DDMT sensitivity to setup/runtime scenarios.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI IPC		**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
Limit	DDIs renamed (M)	42.73	47.22	184.74	249.73	15.29	109.06	31.66	186.31	76.47
	DDIs integrated (M)	25.05	34.29	57.59	109.18	5.48	44.28	10.83	58.20	14.45
	BMR lat. (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load lat. (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
Offline	DDIs rename (M)	41.30	0.00	93.23	285.54	29.77	162.28	51.13	92.61	68.26
	DDIs integrated (M)	25.18	0.00	51.89	122.59	8.80	49.27	14.47	22.62	19.04
	BMR lat. (c)	2.17	404.81	11.85	7.64	8.00	6.63	7.01	20.22	12.71
	Load lat. (c)	13.48	54.73	5.53	2.81	2.82	2.70	2.77	4.44	3.49
	IPC	**1.86**	**0.29**	**3.99**	**3.68**	**3.54**	**3.13**	**3.49**	**2.00**	**2.46**
	Speedup (%)	**45.25**	**0.00**	**2.04**	**3.37**	**1.78**	**6.47**	**3.95**	**2.67**	**1.91**
Online	DDIs executed (M)	42.73	0.00	153.25	274.96	14.98	121.49	36.36	151.65	39.92
	DDIs integrated (M)	25.05	0.00	35.64	120.25	5.27	42.16	10.86	42.63	7.52
	BMR lat. (c)	3.54	404.81	11.69	7.36	8.13	6.76	7.05	19.99	13.19
	Load lat. (c)	12.30	54.73	5.55	2.81	2.85	2.69	2.79	4.40	3.48
	IPC	**1.83**	**0.29**	**3.99**	**3.71**	**3.57**	**3.15**	**3.52**	**2.02**	**2.46**
	Speedup (%)	**42.99**	**0.00**	**2.03**	**4.37**	**2.65**	**7.25**	**4.89**	**3.53**	**1.69**

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI IPC		**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Limit	DDIs rename (M)	493.08	740.37	223.97	894.94	586.71	499.00	49.51	115.33	239.12
	DDIs integrated (M)	135.39	131.26	48.37	147.48	104.95	128.93	29.74	42.65	82.24
	BMR lat. (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load lat. (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
Offline	DDIs renamed (M)	441.61	329.22	138.21	114.61	312.32	192.65	42.52	90.81	277.00
	DDIs integrated (M)	107.95	93.08	30.84	14.26	41.62	62.29	21.19	40.36	122.94
	BMR lat. (c)	16.20	50.69	19.08	11.73	14.09	9.12	11.28	5.53	31.34
	Load lat. (c)	3.07	19.75	3.94	3.43	2.82	3.29	3.15	2.57	3.55
	IPC	**2.78**	**0.65**	**1.86**	**2.60**	**2.77**	**2.40**	**4.61**	**3.12**	**1.94**
	Speedup (%)	**1.52**	**4.12**	**1.91**	**0.04**	**-0.30**	**8.75**	**1.62**	**15.47**	**15.54**
Online	DDIs renamed (M)	494.75	702.01	205.52	960.02	559.56	496.66	41.61	113.79	201.75
	DDIs integrated (M)	135.70	161.26	36.50	159.17	101.75	125.93	26.43	42.25	132.75
	BMR lat. (c)	15.94	40.79	18.19	10.32	13.30	8.14	9.86	6.20	31.87
	Load lat. (c)	3.05	17.28	3.83	3.38	2.77	2.91	3.13	2.50	3.49
	IPC	**2.79**	**0.71**	**1.88**	**2.65**	**2.80**	**2.46**	**4.67**	**3.05**	**1.97**
	Speedup (%)	**1.87**	**13.71**	**2.97**	**1.82**	**0.95**	**11.60**	**3.07**	**12.88**	**16.97**

5.3.4 Sensitivity to Underlying Processor Configuration

The base processor we have assumed so far is meant to represent an aggressive next generation machine. It was initially configured to resemble the size and shape of the since canceled Alpha 21464 [26], an 8-wide dynamically scheduled processor with large caches, a sophisticated branch predictor and support for 4 concurrent threads.

It is unclear whether future generation processors will copy this wide, large cache design. Wide superscalar designs are difficult to implement at high clock rates. Hence, microprocessors of the future may be narrower, pipelined much more deeply, or both. A combination of high clock rates and the desire to keep the common-case load latency to a few cycles may require the use of small, fast, low-associativity L1 data caches. Similarly, smaller branch predictor tables may be needed to allow branches to be predicted in a single cycle. High clock rates will also cause an increase in the relative latency to memory. L2 cache latency will increase similarly, unless smaller L2 caches are also employed. This shift in design philosophy is already evident in Intel's Pentium 4 [41], a microprocessor with a low-end clock frequency of 1.4GHz. The Pentium 4 is 3-wide, has a 4K-entry branch predictor, a deep pipeline and uses a direct-mapped, 8KB L1 data cache, and a 256KB L2 cache. The Pentium 4 also has a large L3 cache, which we do not model.

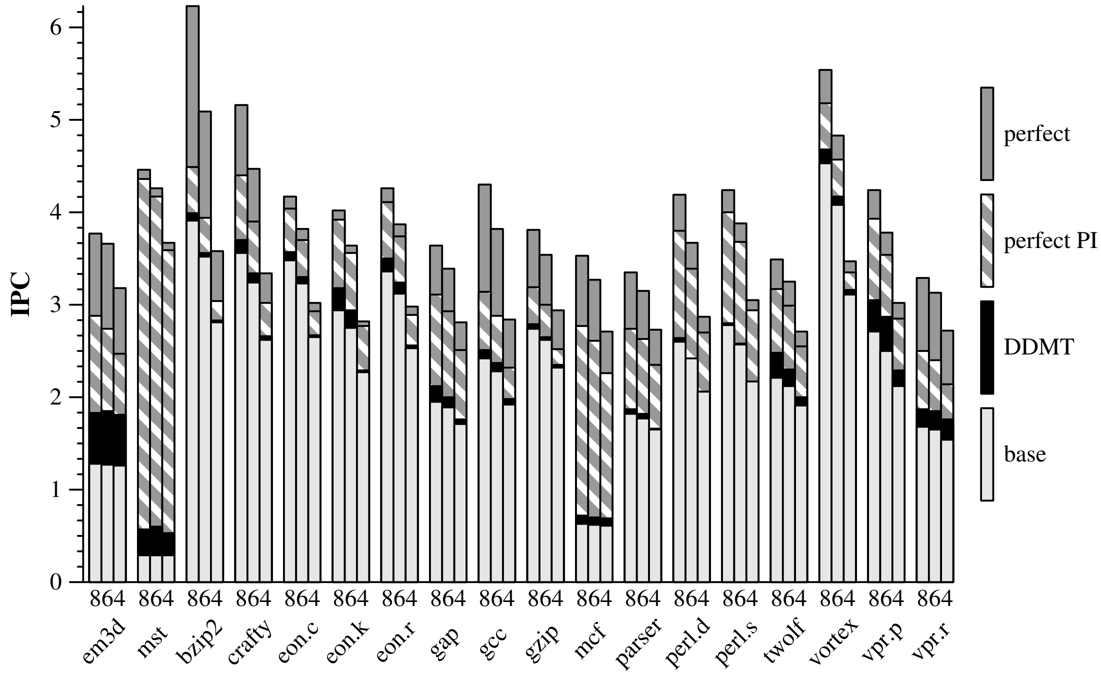
In this section, we examine DDMT's sensitivity to changes in the microarchitecture of the base processor. There are many possible changes to consider. In addition to processor width, cache size, and memory latency, we could explore the effects of larger and smaller instruction windows, deeper re-order buffers, and clustering in the microarchitecture. However, in the interest of space, we focus on the three initially discussed parameters as these are likely to have the most direct impact on DDMT. Processor width determines the master thread's sequencing constraints and level of resource consumption and, consequently, a DDT's sequencing advantage and the amount of bandwidth available for pre-execution. Cache and branch predictor size effect the number of cache misses and branch mis-predictions—PDIs—that may be targeted using pre-execution. L2, memory and scheduling latencies effect the cost of each individual PDI.

5.3.4.1 Processor Width

DDMT has a complex sensitivity to processor width, one which—for reasons given above—we have investigated only with processors narrower than the 8 of our central configuration. On one hand, a narrow processor imposes stricter fetch constraints on the master thread and makes the data-driven sequencing employed by DDTs relatively more powerful. On the other, relative bandwidth utilization is higher on narrower processors, making DDT overhead relatively larger as well.

In this section we explore the effect of DDMT on processors of widths 4 and 6 in addition to our central 8-wide configuration. All other processor parameters—the pipeline depth, amount of instruction buffering and the cache and branch predictor configurations—are fixed. Also fixed are the PI definition and the parameters of the DDT selection algorithm. The one microarchitectural parameter that

Figure 5.12 DDMT sensitivity to processor width.



we vary with width is the DDT sequencing injection policy. In order to keep injection bandwidth constant across experiments, we inject DDTs at processor width and the corresponding frequency. That is, to implement a DDT-1 injection policy on a 4-wide processor we inject DDTs 4 instructions at a time once every 4 cycles.

The results are shown in the graph in Figure 5.12 and in three supporting tables. With a new underlying processor configuration, all *five* levels of the implementation stack—including the baseline and the absolute upper bound (the *perfect* experiment)—must be re-evaluated. Rather than use five tables for each implementation level, we divide the evaluation into three tables, and evaluate a full implementation stack for each processor width in one table. Table 5.35 is the summary evaluation of the 8-wide processor, Table 5.36 evaluates a 6-wide processor, and Table 5.37 shows results for a 4-wide processor. We will use this kind of organization in the remainder of the section.

First, lets us comment on the comparative character of the baseline machines. Baseline IPC degrades by about 10% when processor width narrows from 8 to 6, and by an additional 15% or so when it narrows from 6 to 4. Programs whose IPC's on the 8-wide processor are already low—e.g., *em3d*, *mst* and *mcf*—don't experience significant changes. In general, a narrower processor is associated with a slightly higher average load latency and a slightly lower branch mis-prediction resolution latency. Again, these variations are slight.

Next, we are interested in the results of the *perfect* and *perfect PIs* experiments. In addition to performance and speedup, the tables summarize these two experiments in a single metric—*PDIs covered*. This single number, more so than the number of static PIs or the number of PIIs executed, characterizes the extent of the load and branch problem in the baseline machine and the coverage of the PI definition. As the tables show, processor width does not effect this metric.

We proceed to show a summary characterization of the selected DDTs. Although the PI definitions are identical, as are the cache and branch predictor configurations and the latencies of all operations, the DDTs selected for processors of different widths need not be identical. Recall, both processor width *and* the program's base IPC are inputs to the DDT selection algorithm. We combine them into a single metric which estimates the master thread's sequencing bandwidth consumption ($BW_{seq\text{-}CD}$). $BW_{seq\text{-}CD}$ is used in latency tolerance calculations

Table 5.35 DDMT on an 8-wide processor.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI	BMR lat. (c)	3.86	404.81	12.78	9.41	10.51	9.50	9.76	21.07	13.86
	Load lat. (c)	21.17	54.73	5.64	2.84	2.96	2.87	2.92	4.59	3.56
	IPC	**1.28**	**0.29**	**3.91**	**3.56**	**3.48**	**2.94**	**3.36**	**1.95**	**2.42**
Perfect	PDIs covered (M)	12.98	11.81	53.92	24.74	0.91	8.06	1.85	11.18	7.56
	IPC	**3.77**	**4.46**	**6.23**	**5.16**	**4.17**	**4.02**	**4.26**	**3.64**	**4.30**
	Speedup (%)	**194.74**	**1421.8**	**59.10**	**45.05**	**19.81**	**36.75**	**26.94**	**86.81**	**78.01**
Perfect PI	PDIs covered (M)	11.60	11.74	16.42	13.83	0.69	6.93	1.50	7.69	3.27
	IPC	**2.88**	**4.36**	**4.49**	**4.40**	**4.04**	**3.92**	**4.11**	**3.11**	**3.14**
	Speedup (%)	**125.16**	**1385.9**	**14.61**	**23.71**	**16.04**	**33.43**	**22.47**	**59.36**	**29.86**
	Speedup cov'g (%)	64.27	97.48	24.72	52.63	80.97	90.95	83.41	68.38	38.28
DDT	Avg. dyn. length	12.42	20.50	13.86	9.70	8.52	8.51	9.14	9.60	11.74
	Avg. dyn unroll	1.00	1.00	2.26	0.23	0.00	0.00	0.00	0.59	0.36
	PDIs covered (M)	9.65	11.03	9.87	6.19	0.37	3.69	0.84	3.23	1.79
DDMT	DDIs renamed (M)	42.73	47.22	184.74	249.73	15.29	109.06	31.66	186.31	76.47
	DDIs integrated (M)	25.05	34.29	57.59	109.18	5.48	44.28	10.83	58.20	14.45
	BMR lat. (c)	3.54	180.54	11.71	7.56	8.30	6.48	7.32	18.08	12.26
	Load lat. (c)	12.30	23.40	5.52	2.81	2.86	2.69	2.78	4.20	3.38
	IPC	**1.83**	**0.57**	**3.99**	**3.70**	**3.57**	**3.18**	**3.50**	**2.12**	**2.51**
	Speedup (%)	**42.99**	**93.83**	**2.02**	**3.86**	**2.56**	**8.21**	**4.30**	**8.51**	**4.05**
	Speedup cov'g (%)	22.08	6.60	3.41	8.57	12.91	22.34	15.96	9.80	5.19

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI	BMR lat. (c)	17.25	53.94	20.12	11.81	14.25	11.19	12.37	10.34	34.82
	Load lat. (c)	3.10	21.82	4.04	3.45	2.83	3.62	3.20	2.86	4.20
	IPC	**2.74**	**0.63**	**1.82**	**2.60**	**2.78**	**2.21**	**4.53**	**2.71**	**1.68**
Perfect	PDIs covered (M)	64.45	99.15	23.78	43.57	24.12	43.15	5.61	6.85	21.37
	IPC	**3.81**	**3.53**	**3.35**	**4.19**	**4.24**	**3.49**	**5.54**	**4.24**	**3.29**
	Speedup (%)	**38.85**	**462.68**	**83.50**	**61.13**	**52.69**	**58.03**	**22.16**	**56.61**	**95.48**
Perfect PI	PDIs covered (M)	33.58	89.67	15.95	35.26	21.20	35.65	3.26	5.47	14.79
	IPC	**3.19**	**2.77**	**2.74**	**3.80**	**4.00**	**3.17**	**5.18**	**3.93**	**2.50**
	Speedup (%)	**16.32**	**341.58**	**50.10**	**46.34**	**44.10**	**43.53**	**14.17**	**45.36**	**48.66**
	Speedup cov'g (%)	42.00	73.83	60.00	75.79	83.71	75.01	63.95	80.11	50.96
DDT	Avg. dyn. length	7.58	11.71	5.91	13.64	17.37	9.91	19.10	12.27	18.96
	Avg. dyn unroll	0.03	1.70	0.75	0.01	0.05	1.06	0.00	0.53	0.78
	PDIs covered (M)	12.59	62.06	7.61	26.50	11.88	26.20	1.62	4.00	8.51
DDMT	DDIs renamed (M)	493.08	740.37	223.97	894.94	586.71	499.00	49.51	115.33	239.12
	DDIs integrated (M)	135.39	131.26	48.37	147.48	104.95	128.93	29.74	42.65	82.24
	BMR lat. (c)	15.95	40.29	18.25	10.45	13.29	8.06	9.96	6.21	34.29
	Load lat. (c)	3.07	17.29	3.82	3.38	2.76	2.94	3.13	2.50	3.75
	IPC	**2.79**	**0.72**	**1.87**	**2.64**	**2.80**	**2.48**	**4.68**	**3.05**	**1.87**
	Speedup (%)	**1.82**	**15.13**	**2.38**	**1.72**	**0.97**	**12.34**	**3.24**	**12.69**	**11.07**
	Speedup cov'g (%)	4.68	3.27	2.85	2.81	1.84	21.26	14.60	22.42	11.60

Table 5.36 DDMT on a 6-wide processor.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI	BMR lat. (c)	3.79	403.57	12.25	9.37	9.88	9.03	9.23	20.65	13.62
	Load lat. (c)	20.84	54.72	5.55	2.90	2.98	2.91	2.94	4.61	3.60
	IPC	**1.27**	**0.29**	**3.52**	**3.24**	**3.23**	**2.75**	**3.12**	**1.89**	**2.28**
Perfect	PDIs covered (M)	12.98	11.81	53.94	24.75	0.90	7.98	1.84	11.17	7.55
	IPC	**3.66**	**4.26**	**5.09**	**4.47**	**3.82**	**3.64**	**3.87**	**3.39**	**3.82**
	Speedup (%)	**187.55**	**1354.1**	**44.79**	**37.96**	**18.29**	**32.36**	**24.03**	**79.71**	**67.52**
Perfect PI	PDIs covered (M)	11.58	11.75	16.42	13.99	0.68	6.89	1.46	7.65	3.24
	IPC	**2.74**	**4.17**	**3.94**	**3.90**	**3.70**	**3.56**	**3.74**	**2.93**	**2.88**
	Speedup (%)	**115.51**	**1323.6**	**11.96**	**20.53**	**14.51**	**29.62**	**19.79**	**55.45**	**26.65**
	Speedup cov'g (%)	61.59	97.75	26.70	54.09	79.33	91.53	82.38	69.57	39.47
DDT	Avg. dyn. length	2.45	11.00	9.91	6.20	7.63	6.00	5.01	7.09	7.52
	Avg. dyn unroll	1.00	1.00	1.65	0.24	0.00	0.00	0.00	0.53	0.32
	PDIs covered (M)	8.14	11.03	9.98	6.47	0.35	4.15	0.77	3.54	1.91
DDMT	DDIs renamed (M)	37.97	49.23	193.20	183.85	12.47	106.16	21.22	153.94	64.32
	DDIs integrated (M)	23.23	35.58	70.91	106.01	5.56	53.57	9.77	55.02	16.23
	BMR lat. (c)	3.41	174.56	11.24	7.39	7.54	5.57	6.62	17.92	11.74
	Load lat. (c)	11.92	22.68	5.41	2.85	2.87	2.68	2.81	4.27	3.43
	IPC	**1.85**	**0.60**	**3.56**	**3.34**	**3.30**	**2.94**	**3.24**	**2.00**	**2.37**
	Speedup (%)	**45.08**	**104.16**	**1.38**	**3.12**	**2.12**	**7.19**	**3.73**	**5.85**	**3.96**
	Speedup cov'g (%)	24.03	7.69	3.09	8.22	11.61	22.22	15.53	7.34	5.87

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI	BMR lat. (c)	16.53	53.66	19.73	11.31	13.83	10.96	11.95	10.25	33.87
	Load lat. (c)	3.12	21.79	4.06	3.49	2.89	3.66	3.20	2.93	4.23
	IPC	**2.62**	**0.62**	**1.77**	**2.42**	**2.57**	**2.12**	**4.08**	**2.50**	**1.65**
Perfect	PDIs covered (M)	64.44	99.15	23.72	43.54	24.10	43.13	5.65	6.85	21.32
	IPC	**3.54**	**3.27**	**3.15**	**3.67**	**3.88**	**3.25**	**4.83**	**3.78**	**3.13**
	Speedup (%)	**35.45**	**427.25**	**77.50**	**51.97**	**50.92**	**52.93**	**18.45**	**51.05**	**89.81**
Perfect PI	PDIs covered (M)	31.62	90.56	16.34	35.23	21.03	35.52	3.26	5.47	14.83
	IPC	**3.00**	**2.61**	**2.63**	**3.39**	**3.68**	**2.99**	**4.57**	**3.54**	**2.40**
	Speedup (%)	**14.71**	**320.58**	**48.13**	**40.23**	**43.28**	**40.64**	**12.01**	**41.69**	**45.49**
	Speedup cov'g (%)	41.51	75.03	62.11	77.40	84.99	76.79	65.06	81.66	50.66
DDT	Avg. dyn. length	9.10	5.66	3.84	11.43	9.58	4.27	13.73	6.14	10.64
	Avg. dyn unroll	0.03	1.33	0.67	0.01	0.00	0.59	0.00	0.53	0.81
	PDIs covered (M)	10.56	73.23	7.77	25.90	13.38	23.38	1.80	3.65	8.36
DDMT	DDIs renamed (M)	464.12	631.22	175.68	668.59	458.68	342.91	43.63	100.63	257.33
	DDIs integrated (M)	121.03	160.49	45.30	129.34	85.06	116.11	27.49	51.88	76.41
	BMR lat. (c)	14.92	38.83	17.66	10.35	12.22	7.93	10.92	4.44	32.84
	Load lat. (c)	3.11	17.12	3.80	3.43	2.81	3.01	3.14	2.45	3.70
	IPC	**2.65**	**0.70**	**1.82**	**2.38**	**2.58**	**2.30**	**4.17**	**2.87**	**1.85**
	Speedup (%)	**1.24**	**13.26**	**2.32**	**-1.36**	**0.33**	**8.44**	**2.22**	**14.71**	**12.07**
	Speedup cov'g (%)	3.50	3.10	2.99	-2.61	0.65	15.94	12.02	28.81	13.44

Table 5.37 DDMT on a 4-wide processor.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI	BMR lat. (c)	3.97	402.22	11.53	9.36	9.89	9.11	9.44	20.10	13.50
	Load lat. (c)	20.26	54.70	5.39	2.97	2.98	2.94	2.96	4.61	3.67
	IPC	**1.26**	**0.29**	**2.81**	**2.62**	**2.65**	**2.27**	**2.53**	**1.71**	**1.92**
Perfect	PDIs covered (M)	12.94	11.81	53.89	24.76	0.88	7.98	1.82	11.16	7.54
	IPC	**3.18**	**3.67**	**3.58**	**3.34**	**3.02**	**2.82**	**2.98**	**2.81**	**2.84**
	Speedup (%)	**153.32**	**1153.4**	**27.70**	**27.11**	**13.94**	**24.45**	**17.76**	**63.94**	**47.81**
Perfect PI	PDIs covered (M)	11.62	11.76	16.19	13.99	0.66	6.88	1.45	7.61	3.21
	IPC	**2.47**	**3.59**	**3.04**	**3.02**	**2.93**	**2.77**	**2.89**	**2.51**	**2.32**
	Speedup (%)	**96.73**	**1126.6**	**8.39**	**15.10**	**10.60**	**22.31**	**14.45**	**46.49**	**20.80**
	Speedup cov'g (%)	63.09	97.68	30.29	55.71	76.10	91.24	81.39	72.71	43.52
DDT	Avg. dyn. length	2.39	11.00	7.47	5.71	5.22	5.38	4.46	6.29	6.61
	Avg. dyn unroll	0.87	1.00	1.26	0.23	0.00	0.00	0.00	0.51	0.27
	PDIs covered (M)	7.97	11.04	10.33	5.80	0.36	4.85	0.81	3.52	2.03
DDMT	DDIs renamed (M)	40.20	46.92	162.82	126.88	7.60	95.28	16.58	109.67	50.54
	DDIs integrated (M)	23.80	36.69	68.75	78.92	4.72	42.03	9.09	60.10	19.02
	BMR lat. (c)	3.21	209.07	10.16	7.57	7.50	6.63	7.16	17.31	11.07
	Load lat. (c)	10.75	26.58	5.25	2.92	2.90	2.77	2.83	4.30	3.47
	IPC	**1.81**	**0.53**	**2.83**	**2.66**	**2.67**	**2.29**	**2.56**	**1.76**	**1.98**
	Speedup (%)	**43.78**	**81.61**	**0.93**	**1.44**	**0.75**	**0.96**	**1.37**	**2.62**	**3.17**
	Speedup cov'g (%)	28.55	7.08	3.36	5.31	5.39	3.93	7.71	4.09	6.63

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI	BMR lat. (c)	15.00	52.86	18.85	11.10	12.94	10.75	12.06	10.49	33.73
	Load lat. (c)	3.23	21.64	4.08	3.51	2.95	3.65	3.21	3.05	4.27
	IPC	**2.32**	**0.61**	**1.65**	**2.06**	**2.17**	**1.91**	**3.11**	**2.12**	**1.54**
Perfect	PDIs covered (M)	64.59	99.12	23.58	43.44	24.12	43.04	5.65	6.85	21.27
	IPC	**2.94**	**2.71**	**2.73**	**2.87**	**3.05**	**2.71**	**3.47**	**3.02**	**2.72**
	Speedup (%)	**26.51**	**346.75**	**65.67**	**39.02**	**40.33**	**41.87**	**11.41**	**42.49**	**76.41**
Perfect PI	PDIs covered (M)	18.87	91.32	16.21	34.67	20.85	35.45	3.24	5.46	14.80
	IPC	**2.52**	**2.26**	**2.35**	**2.70**	**2.94**	**2.55**	**3.35**	**2.85**	**2.14**
	Speedup (%)	**8.62**	**271.63**	**42.72**	**30.94**	**35.28**	**33.07**	**7.64**	**34.53**	**39.00**
	Speedup cov'g (%)	32.52	78.34	65.06	79.30	87.49	79.00	66.94	81.26	51.04
DDT	Avg. dyn. length	16.12	5.07	3.67	8.21	9.01	3.82	13.52	5.63	8.01
	Avg. dyn unroll	0.01	1.19	0.51	0.00	0.00	0.45	0.00	0.45	0.67
	PDIs covered (M)	2.49	61.84	7.82	25.01	14.26	23.66	1.95	3.36	8.91
DDMT	DDIs renamed (M)	233.36	610.52	150.57	419.25	370.80	220.64	43.60	83.10	223.81
	DDIs integrated (M)	138.48	165.52	59.81	155.66	82.42	109.46	29.18	48.06	143.79
	BMR lat. (c)	13.47	37.61	16.84	9.23	11.35	7.50	9.66	4.31	28.33
	Load lat. (c)	3.16	17.01	3.75	3.45	2.86	3.07	3.14	2.54	3.41
	IPC	**2.34**	**0.69**	**1.66**	**2.06**	**2.14**	**2.00**	**3.16**	**2.29**	**1.76**
	Speedup (%)	**0.69**	**13.34**	**0.72**	**-0.19**	**-1.57**	**4.66**	**1.61**	**7.77**	**14.39**
	Speedup cov'g (%)	2.60	3.85	1.10	-0.48	-3.89	11.13	14.11	18.28	18.83

where we use it to approximate the master thread's sequencing constraint, and thereby a DDT's sequencing advantage. The ratio of $BW_{seq\text{-}CD}$ to the width of the machine is also used in overhead estimations as a master thread utilization factor. Certainly, changing this one metric can have a large impact on the character of DDTs selected.

A narrower processor results in a lower $BW_{seq\text{-}CD}$ as both processor width and the particular program's base IPC drop. In general, a lower $BW_{seq\text{-}CD}$ will result in shorter DDTs that employ unrolling less aggressively. With a lower $BW_{seq\text{-}CD}$ value, the DDT selection algorithm sees the master thread as extremely sequencing constrained, making data-driven sequencing relatively that much more powerful. On a narrower processor, every master thread instruction skipped in a DDT has a proportionately larger impact on latency tolerance. Since fewer master thread instructions must be skipped to allow a DDT to achieve a certain sequencing advantage, shorter DDTs are needed. This effect is dampened somewhat in DDTs that attack long L2 cache misses, as hiding the latency of the operation may require a high sequencing advantage even on a narrower machine. Overhead also contributes to keeping DDTs short. Since a narrower processor is usually more highly utilized by the master thread, the cost of longer DDTs is correctly seen as relatively higher by the DDT selection algorithm.

The tables show exactly these trends; average DDT length and unrolling degree are lower in the narrow processor cases. The long exception is *gzip*. In *gzip*, a narrower machine results in longer (albeit less unrolled) DDTs. On a wider machine, the DDT selection algorithm fails to create DDTs for certain

PDIs, perceiving the DDT's sequencing advantage as insufficient to achieve a certain level of latency tolerance within the size and scope restrictions of the algorithm. However, on a narrower machine the DDT's perceived sequencing advantages grows. In *gzip*'s case, it grows to the point where DDTs with sufficient latency tolerance can now be found within the search constraints. These DDTs will be long as they just barely "fit" under one or more of the size, scope and unrolling constraints.

Finally, we show the results of our DDMT experiment. With shorter DDTs, a narrower processor renames fewer DDT instructions. This is a general observation, not a rule, as while DDTs may be shorter, there may be more of them. Surprisingly perhaps, while fewer DDT instructions are renamed on the narrower machine, in general more are integrated. Not only are shorter DDTs less likely to have difficult to integrate regions, they are more likely to be control-equivalent with their trigger instruction. Latency reductions—both load and branch resolution—are similar on processors of different widths. This is not a surprise as the latency tolerance criteria of the DDT selection algorithm were fixed.

Ultimately, DDMT's performance impact on narrower machines is slightly lower than it is on wider machines, as higher processor utilization by the master thread makes DDMT relatively more expensive. In fact, while no program experience a DDMT induced slowdown on an 8-wide processor, *perl.d* does on a 6-wide processor, and both *perl* benchmarks do on the 4-wide processor. Performance impact *can* be relatively higher on a narrower machine as the result of shorter, more efficient DDTs enabled by DDMT's increased sequencing advantage. However, the important point is that performance impact relative to potential performance impact—i.e., *speedup coverage*—is stable if not slightly increasing due to the "compressed nature" of the narrower processors' performance stack.

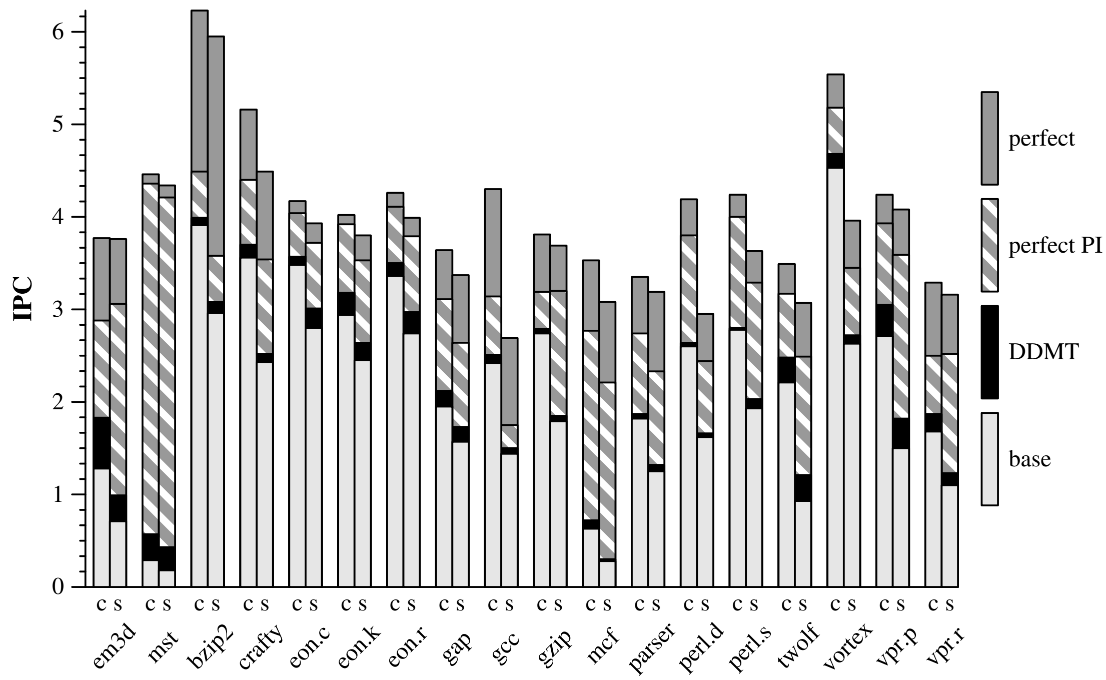
On a closing note, in Section 5.2.5.4, we claimed that while DDMT overhead is negligible on a wide processor its effects would be felt more strongly on a narrower machine. While we still believe this to be the case, results for an "overhead-less" DDMT experiment are not shown. The flaw in our "overhead-less" model is magnified on a narrower processor.

5.3.4.2 Cache and Branch Predictor Table Size

As seen in our functional benchmark characterization (Table 5.5), most of the benchmarks we study have low cache miss rates for our central, large-cache configuration. This is especially true for L2 misses, which are effectively zero in many programs. Branch mis-predictions are also relatively infrequent—although less so when compared to cache misses—as the baseline predictor has extremely large, 16K-entry tables.

For our second experiment in our base processor sensitivity analysis, we measure the performance of DDMT on a processor with smaller caches and a smaller branch predictor. In this new configuration, we cut the size of the L1 data cache, the L2 cache and the branch predictor tables by a factor of 8. The branch predictor algorithm and organization and the cache block size remain the same. However, we use a direct-mapped L1 data cache, rather than one with an associativity of 2. The direct-mapped, 8KB L1 data cache matches that of the Pentium 4. The

Figure 5.13 DDMT sensitivity to cache and branch predictor size.



branch predictor tables and the 128KB L2 cache are a factor of 2 smaller than those used in the Pentium 4. The purpose of this experiment is not to model any single future processor carefully—the Pentium 4 is a 3-wide machine whereas our base processor is an 8-wide machine—but to gauge the effectiveness of DDMT in scenarios with far larger numbers of cache misses and mis-predicted branches. As in the previous section, we keep all other parameters, including PI definition and DDT selection algorithm configuration, constant.

We compare the performance stack of this, smaller cache/predictor-table processor configuration (*s*) with that of our central configuration (*c*) in the graph in Figure 5.13. A hierarchical evaluation of the new configuration is presented in Table 5.38. Again, the parallel evaluation of the central configuration is in

Table 5.38 DDMT on a processor with smaller caches and branch predictor tables.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI	BMR lat. (c)	3.73	864.46	16.36	10.88	10.86	10.36	10.44	20.95	14.44
	Load lat. (c)	46.46	61.05	8.83	4.08	3.58	3.49	3.52	5.68	5.42
	IPC	**0.71**	**0.18**	**2.96**	**2.43**	**2.80**	**2.45**	**2.74**	**1.57**	**1.44**
Perfect	PDIs covered (M)	15.62	12.26	78.73	142.36	4.75	25.20	7.61	32.27	18.14
	IPC	**3.76**	**4.34**	**5.95**	**4.49**	**3.93**	**3.80**	**3.99**	**3.37**	**2.69**
	Speedup (%)	**428.75**	**2309.6**	**100.95**	**84.59**	**40.38**	**54.94**	**45.76**	**114.07**	**86.14**
Perfect PI	PDIs covered (M)	14.17	12.16	25.23	86.14	3.56	18.28	5.84	18.85	5.55
	IPC	**3.06**	**4.21**	**3.58**	**3.54**	**3.72**	**3.53**	**3.79**	**2.64**	**1.75**
	Speedup (%)	**329.96**	**2237.2**	**21.12**	**45.66**	**32.92**	**44.15**	**38.37**	**67.56**	**21.44**
	Speedup cov'g (%)	76.96	96.87	20.92	53.98	81.54	80.37	83.85	59.23	24.88
DDT	Avg. dyn. length	2.00	10.74	8.01	6.61	5.34	6.09	5.65	7.74	7.03
	Avg. dyn unroll	1.00	0.98	0.83	0.11	0.04	0.04	0.11	0.33	0.56
	PDIs covered (M)	10.86	11.42	8.89	39.11	2.73	13.61	4.49	10.75	2.80
DDMT	DDIs renamed (M)	53.23	50.72	184.03	707.56	38.22	255.93	70.90	364.10	85.97
	DDIs integrated (M)	25.28	34.65	47.28	224.74	11.45	63.17	20.20	88.12	18.22
	BMR lat. (c)	3.42	231.53	13.38	9.07	8.04	7.43	7.23	16.86	12.87
	Load lat. (c)	24.96	24.38	8.69	3.86	3.15	3.07	3.06	5.09	5.17
	IPC	**0.99**	**0.43**	**3.08**	**2.52**	**3.01**	**2.64**	**2.97**	**1.73**	**1.50**
	Speedup (%)	**38.73**	**137.50**	**4.00**	**3.56**	**7.45**	**7.75**	**8.47**	**9.77**	**3.92**
	Speedup cov'g (%)	9.03	5.95	3.96	4.21	18.44	14.11	18.50	8.56	4.55

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI	BMR lat. (c)	29.26	76.22	29.61	13.50	15.64	29.28	11.33	22.32	58.45
	Load lat. (c)	5.30	40.57	6.94	4.32	3.56	14.33	4.12	6.88	6.82
	IPC	**1.79**	**0.28**	**1.25**	**1.62**	**1.93**	**0.93**	**2.63**	**1.50**	**1.10**
Perfect	PDIs covered (M)	140.99	110.94	47.10	125.87	108.89	67.35	40.88	15.35	47.08
	IPC	**3.69**	**3.08**	**3.19**	**2.95**	**3.63**	**3.07**	**3.96**	**4.08**	**3.16**
	Speedup (%)	**106.81**	**987.90**	**155.34**	**82.48**	**88.56**	**230.81**	**50.44**	**171.83**	**188.10**
Perfect PI	PDIs covered (M)	116.30	98.52	31.55	73.77	82.83	54.67	25.86	13.43	39.33
	IPC	**3.20**	**2.21**	**2.33**	**2.44**	**3.29**	**2.49**	**3.45**	**3.59**	**2.52**
	Speedup (%)	**79.04**	**680.90**	**86.44**	**51.24**	**70.76**	**168.61**	**31.09**	**139.20**	**129.82**
	Speedup cov'g (%)	74.00	68.92	55.64	62.12	79.90	73.05	61.63	81.01	69.02
DDT	Avg. dyn. length	8.44	8.45	5.25	7.62	8.44	5.98	11.97	6.31	10.45
	Avg. dyn unroll	0.13	2.47	0.45	0.05	0.00	1.07	0.01	0.53	0.50
	PDIs covered (M)	44.05	79.15	13.91	52.08	57.34	37.93	18.54	8.18	13.85
DDMT	DDIs renamed (M)	960.41	720.61	365.90	1142.4	1079.8	561.86	411.02	147.80	388.55
	DDIs integrated (M)	188.55	161.96	82.07	213.58	273.13	149.95	143.12	52.82	108.93
	BMR lat. (c)	27.41	67.41	26.12	11.80	12.95	20.84	9.92	13.26	57.07
	Load lat. (c)	4.74	36.42	6.47	4.06	3.28	9.46	3.76	5.34	5.82
	IPC	**1.85**	**0.30**	**1.32**	**1.66**	**2.03**	**1.21**	**2.72**	**1.82**	**1.23**
	Speedup (%)	**3.42**	**5.45**	**6.09**	**2.43**	**5.21**	**30.00**	**3.37**	**21.17**	**12.01**
	Speedup cov'g (%)	3.20	0.55	3.92	2.94	5.89	13.00	6.69	12.32	6.39

Table 5.35.

Table 5.5 showed that the smaller cache and branch predictor configuration produces a significant number of additional branch mis-predictions, L1 cache misses, and L2 cache misses. As the graph and Table 5.38 show, these additional misses result in significantly higher average load latencies and significantly lower baseline performance. The branch mis-prediction resolution latency (BMR) also rises, but less significantly. This rise is due the increased cache miss incidence in mis-predicted branch computations.

With a fixed PI definition, the higher miss and mis-prediction rates result in higher numbers of PDIs. This increase is due both to a larger number of PDIs per PI and the addition of new PIs which now meet the definition's thresholds.

The character of the chosen DDTs is also affected by the underlying processor configuration. However, in contrast with processor width which affects DDT selection in one main way, multiple effects are at work here. The effect we saw in the processor width case—the lower $BW_{seq\text{-}CD}$—is also present here. Although processor width is fixed, base IPC is much lower relative to the central configuration. We have seen that a lower $BW_{seq\text{-}CD}$ results in shorter DDTs. However, here the shorter DDT effect is less pronounced since overhead considerations are lower due to our use of a wide machine. Counter-acting this first effect, is the fact that PDIs have a higher average level of desired latency tolerance, LT_{des}. In an aggregate sense, this is due to the increased presence of L2 cache misses. However, LT_{des} increases even for L1 cache miss and branch mis-prediction PDIs due to an increased incidence of cache misses in their computations. A higher LT_{des} tends to create longer, more highly unrolled DDTs, as these are needed to hide the additional latency. The net change in the character of chosen DDTs reflects the stronger of these two effects. In *gcc* and *mcf*, the average PDI latency increase is higher than the gain in sequencing advantage resulting in higher levels of unrolling. In *bzip2* and *crafty*, the sequencing advantage is higher resulting in shorter DDTs.

The relative character of the chosen DDTs determines the runtime level of pre-execution activity. Programs for which longer DDTs were chosen typically execute and integrate more DDT instructions. However, this is not always the case here. With more PDIs to cover and more PDIs covered by the chosen DDTs, more total instructions can pre-execute and be integrated as a result of the increased number of DDT, while each individual DDT is shorter. This was not the case in the processor-width experiment where the number of PDIs effectively stayed constant. Absolute reductions in BMR and load latencies are larger in the smaller cache configuration. Relative reductions vary and performance impact varies with them. In *twolf*, for instance, relative reductions increase from about 20% to about 35%. Consequently, DDMT's performance impact in *twolf* increases from a 12% improvement to a 30% improvement. Most benchmarks, in fact, see substantial increases in performance impact. None are as large in an absolute sense as the one experience by *twolf*, but several benchmarks see the impact of DDMT double and more. In *parser*, for instance, speedup jumps from 2% to 6%, and in vpr.p from 12% to 21%. In a few programs, DDMT's impact is reduced. *Mcf* is the primary example with performance improvement dipping from 15% to 5%. In *mcf*, the constraints of the DDT selection algorithm—which we left fixed—prevent the chosen DDTs from dealing effectively with the increased latency of the PDI computations which contain many cache misses. The average dynamic unrolling degree for *mcf* in the small cache experiment is 2.47. With a maximum unrolling degree of 4, this means that many dynamic DDTs are unrolled to the maximum allowed level. Other benchmarks like *em3d* and *gcc* also see a decline in DDMT's effectiveness, although this decline is less than 10% (relative).

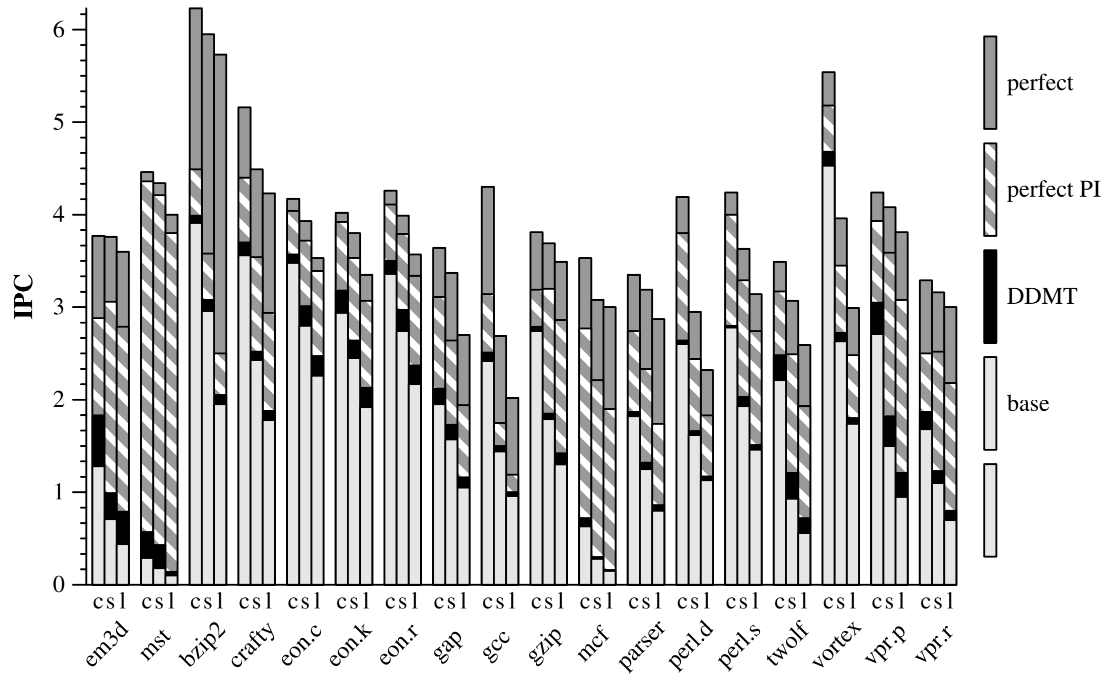
Experimenting with narrower processors, we saw DDMT's absolute impact decline but its efficiency—speedup coverage or impact relative to potential impact—rise. The opposite is true here. With lower baseline performance, the "performance potential stack" expands, leading to lower relative efficiencies although absolute gains improve.

Before we proceed, we must make a point about our experiences with different base processor configurations. Earlier, we mentioned that DDMT's effectiveness on *mcf* degrades as the DDT selection algorithm becomes actively constrained by the allowed unrolling degree. Presumably, better performance could be obtained by unconstraining this parameter. While we have experimented with several base processor configurations, we have not performed the sensitivity analysis required to tune DDMT for each of these configurations. For instance, we have seen that our central configuration is relatively performance insensitive to the PI definition. However, our central configuration also had relatively few PDIs. If more PDIs are present as they are in this case, it may be advantageous to narrow the PI definition, or perhaps to expand it. These and other investigations are fertile ground for future work.

5.3.4.3 L2 Cache, Memory and Branch Resolution Latencies

For the final experiment of this portion of the sensitivity analysis, we measure DDMT's sensitivity to PDI latency. Smaller caches and branch predictor tables increase the PDI population. A faster clock and deeper pipelining increase the latency of load and branch PDIs, respectively. Breaking with our tradition of investigating a single change from our central design point, we model these additional latencies on top of the small cache and branch predictor configuration introduced in the previous section. We do this because our central configuration does not experience enough L2 cache misses to expose memory latency. We increase the L2 cache latency from 6 to 12 cycles, increasing the effective L2 cache hit time from 10 to 16 cycles. We increase memory latency from 70 to 140 cycles. Branch mis-prediction resolution latency is increased by increasing pipeline depth. Adding depth at any logical pipeline stage suffices to increase resolution latency. However, because DDMT uses register integration as its mis-prediction resolution mechanism, only the additional stages after register renaming can be attacked by DDMT. Our central configuration has a “scheduling latency” of 3 cycles—we model a single cycle scheduler and 2 cycles to read the physical register file. This latency is observed only upon entry to the instruction window. It is not added to the program's critical path as the scheduler can schedule dependent instructions in consecutive cycles. In this new configuration, we increase the scheduling latency to 6 cycles. The Pentium 4 also has a 6 cycle scheduling latency.

Figure 5.14 DDMT sensitivity to cache, memory and branch resolution latencies.



The performance of DDMT in this new configuration is shown in Figure 5.14. The new configuration is called *slow (l)* and is the right bar in each group. The two other bars are our *central (c)* configuration and the *small (s)* configuration from the previous section. The hierarchical evaluation of the slow configuration is presented in Table 5.39.

The increased pipeline depth and cache and memory latencies cause the average branch mis-prediction resolution (BMR) and load latencies to rise by about 80% from their *small (s)* configuration values. Performance degrades about 40% from the same, stretching the "performance potential stack" even further as most benchmarks have difficulty achieving IPCs much greater than 1.

At the same time, the PDI coverage of the PI definition expands only slightly.

Table 5.39 DDMT on a processor with smaller caches and longer memory latencies.

		em3d	mst	bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc
RI	BMR lat. (c)	8.15	1656.9	25.92	17.37	16.21	15.58	15.80	33.55	24.16
	Load lat. (c)	75.96	115.87	14.33	5.42	4.34	4.27	4.24	8.33	7.93
	IPC	**0.44**	**0.10**	**1.95**	**1.78**	**2.26**	**1.92**	**2.17**	**1.05**	**0.96**
Perfect	PDIs covered (M)	15.60	12.26	78.12	142.64	4.76	25.10	7.60	32.40	18.17
	IPC	**3.60**	**4.00**	**5.73**	**4.23**	**3.53**	**3.35**	**3.57**	**2.70**	**2.02**
	Speedup (%)	**718.08**	**4115.7**	**193.48**	**137.37**	**56.41**	**73.96**	**64.23**	**157.33**	**109.44**
Perfect PI	PDIs covered (M)	15.49	12.16	32.78	89.63	4.22	20.58	6.47	19.90	5.87
	IPC	**2.79**	**3.80**	**2.50**	**2.94**	**3.39**	**3.07**	**3.34**	**1.94**	**1.19**
	Speedup (%)	**533.87**	**3901.4**	**28.37**	**64.69**	**50.19**	**59.56**	**53.58**	**84.51**	**23.93**
	Speedup cov’g (%)	74.35	94.79	14.66	47.09	88.98	80.53	83.41	53.72	21.87
DDT	Avg. dyn. length	2.71	12.21	5.86	6.87	5.98	6.80	6.79	8.89	8.27
	Avg. dyn unroll	1.24	1.47	0.33	0.19	0.03	0.06	0.11	0.42	0.41
	PDIs covered (M)	12.62	11.42	14.65	38.60	3.09	14.98	4.43	9.97	2.64
DDMT	DDIs renamed (M)	63.61	53.62	443.59	862.81	55.11	324.97	98.32	471.26	102.39
	DDIs integrated (M)	32.79	26.29	77.57	249.12	13.20	65.39	20.12	87.56	18.03
	BMR lat. (c)	2.89	893.46	19.86	14.68	12.66	11.16	11.91	27.91	21.85
	Load lat. (c)	25.47	63.83	14.14	5.17	3.60	3.63	3.66	7.51	7.60
	IPC	**0.79**	**0.14**	**2.05**	**1.88**	**2.47**	**2.13**	**2.37**	**1.16**	**1.00**
	Speedup (%)	**78.74**	**49.05**	**5.18**	**5.25**	**9.37**	**10.63**	**9.06**	**10.38**	**4.13**
	Speedup cov’g (%)	10.96	1.19	2.68	3.82	16.62	14.37	14.10	6.60	3.77

		gzip	mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
RI	BMR lat. (c)	46.69	139.70	49.07	21.41	23.32	52.01	19.36	39.44	98.26
	Load lat. (c)	7.68	74.31	11.23	6.06	4.36	25.82	5.31	11.36	10.69
	IPC	**1.30**	**0.15**	**0.80**	**1.13**	**1.46**	**0.56**	**1.74**	**0.95**	**0.70**
Perfect	PDIs covered (M)	141.40	111.02	47.20	125.74	109.13	67.59	40.88	15.36	47.12
	IPC	**3.49**	**3.00**	**2.87**	**2.32**	**3.14**	**2.59**	**2.99**	**3.81**	**3.00**
	Speedup (%)	**168.70**	**1845.6**	**259.54**	**105.13**	**115.55**	**364.61**	**71.48**	**299.94**	**327.56**
Perfect PI	PDIs covered (M)	116.38	99.54	32.24	84.18	87.06	57.17	27.28	13.90	39.56
	IPC	**2.86**	**1.90**	**1.74**	**1.83**	**2.74**	**1.93**	**2.48**	**3.08**	**2.18**
	Speedup (%)	**120.17**	**1133.8**	**117.51**	**61.56**	**88.03**	**245.52**	**42.48**	**223.13**	**210.78**
	Speedup cov’g (%)	71.23	61.43	45.28	58.56	76.18	67.34	59.43	74.39	64.35
DDT	Avg. dyn. length	8.43	6.80	5.92	8.15	9.23	6.50	11.23	6.05	11.12
	Avg. dyn unroll	0.22	1.43	0.59	0.06	0.16	0.99	0.06	0.57	0.47
	PDIs covered (M)	39.71	23.22	12.03	47.35	55.04	30.41	18.91	7.40	11.66
DDMT	DDIs renamed (M)	1214.2	381.37	461.47	1553.8	1138.2	581.66	434.01	164.27	418.08
	DDIs integrated (M)	188.89	97.88	80.42	203.31	248.94	143.76	150.91	54.99	83.15
	BMR lat. (c)	40.52	134.26	42.97	18.89	20.84	39.60	17.28	22.65	94.00
	Load lat. (c)	6.55	68.68	10.46	5.83	4.13	18.68	4.78	8.87	9.06
	IPC	**1.42**	**0.16**	**0.86**	**1.17**	**1.51**	**0.72**	**1.80**	**1.21**	**0.80**
	Speedup (%)	**9.18**	**3.11**	**7.65**	**3.64**	**3.87**	**29.55**	**3.29**	**26.71**	**14.03**
	Speedup cov’g (%)	5.44	0.17	2.95	3.46	3.35	8.10	4.60	8.90	4.28

Recall, our central PI definition used minimum problem latency thresholds for both loads and branches. No new PIs are admitted by this definition by sole virtue of the increase in their problem latencies.

The character of selected DDTs is also predictable from trends we have observed previously. Longer latencies require longer DDTs and higher degrees of unrolling to hide them. In most benchmarks, such longer DDTs are, in fact, chosen. However, when very long serial latencies must be tolerated, as is the case in *mst* and *mcf*, a second effect is activated. Specifically, the desired levels of latency tolerance of serial loads are so high, that within the slicing window and size constraints no DDTs can be found that tolerate a sufficient fraction of this latency as mandated by the latency tolerance acceptability threshold (LCAF). DDTs that tolerate latency at sub-LCAF levels are rejected. The increased minimal latency tolerance demands and the upper bound constraints of the DDT selection algorithm effectively “cross” one another, resulting in an empty set of DDTs for a given set of PDIs. This “cross-over” effect is a pathology in our current implementation which degrades its robustness under extreme conditions. Fixing this pathology is a straightforward piece of future work. To find DDTs that can tolerate very long latencies with our current implementation, either the scope and size constraints of the DDT selection algorithm must be relaxed, or the LCAF must be lowered so that it admits as much partial latency tolerance as possible. We have not performed the sensitivity analysis to measure the validity of this hypothesis.

DDMT performance impacts track those observed in the small cache configuration well. In general, performance improvement stays steady or even increases slightly as the chosen DDTs tolerate more absolute latency than those chosen for the lower latency *small (s)* configuration. Exceptions to this general rule are programs, like *mst* and *mcf*, which experience the counter-productive "cross-over" effect described in the previous paragraph. On the other hand, speedup coverage is low, again due to the depressed baseline IPCs and the relatively high potential improvement (*perfect* or *perfect PIs*) associated with them.

5.4 Chapter Summary

This chapter presented a simulation-driven performance evaluation of DDMT. We performed a detailed analysis of a single DDMT design point, attributing performance and overhead to each component of the system. We follow with a sensitivity analysis, measuring the impact of changes to DDMT-specific components, the DDT selection algorithm, the relationship between DDMT's setup and runtime phases, and the configuration of the base microprocessor.

The results of this broad but preliminary performance evaluation are encouraging. Added to an aggressive, wide, superscalar processor with large caches and a large branch predictor, DDMT produces large speedups on challenging microbenchmarks and performance improvements in the 10% to 15% range on integer benchmarks. These performance improvements are 20% to 35% of the improvements obtained by "perfecting" (idealizing) the targeted instructions.

Our sensitivity analysis showed that DDMT is sensitive to only a few pre-execution specific microarchitectural parameters, most notably IT associativity and the load integration suppression mechanism. DDMT's static finite DDTs and its exclusion of trigger chaining result in a “naturally-throttled” pre-execution model that suppresses resource contention. Consequently, DDMT is insensitive to most bandwidth and resource variations.

DDT selection can effect DDMT performance by covering varying numbers of PDIs and achieving different levels of latency tolerance per PDI covered. Both quantities (and hence DDMT performance) are particularly sensitive the estimate of the master thread’s sequencing rate. This value determines the perceived DDT sequencing advantage and controls the size and latency tolerance of the selected DDTs. DDT size and unrolling constraints and latency tolerance requirements also play roles in DDT selection and DDMT performance.

One extremely positive result is that both the PI definition and the DDTs that pre-execute the PDIs of chosen PIs are largely independent of the program sample used to acquire them. This implies that DDMT’s setup component may be implemented in any number of ways, including offline using different input sets.

DDMT’s performance impact grows when we vary the underlying microarchitecture to model characteristics of likely future processors. These characteristics—including smaller caches, deeper pipelines, and longer perceived memory latencies—are all driven by increasing clock frequency. Smaller caches and branch predictor tables create more PDIs, while longer pipelines and cache and memory latencies increase the cost of each individual PDI. With more work to do and lower master thread processor utilization, DDMT’s performance gains double for certain programs. This doubling occurs without tuning the DDMT microarchitectural components or the selection algorithm for these new conditions.

Chapter 6

Related Work

My work on pre-execution and DDMT was inspired and influenced by prior and contemporaneous work of many others on various realized and proposed systems. I would also like to think that it has influenced a number of subsequent research initiatives and has played a part in popularizing this research direction. In this chapter, I describe pre-execution's influences—both backwards and forwards—and point out the similarities and differences. The chapter is organized into four sections. The first describes pre-execution's development from my research perspective, beginning with simple, domain-specific mechanisms and slowly generalizing to DDMT. The last three describe prior, contemporaneous and subsequent efforts by others.

6.1 Pre-Execution: My Retrospective

DDMT is the last of my four incarnations of pre-execution. Early incarnations were simple, highly specialized mechanisms for attacking specific classes of problem instructions. They gradually evolved into DDMT, the general purpose Swiss-

Army-knife of pre-execution. This section is my retrospective on this process. In hindsight, pre-execution was as much a product of other research going on in the group at the time as it was of almost anything else.

6.1.1 Dependence-Based Prefetching for Linked Data Structures

Dependence based prefetching (DBP) [68] was my initial foray into pre-execution. It was inspired by Luk and Mowry's work on compiler prefetching for linked—i.e., pointer-based—data structures [55]. Recognizing that prefetch addresses for linked data structures (LDS) could not be guessed arithmetically, the initial aim was to create a dynamic hardware version of Luk and Mowry's compiler-based greedy prefetching algorithm. In the software version, the compiler recognized instructions that accessed pointers and then scheduled copies of those instructions immediately after the instruction that loaded the corresponding pointer address. The idea was to create an analogous hardware mechanism that recognizes loads that load pointer values and loads that dereference pointer values and then to use dynamic instances of the former to trigger scheduling of copies of the latter. The argument was that decoupling and the lack of sequencing overhead would give the hardware version better latency tolerance and performance characteristics than its software counterpart. The term dependence-based prefetching was coined to distinguish this early form of pre-execution from address-based prefetching.

Implementing DBP—at least in a simulator—was easy. My office-mate at the time, Andreas Moshovos, knew all about address dependences and mechanisms for capturing and representing them. The dependence predictor used was a reverse—forward rather than backward—version of one of his post-retirement buffer-based designs [59]. The predictor acted in conjunction with a small queue of prefetch addresses. The predictor was accessed on the completion of every load. If the load matched as a pointer producer, dedicated address generators used the loaded values to compute prefetch addresses of the pointer consumer loads and enqueued those addresses onto the prefetch queue. Asynchronous prefetching—i.e., pre-execution—was implemented by allowing completed prefetches to access the predictor and spawn other prefetches as well.

6.1.2 Dependence-Based Pre-Computation for Virtual Function Calls

The simplicity and success of dependence based prefetching led me to look for other applications of the same technique. Pre-computation of chronically mis-predicted branches was an obvious choice, but required extreme generalizations to the dependence detection and pre-execution mechanisms. The simplicities of the pointer chasing idiom—i.e., its exclusive use of loads which implied a single input and a single trigger for every pre-executed instruction—allowed dependence based prefetching to use an extremely simple mechanism. Expanding this mechanism to encompass computations of general shape was a difficult task. Fortunately, one class of pathologically mis-predicted control-transfers was idiomatically similar to pointer-chasing. The target addresses of virtual functions are computed by following pointers from objects to virtual function tables, allowing pre-computation of these targets using an unmodified version of the dependence based prefetch mechanism [69].

Pre-computation of virtual function addresses did require one component that dependence-based prefetching did not. In cache—i.e., non-binding—prefetching, there is no need for a mechanism to correlate a given prefetch with a given load. That function is performed implicitly and perfectly by the cache. In target pre-computation, however, a mechanism that correlates a pre-computed target with its intended dynamic function call is crucial as mis-correlated results are useless—if not harmful—results. The solution I found relied on control-based interleaving invariants. The observation is that instructions that trigger pre-computations and their intended dynamic virtual function calls form interleaved sequences. A simple sequence numbering scheme could divide the dynamic stream into epochs separated by virtual function call instances. A virtual function call always used a prediction that was triggered by an instruction from the immediately older epoch. A more robust formulation of this same solution was later found by Zilles, who called epochs valid regions [106]. As it turns out, the key insight is that with some static analysis and statically-generated hints, control-flow in the master thread can be used to manage epochs with extreme accuracy.

6.1.3 A Speculative Dataflow Processor

Following the work on dependence-based prefetching and pre-computation, the next step was to create a pre-execution system that could handle all problem instructions and hence computations of all shapes. The internal static representation of inter-instruction dependences used by the first two systems was similar—not coincidentally of course—to the static program representation used by dataflow machines [5, 24, 42, 46, 63, 74, 75, 89, 91, 98]. And, of course, the sequencing model was data-driven. I argued that, since a dataflow machine could theoretically handle arbitrary computations, it should be possible to create a small speculative dataflow machine, interface it with the main superscalar processor and use it to execute generalized computations learned by a generalized dependence detection circuit.

Speculative dataflow [72] worked beautifully. The erstwhile prefetch queue—which circularly fed back to itself via the memory system and the dependence predictor was the perfect analog of the token queue used in dataflow machines. Replaced with a generalized work (token) queue, it provided the perfect interface point between the main superscalar processor and the speculative dataflow processor. The dataflow component—a small, simplified implementation of the tagged-token dataflow architecture (TTDA) [5]—meshed naturally with the superscalar processor's dynamic scheduler.

It was at this time that register integration was also first conceived. Inspiration for register integration came from two directions. The first was a desire to fully mesh the dataflow processor with the out-of-order execution engine of the superscalar processor—this implied sharing the physical register file. The second was a performance-driven desire to avoid re-execution of pre-executed results by adapting Avinash Sodani's instruction reuse [82].

6.1.4 DDMT

The speculative dataflow processor had two problems. First, dynamically learning and internally representing computations of arbitrary shapes was a much more difficult task than I initially realized. Even the unreasonably complicated dependence detection machinery we employed could only recognize computations with simple structure. The dual superscalar/dataflow organization of the processor was perceived as unduly complex and unattractive from an implementation standpoint.

Speculative data-driven multithreading (DDMT) [71, 73] was conceived to address these concerns. Rather than dataflow, DDMT was initially designed as an extension to SMT which seemed like a probable microarchitecture for future processors. To mesh with SMT, the pre-execution static representation was converted from the difficult-to-construct static data-driven form to a more palatable, more manageable sequential form—a compressed sequence of instructions called the data-driven thread (DDT). Finally, the task of discovering and extracting pre-execution candidate computations was moved out of simulated hardware to a software form which we claimed modeled an offline, profile-driven implementation. The move from speculative dataflow to DDMT has paid dividends. The current implementation of DDMT is relatively clean and straightforward while the separation and juxtaposition of computation selection from computation pre-execution has helped provide valuable insight into both aspects.

6.2 Earlier Work

This section describes prior work that relates—either by comparison or contrast—to pre-execution's processing model or to any high-level component of the model like result reuse.

6.2.1 Dataflow Architectures

The most important aspect of pre-execution is its ability to fully decouple the processing of long latency problem computations from the rest of the program. Decoupling allows hiding (overlapping) of long latencies. It also enables an easy and straightforward implementation of "compressed" data-driven sequencing of problem computations, which allows problem operations to be reached and initiated at data-flow speeds. The importance of these two properties—latency overlapping and data-driven sequencing—to performance was recognized long ago. At that time, however, the transistor budgets or the expertise to implement the dynamic scheduling machinery and hardware multithreading required to support pre-execution—and which are commonplace today—were not available. Architects sought to imbue processors with these desirable properties at the architectural level, leaving compilers to reconcile the specification needs of the enriched interface with the sanity needs of the programmer.

Dataflow architectures [5, 24, 42, 46, 63, 74, 75, 89, 91, 98] provide both data-driven sequencing and full decoupling using a data-driven interface and a circular organization. A data-flow architecture is the exact opposite of a control-flow—sequential or von Neumann—architecture. In a control-flow architecture, parallelism is implicit as instructions implicitly name their predecessors—the instructions whose output values they consume—via the use of shared name-spaces. In a dataflow architecture, parallelism is fully exposed as every instruction explicitly names the instructions that will consume its output value. At any point in the execution of a dataflow program the identity of all ready (or partially ready) to execute instructions is known. A sequential processor is organized in a pipeline with instructions entering and exiting in sequential order. A dataflow processor is also pipelined, but the pipeline is fed by a worklist of ready to execute operations—often called the token queue—which itself is filled by the dataflow successor function implemented as the last stage of the pipe.

Dataflow architectures expose all the available parallelism in an entire program, maximizing latency tolerance and performance. Dataflow processors operate at near peak efficiency as sequencing and execution schedules are matched as close to perfectly as possible. However, despite their positive performance and efficiency characteristics and despite years of research and the construction of several fully functional prototypes, dataflow machines have failed commercially.

As it turns out, complete parallelism exposure is not such a good thing when it must be represented explicitly and is directly tied to the architectural interface, and hence correct execution. A dataflow processor contains two unique structures which allow it to implement a data-driven sequencing model. The token queue contains descriptors of all instructions that have been specified as dataflow successors by completed instructions. The rendezvous store holds partially ready two-input instructions triggered by the completion of one input operand and waiting for the second. With parallelism fully exposed, the number of instructions in-flight—i.e., in one or both of these two structures—can grow extremely large. And big problems occur when one of these structures overflows. Instructions cannot simply be discarded from either the token queue or the rendezvous station as the event which triggered and specified their entry into those structures—the completion of a predecessor instruction—has already taken place and will never take place again. In cases of overflow, the contents of these structures must be spilled to memory by software. Now, token queue overflows are costly, but may be manageable because of the simplicity of both the overflow and the overflow recovery operations. However, the rendezvous store must be associatively searched by every instruction. Rendezvous store overflows effectively turn a dataflow processor into a dataflow processor emulator. In essence, a dataflow processor can only reasonably execute programs whose maximum parallelism does not exceed the size of these structures. Ironically, while sequential processor research has focused on increasing parallelism, much dataflow processor research has been aimed at artificially *restricting* parallelism to deal with these constraints [21].

Another obstacle faced by dataflow processors is that a data-driven program representation can only be automatically generated from programs written in languages whose data dependences can be statically analyzed in full. The languages for which this is true are ones that do not permit the use of state variables through which statements may interact in untraceable ways—i.e., functional languages. Functional languages are well suited for certain tasks. However, because of their exclusion of state and their resulting awkward interface with I/O—which is inherently sequential—these languages have not been widely adopted for system use. Using a dataflow machine for system purposes is difficult regardless of the programming language, as dataflow machines—due to their inherently parallel architectural interface—do not support repeatable executions that are crucial for program development and debugging.

Pre-execution draws great inspiration from dataflow architectures. In some sense, it strives to bring some small measure of the good properties of these architectures—latency decoupling and data-driven sequencing—into the realm of sequential processing, using speculation to remove first the constraints of architectural correctness and subsequently the dependence on a data-driven architectural interface.

6.2.2 Decoupled Access/Execute Architecture

It is impossible to generate a true data-driven sequencing representation for a general program written in a sequential language. However, some degree of automatically generated decoupling may still be possible for such programs, and with decoupling the effects of data-driven sequencing can be approximated. These observations led to the proposal and development of the decoupled access/execute architecture [78].

Per its name, the decoupled access/execute architecture divides a program into two streams. The access stream contains all memory accesses and their computations. The execute stream contains everything else. The two streams communicate via a set of architected result queues. In general, dividing a sequential program into two architectural streams requires determining its data dependences precisely and in full. In other words, it is quite difficult. In fact, this is the same problem that dataflow architectures face. However, in one specific and important case, an access/execute division is trivial. Since floating-point operations cannot contribute to address calculation, putting all floating-point operations into the execute stream and all other operations into the access stream results in a correct division. For floating-point intensive scientific programs—which, not coincidentally, dominated the era in which the decoupled architecture was conceived—it also results in a highly balanced, highly efficient division as well.

A decoupled processor tolerates memory latency by providing limited slip between the access and execute stream. Although each stream executes in order, the access stream can advance ahead of its original position within the execute stream. This advance—or slip—is limited by the finite architectural buffering via which the two streams exchange results.

Access decoupling is another attractive paradigm that did not enjoy commercial success. Although it successfully deals with sequential programs, architectural access decoupling is really only viable for floating-point scientific programs where the division can be performed automatically and where the execute stream has many long-latency operations that balance the load with the access stream. It is less useful for integer programs which dominate today's workloads. Another problem with architectural decoupling is the lack of scalability of memory latency tolerance due to the limited slip.

Pre-execution has a similar relationship with the decoupled access/execute architecture as it does with dataflow architectures. Again, speculation is pre-execution's primary tool in taking decoupling from the architectural level to the microarchitectural level, freeing it both from correctness obligations and from the constraints of finite architectural name-spaces. In DDMT, communication between the pre-execution threads and the master thread takes place via the microarchitectural physical register name-space which (theoretically) can be made arbitrarily large and permit an arbitrary amount of result buffering, slip and latency tolerance.

6.2.3 Runahead Execution

Arguably, the most important performance enhancing effect of dynamic scheduling is its ability to reorder and parallelize memory operations and enable *memory-level parallelism (MLP)*—the overlapping of cache misses. The *runahead microarchitecture* [30] is a clever approach for providing MLP in an in-order processor. A runahead processor is a conventional in-order processor that contains an extra register file. When the main thread stalls on a cache miss, its register state is checkpointed to the spare register file. The processor keeps executing, using garbage values when real ones are not available and potentially initiating more cache misses. When the cache miss that initially caused the main thread to block returns, the register state is restored from the checkpoint file and execution resumes, hopefully with all near future would-be cache misses already in flight. Since an in-order register file is the only structure needed for speculation, runahead execution has theoretically unlimited ability to advance past the architectural thread.

Runahead and pre-execution are completely different, both philosophically and mechanically. Runahead *waits* for a stall, then uses control-driven sequencing to overlap it with sequentially younger instructions. Pre-execution *anticipates* a stall, and aggressively uses data-driven sequencing to overlap its execution with older instructions. However, like pre-execution, runahead execution also creates a speculation context that exploits a processor's normal execution mode but yet is separate from it and uses this context to reduce the latencies of problem instructions. Recent proposals have combined aspects of runahead execution and pre-execution in a number of ways.

6.2.4 Speculative (Control-Driven) Multithreading

Pre-execution is not the only way to speculatively multithread a single architectural thread in order to improve its performance. In fact, the older more established form is to speculatively divide a program into threads along control-flow—rather than data-flow—lines. *Speculative control-driven multithreading* is usually called just *speculative multithreading*, because initially there was no need to distinguish it from speculative data-driven multithreading—pre-execution—which only surfaced about five years later. There have been several proposals for speculative control-driven multithreaded processors, including the Multiscalar Architecture [38, 83], Single-Program Speculative Multithreading (SPSM) [29], Thread-Level Data Speculation (TLDS) [87] and Dynamic Multithreading (DMT)

[2]. There have even been two implementations—Sun's MAJC [92, 93] and NEC's Merlot [32].

Although similar in name and mechanics, the two forms of speculative multithreading are diametrically opposite in approach. Speculative (control-driven) multithreading actually exploits a different form of parallelism—speculative *thread-level parallelism (TLP)*—than speculative data-driven multithreading, which exploits an enhanced form of ILP. In speculative multithreading, a sequential program is speculatively executed as a parallel program. The program's dynamic instruction stream is divided along control-flow boundaries into contiguous chunks sometimes called tasks. The processor then executes adjacent tasks in parallel, using its knowledge of the actual task order to detect sequentiality violations. Because individual tasks are sequenced in a control-driven manner, there is no need to re-sequence work performed in speculative control-driven threads. When the oldest task in the processor is ready to retire, its final state is compared (logically) to the initial state of the immediately younger task. If the states match, then responsibility for the architectural state of the processor is simply transferred to the younger task. Of course, boundary state comparison need not take place literally. Boundary states are typically presumed to match at the outset with the older task assuming responsibility for signalling changes.

Latency tolerance in speculative multithreaded processors is achieved by overlapping the execution of problem instructions from different concurrently executing tasks. This is true even for branch misprediction latency. In contrast with a contiguous window superscalar processor, a branch mis-prediction in a task need not trigger a sequential squash of all younger tasks. As long as tasks are *control-independent*—i.e., internal task control-flow converges and does not impact inter-task sequencing—younger tasks can continue to execute, relying on the older task's signalling of data dependence violations to preserve a sequentially correct execution. The exploitation of control-independence to decouple branch mis-predictions from one another is a completely different mechanism for tolerating branch mis-prediction latency than the one used by pre-execution. Rather than tolerate and overlap branch mis-predictions, pre-execution tries to reduce the mis-prediction resolution latency or "fix" and avoid mis-predictions altogether. Pre-execution does, however, decouple problem computations from unrelated branch mis-predictions that occur within the master thread.

Like pre-execution, speculative multithreading uses a discontiguous window—allowing instructions from different places in the program to execute in parallel without having all intermediate instructions in the machine. However, in contrast with pre-execution, speculative multithreading must have buffering for the results of *all* intermediate instructions in speculative tasks. If internally tasks execute in-order, then this amounts to buffering for store values. If speculative buffering fills, speculative task execution must stall since discarding any results will violate boundary state matching assumptions. This does not imply deadlock since the lead, non-speculative task may retire store values to the memory system. DDMT has a different version of this problem. Its ability to run ahead is tied not to the number of control-driven store buffering slots, but to the number of physical registers that can hold the pre-executed results. Of course, this limit can be ignored at the cost of forfeiting pre-execution reuse via register integration.

6.2.5 Multi-Path Execution

Like speculative control-driven multithreading, speculative multipath execution [44, 50, 97, 100] is another way of using decoupling and speculative control-driven threads to overcome some limitations of the vanilla superscalar model. Multipath execution attempt to tolerate branch mis-prediction latency by hedging its bets and executing along both arms of a pathologically mis-predicted conditional branch. When the branch is ultimately resolved, execution continues along the right path, while the alternate path is discarded. Although mechanically multithreaded, multi-path execution's closest counterpart from an approach standpoint is runahead execution. While runahead execution is a reactive technique for speculating past cache misses, multi-path execution is a reactive technique for speculating past branch mis-predictions.

Multi-path executions proximity to pre-execution is in its mechanics, specifically its notion of forking a microarchitectural execution context. However, while multi-path forks a fully functional control-driven context, pre-execution forks a reduced data-driven context which does not take full responsibility for architectural state. Philosophically, the two approaches are wildly different. Like runahead execution, multi-path execution is reactive, overlapping problem instructions with downstream work after the actual problems have manifested sequentially. Pre-execution is proactive, aggressively sequencing and solving problem instructions before they are sequentially encountered.

Note, DDMT can exploit multipath execution within DDTs via its use of greedy control.

6.2.6 Pre-Execution

In addition to our own work on dependence-based prefetching and the pre-computation of virtual function calls, pre-execution—the idea of augmenting a sequential program with auxiliary threads that execute specific performance-enhancing tasks on its behalf—was initially and independently proposed by Song and Dubois in *Assisted Execution* [85], Farcy et. al. in the *Branch Flow Microarchitecture* [36] and by Chappell et. al. in *Simultaneous Subordinate Microthreading (SSMT)* [14].

Assisted Execution is an architectural software pre-execution implementation that leverages SMT processors. By virtue of its narrow architectural communication channel with the main program, an assist thread is limited to enhancing master thread performance by prefetching into the shared cache. Assist threads are control-driven threads that are not extracted from the original program but rather generated separately. This is a common combination for pre-execution systems—architecture level (software) threads must be sequenced in a control-driven manner which in turn makes extraction from the original program difficult.

SSMT threads are microcode threads or microthreads. Microthreads are composed of microinstructions that the processor can execute internally and may or may not be architecturally accessible. SSMT does not rely on an underlying multithreaded processor architecture. Similar to the microarchitectural DDMT, microinstructions are injected directly into the execution core. Microthreads must be programmed manually and must supply their own internal control flow. However, their advantage is that they can interact closely with microarchitectural components that do not have a software interface. In the original proposal, SSMT microthreads were used to implement a per-static instruction local-history branch prediction algorithm which selectively overrode the predictions made by the hardware predictor. This interface is readily implemented in microcode. Certainly microthreads can be adapted to perform any one of a variety of functions, including data and even instruction prefetching. However, they cannot offload any actual work from the master thread.

The branch flow microarchitecture targets mis-predicted branches within inner loops. The computation of the branch and the loop induction and control code are annotated statically. At runtime, annotated instructions are copied to a second pipeline where they execute the reduced loop in parallel with the complete loop executed by the original program. The reduced loop advances past the original program due to its smaller size. Branch outcomes pre-computed by the branch flow pipeline are enqueued onto the branch anticipation queue from which the main thread dequeues them. By limiting branch-flow pre-computations to unconditionally executed branches—i.e., branches which are not embedded in other single-iteration control-structures—a simple one-to-one correspondence between pre-computed branches and dynamically observed branches is maintained, making anticipation queue management a straightforward matter. Although the branch flow pipeline uses explicit control-flow, the branch flow microarchitecture performs true—albeit extremely restricted—pre-execution in the sense that it executes copies of instructions from the original program before the original program executes them.

6.2.7 Instruction Reuse and Unified Renaming

Register integration is an important component of DDMT that plays a central role in supporting DDMT's pre-execution model. In hindsight, register integration can be thought of as a combination of two separate techniques— *instruction reuse* [82] and *unified renaming* [48]. Instruction reuse supplies the empirical basis for squash reuse—generalized to same-parent-context speculative-execution reuse—and the basic structure of the reuse test. Unified renaming supplies the motivation for renamer-based physical register sharing.

Instruction reuse creates a hardware memoization scheme that indexes values computed by recent instructions using information about the creating instance's inputs. The reuse buffer (RB), in which memoization information is stored, is structurally identical to an integration table (IT), or rather an IT is structurally identical to an RB. The main difference between the two is that instruction reuse is based on architectural quantities like logical register names and values whereas register integration is based on microarchitectural quantities like physical register numbers. This difference is crucial. Instruction reuse applies more broadly to more baseline microarchitectures—even ones that don't use physical registers—and to more manifestations of reuse. However, it cannot take many microarchitectural shortcuts available to register integration—e.g., reused values must be written to the register file and re-propagated. In contrast, register integration is extremely efficient, but is restricted to certain base microarchitecture designs and to certain constrained—albeit useful—reuse scenarios.

Unified renaming is a scheme for true sharing of physical register results by different instructions in the same execution context. Unified renaming extends the semantics of physical register allocation and deallocation with reference counting. As for locating matching results during register renaming, unified renaming largely relies on the detection of identity operations—i.e., single instructions or groups of instructions whose input is equal to its output with high probability. Examples of identity operations are register move instructions and communicating store-load pairs. Unified renaming effectively collapses the data-flow graphs of identity operations to zero height.

Interestingly, it may be possible to fuse register integration and unified renaming even further, using reference counting schemes to allow register integration to exploit general reuse within the same execution context.

6.3 Contemporaneous Work

During the two years in which my work on pre-execution and DDMT was taking place, two summaries of efforts in related areas were also published. These are discussed here.

6.3.1 Data-Threaded Microarchitecture

The *Data-Threaded Microarchitecture* [101] is the most ambitious proposal to date for a system with pre-execution characteristics. The data-threaded microarchitecture is structurally reminiscent of the speculative dataflow processor. It has a superscalar pipeline with a sequential architectural interface, a dependence tracking mechanism that learns backwards consumer-to-producer data dependences and creates reverse, data-driven producer-to-consumer static mappings for them, and a dataflow-like engine. However, while the dataflow portion of the speculative dataflow processor is content to remain in a speculative supporting role, pre-executing only that small fraction of the program that is most performance critical, the dataflow portion of the data-threaded microarchitecture is much more aggressive.

The approach taken by the data-threaded microarchitecture is to slowly migrate all execution away from the superscalar side to the dataflow side. When the entire program, or a complete portion of a program has been fully mapped—its dependences have been fully analyzed—the superscalar core stops executing altogether and the program begins executing in pure, high-performance dataflow mode. An argument is made that this approach is especially effective for loops, where the dataflow side would explode a loop and ostensibly execute all of its iterations in parallel.

The data-threaded proposal did not come with an evaluation or an indication that a working version has been successfully simulated in any detail. There are many difficult problems this proposal simply ignores or claims to be able to solve. Chief among these is the ability to guarantee correct sequential semantics without re-sequencing when the superscalar side is operating, much less when the superscalar side is turned off. There may indeed be solutions to this and related problems and, if there are, then the data-threaded approach could provide extremely high performance. However, having worked in this area for the better part of four years—with potentially another twenty man years put in by others—with still no hope for a solution, I reserve the right to be skeptical.

6.3.2 Slipstream Processors

Proposed several years after the initial pre-execution work, *slipstream processors* [65, 88] present an interesting alternative approach. Slipstreaming is a microarchitectural technique. An interesting juxtaposition develops if we compare it to a microarchitectural implementation of DDMT—i.e., where DDT construction happens at runtime. DDMT starts with a single copy of the complete program and gradually augments it with small DDTs to remove stalls from repeatedly encountered problem instructions. In complete contrast, a Slipstream processor starts with two copies of the program—running one behind the other—and continuously whittles down the lead copy by removing computations of non-problem instructions. In theory, both approaches converge to the same steady state—a complete program running behind a much reduced copy of itself. The reduced copy accelerates the complete program by absorbing latency on its behalf. The complete program accelerates the reduced copy by enabling its reduction. Slipstreaming is a fitting metaphor for this synergy.

In practice, slipstreaming and DDMT do not closely converge to the same steady state. In slipstreaming, the lead copy of the program maintains largely control-driven semantics and achieves only about a 25% reduction in size over the complete program. In DDMT, the aggregate dynamic size of all DDTs rarely tops 10% of the original program. These ratios suggest differing implementations for each scheme. While DDMT can effectively siphon resources in a centralized organization, a Slipstream processor's high level of redundancy mandates that each copy of the program essentially run on its own separate processor with its own cache. In other words, DDMT is a better fit for SMT or superscalar while sliptreaming is a better fit for CMP.

6.4 Subsequent Work

The topic of pre-execution has become quite popular and several proposed implementations have been published in recent months. We separate these proposals into two camps. Proposals in the first camp use strictly control-driven sequencing and have distinct runahead mechanics. However, these systems use notions of problem instructions, backward-slicing, and register integration to enhance the runahead process. Proposals in the second camp resemble pre-execution much more closely, using decoupled data-driven sequencing to accelerate problem computations.

6.4.1 Dependence Graph Pre-Computation and ILP Balancing

Dependence-graph pre-computation (*DGP*) [4] enhances runahead by using slicing to prune the runahead execution stream so that it includes only computations of problem loads. A DGP processor is a superscalar processor enhanced with a second pipeline, an extended IFQ and some backward slicing machinery. When the main thread stalls, the slicing machinery slices backwards from problem instruction instances it finds in the IFQ. These slices are executed on the additional pipeline. Cache prefetching is achieved via cache port sharing. DGP's argument is that a large IFQ enhanced with slicing machinery plus an additional pipeline can perform limited runahead more selectively and efficiently than an extended ROB. This much is probably true as long as re-execution—which would not be required in the extended ROB scheme—is not taken into account. Whether this organization is simpler to implement than an extended ROB (note, the scheduler is not extended in any way) is debatable.

Similar to DGP is a simultaneous independent proposal for enhancing runahead execution by balancing resource allocation between nearby (superscalar) and distant (runahead) ILP [8]. We will shorten the name of the technique to BNDI. BNDI's proposed enhancement also involves reducing and hence accelerating the runahead stream. However, rather than proactively reducing the stream *a priori* via slicing, BNDI reduces the stream reactively by reclaiming storage and scheduling resources used by long-waiting runahead instructions and reallocating them to even further downstream runahead instructions that may make more immediate use of them. BNDI also supports a simple, single-level—i.e., non-recursive—form of register integration, called direct reuse which is implemented using a series of occupancy and overwrite bits.

Both DGP and BNDI enhance runahead one way, by reducing the runahead stream. However, they also handicap runahead a different way, by restricting it to the size of the extended IFQ.

6.4.2 Pre-execution of Speculative Slices

Pre-execution of speculative slices [106]—which we will shorten to PESS—is a primarily software-based pre-execution technique designed for use in SMT processors. PESS is similar to DDMT in its generality; it attacks both loads and branches. It is also similar in its SMT-based resource-siphoning model. However, the similarities end there.

Being software based, PESS is unable to take advantage of register integration. However, this limitation also relieves it from having to work within integration's constraints. Speculative slices *can* contain control-flow and PESS makes liberal use of the most powerful pre-execution idiom—unoverlapped full unrolling. Speculative slices can also be optimized and, in fact, need not look anything like dynamic slices from the actual program execution, as long as they produce the desired results—i.e., access the right memory locations and pre-compute the right branch outcomes. In this sense, PESS can be thought of as a microarchitectural version of Assisted Execution or SSMT. PESS slices are not actual slices that are extracted by slicing from the original program. The term "speculative" refers to a different level of speculation, speculation that the slices are functionally equivalent to actual dynamic program slices.

Sans register integration, PESS must use an alternative mechanism for correlating pre-computed branch outcomes with their intended dynamic instances. To perform this correlation during the fetch stage, the matching process must rely on control-flow information and its speculation vagaries rather than data-flow information and its simple correctness guarantees. PESS uses an architectural scheme in which instruction set annotations are added to the speculative slices that allow speculative slice control-regions to be unambiguously matched with master thread control-regions. Instance matching becomes trivial if at most one instance of a given static pre-executed branch exists per region. This scheme, called *valid regions* is a more robust version of the microarchitectural *epochs* scheme used in the pre-computation of virtual function call targets [69].

To this point, speculative slices must be generated by hand. It remains to be seen whether their construction can be automated. Speculative slices cannot contain stores as they are executed in architectural threads. It is possible that they could be constructed out of actual slices to which store-eliminating structured optimizations—like the ones we described for DDTs—have been applied. However, given their more traditional format and their liberal use of unoverlapped full unrolling, it is likely that they will rely on more traditional static program analysis for their construction.

6.4.3 Speculative Pre-Computation

Speculative Pre-Computation (SPC) [20] is the follow-up proposal that most resembles DDMT. SPC's runtime component is implemented on top of a multi-threaded Itanium processor, applying pre-execution where it has the potential for the greatest benefit—in a statically scheduled in-order environment.

SPC's setup component is trace-driven and offline and like DDMT finds pre-execution computations—which are called p-slices—by actually slicing backwards from problem instruction instances. The precise algorithm for extracting p-slices is left for future work. The implementation presented in the paper is not statistical, does not take overhead or even latency tolerance into account and slices within a window of 128 instructions. It is possible that these limitations have less of an impact in an in-order setting. Copies of pre-executed computations are linked into the executable binary in a control-driven representation where they traverse the instruction memory hierarchy like normal program instructions. This simplified p-slice model resembles the one used in slice processors [58] and could ultimately lend itself to a similar hardware implementation. In a recent conference submission by the same authors, the details of such an implementation are given.

One feature incorporated by SPC that DDMT explicitly excludes is chained triggering—allowing instructions in pre-executing computations to trigger the pre-execution of other computations. Chained p-slice execution is controlled via a combination of mechanisms. The pending slice queue (PSQ) buffers triggered p-slices until hardware contexts are ready to service them. The outstanding slice counter table (OSC) tracks the number of outstanding p-slices and does not execute p-slices if that number exceeds some threshold. Finally, the main thread can abort all child p-slices via a special instruction. In certain benchmarks, chaining combined with these control mechanisms can improve performance over exclusive main thread triggering .

6.4.4 Software-Controlled Pre-Execution

Software-controlled pre-execution (SCPE) [54] is—according to our definitions—only partially true to its name. It is software controlled and it is pre-execution in the sense that it speculatively executes sections of the original program. However, the speculative computations executed by SCPE are not data-driven but rather full control-driven regions from the original program. In a way, SCPE uses speculative control-driven threads in a pre-execution context, employing them redundantly to enhance performance rather than forming a full, non-redundant sequential division of the program.

In SCPE, the compiler annotates a sequential program with *PreExecute_Start*, *PreExecute_Stop* and *PreExecute_Cancel* instructions which have been added to the ISA as extensions. *PreExecute_Start* takes a constant address argument and forks a thread that begins executing at that address with a copy of the forking thread's register context. *PreExecute_Start* returns a thread identifier to the parent thread, much like a *fork* system call. *PreExecute_Stop* ends the execution of the current pre-executing thread. *PreExecute_Cancel* takes a thread identifier argument and ends the execution of that thread. SCPE does not implement chained triggering—*PreExecute_Start* and *PreExecute_Cancel* instructions are ignored from within pre-executing threads. *PreExecute_Stop* is ignored when seen by the main thread.

SCPE's only aim is to prefetch, which somewhat simplifies its implementation and allows it to be implemented on only a slightly modified SMT processor. Pre-executing threads are ordinary SMT threads which are allowed to allocate and free registers. No register integration is implemented so freeing registers in pre-executing threads—as long as care is taken not to free registers belonging to the parent thread—is acceptable. However, either an expensive software register copying mechanism or hardware physical register reference counting must be implemented to ensure this. The chief SMT modification required by SCPE is a scratchpad to which pre-executing store values must be written as these cannot be written to the cache. This scratchpad is the analog our DDSQ. Pre-executing loads access the cache and scratchpad in parallel, allowing them to pick up values from older stores within the same thread.

The compiler algorithm responsible for annotating the program is based on miss and locality analysis and in general tries to maximize pre-execution—i.e., prefetch—distance. The compiler also performs transformations, like loop unrolling, that explicitly facilitate SCPE look-ahead.

6.4.5 Slice Processors

Slice processors [58] support a restricted form of pre-execution. A slice processor is a superscalar processor augmented with a slicer, a slice-cache, and several streamlined scout pipelines each containing an instruction buffer and a simple integer ALU. The slicer is a simple instruction stream post-processor that can sequentially backpedal along register dependences within a fixed window. The slicer is activated on the retirement of a cache miss and the resulting slice—

extracted from the most recent 32 or 64 non-store, non-control, non-FP instructions—is stored in the slice-cache. The oldest instruction in the slice is identified as the lead, the equivalent of our trigger instruction. The slice-cache is organized precisely like a DDTC, as there are only a finite number of ways such things may be organized.

The mechanics of slice execution are as follows. When the master thread renames a lead, it allocates a free scout pipeline to the corresponding slice. The scout pipeline is initialized with a copy of the master thread's map table at the time of the fork. The scout pipeline is a simple, in-order scalar pipeline—there is no need for more, as all it does is execute dependence chains—which nevertheless contains register renaming logic. This logic is required to allow the scout thread to wait for external master-thread values that have not yet computed. Once external values are available, the scout thread processes the slice, using the instruction buffer as temporary register storage. Executing the slice on the scout pipeline implements cache prefetching as the scout pipeline shares the data cache ports with the main superscalar pipeline. Branch pre-execution is not implemented.

From a sequencing standpoint, slice processors do perform actual pre-execution. However, the slices are constrained by the sequential limitations of the slicer itself. The end result is that the system as a whole behaves more like a runahead machine.

6.5 Chapter Summary

This chapter presents some of the research that has supported and motivated my work, as well as concurrent and subsequent efforts in related areas. The work discussed here all relates to pre-execution as a processing model. I have chosen not to discuss work that relates to individual, concrete aspects of pre-execution, the DDMT microarchitecture, or DDT selection. Certainly many of these exist. In the pre-execution domain, much work on alternative approaches for enhancing ILP and other forms parallelism was not discussed. In the DDMT microarchitecture realm, we ignored the wide and varied contributions—in the form of components—made by others to both the base microarchitecture and DDT specific enhancements, including (but not limited to) dynamic scheduling, register renaming, simultaneous multithreading, speculative store-to-load value forwarding, DIVA, and thread scheduling. Finally, in the DDT selection area, we have not acknowledged works on dynamic backward slicing, identification of stable memory dependences, characterization of backward slices, statistical analysis of program graphs, loop unrolling, and algorithms for graph comparison and merging. I apologize for these omissions.

Chapter 7

Conclusion

As we survey the computing needs of the coming decade, sequential program performance—the performance of a single program executing on a single logical processor—will continue to be important. As new constraints on power, area, ease of design and verification are activated, the emphasis shifts from increasing performance by any means and at any cost to increasing performance in new and innovative ways that leverage the existing processor infrastructure. This dissertation introduces pre-execution, a new paradigm for extracting additional instruction level parallelism and performance from sequential programs using a mechanism every processor naturally supports—the execution of instructions.

This dissertation makes four major contributions. First, it introduces and defines *pre-execution*, a new processing model for extracting additional ILP from sequential programs. Second, it describes *speculative data-driven multithreading* (DDMT), a proposed implementation of pre-execution that extends a common dynamically scheduled superscalar design in a straightforward way with additional components that also have uses in simultaneous multithreading and computation reuse. Third, it introduces a quantitative framework for the *automatic extraction of pre-execution computation descriptions* from program traces. Finally, it presents a *simulation driven performance evaluation* of DDMT and automatic computation selection.

This final chapter summarizes these four contributions. Section 7.1 recaps the novel concepts that were introduced. Section 7.2 summarizes the important results of the performance evaluation. Finally, in Section 7.3, we use the insight gained into pre-execution and our understanding of the limitations of the current model to point to some possible future work.

7.1 Summary of Concepts, Mechanisms and Frameworks

This dissertation introduces a novel concept, a novel mechanism, and a novel framework. The concept is *pre-execution*, a technique for stepping outside the superscalar paradigm in a directed way in order to increase the exploitable instruction level parallelism of sequential programs. The mechanism is *data-driven multithreading (DDMT)*, an implementation of pre-execution as a set of extensions to a superscalar processor. The framework is a formalization of the notions of latency tolerance and overhead in an algorithm that automatically selects what to pre-execute by processing program traces.

7.1.1 Pre-Execution

Sequential program performance degrades when the processor needs a certain value to proceed, but execution has yet to provide that value. Late values that result in effective stalls are the outcomes of mis-predicted branches and the addresses of data blocks that are not in the cache but which will soon be accessed. In practice, 95% of dynamic branch outcomes and load addresses can be obtained quickly and accurately via prediction. We call the remaining 5% of dynamic loads and branches, whose outcomes and addresses are unpredictable, *performance degrading instances (PDIs)*. PDI values can be supplied via actual execution, but results from execution are too late, as the stall has already been incurred. Pre-execution is a way of using execution to obtain PDI values in a timely manner.

Empirically, most PDIs are caused by a few static *problem instructions (PIs)*. In pre-execution, copies of PI computations—or simply *problem computations* —are executed in parallel with the complete program. To obtain a speed advantage over the main program, a pre-execution context sequences problem computations in a *proactive, out-of-order (i.e., compressed) manner,* skipping over unrelated instructions from the complete program. Proactive, out-of-order sequencing allows computations to be hoisted and long latencies to be initiated early. *Decoupling the pre-executing copy from the master thread* allows these latencies to be hidden. Since a stall in one execution context does not impact another context, "moving" an operation to a pre-execution context allows its latency to be overlapped with useful instructions from the master thread. While the pre-execution thread stalls, the master thread can keep pushing useful instructions through the processor for many cycles. To reconcile out-of-order sequencing and decoupling with sequential execution, we make pre-execution *redundant* with respect to the complete program. Redundancy relieves pre-execution of architectural correctness obligations, expanding its scope of applicability. However, redundancy also makes pre-execution an expensive technique that should be used judiciously.

Pre-execution is a promising technology. It is a reliable mechanism for the timely supply of critical values that are difficult to obtain in other ways. Its decoupled nature enables a high degree of latency overlapping that is not bounded by physical resource limitations. Its redundant formulation allows it to be used for performance troubleshooting without concern for master thread correcness. Finally, success rates of traditional predictors and the concentration of dynamic problems to a few static PIs allow pre-execution's relatively high cost to be controlled by focusing its use.

7.1.2 Speculative Data Driven Multithreading

An implementation of pre-execution has two components. The *runtime* component pre-executes specified problem computations and communicates pre-executed results to the master thread. The *setup* component identifies candidate pre-execution computations and communicates its selections to the runtime component. The runtime component is obviously dynamic; the setup component may be implemented statically or dynamically.

Speculative data-driven multithreading (DDMT), our proposed implementation of the runtime component, is a set of small, straightforward extensions to a dynamically-scheduled superscalar processor. In this dissertation, we leave implementation of DDMT's setup component open, but we discuss the possibilities of software, hardware, and hybrid implementations.

DDMT's unit of pre-execution is the *data-driven thread (DDT)*—a static sequence of instructions that encodes one or more PDI computations. Each static DDT is associated with a static *trigger instruction*. When the master thread encounters an instance of a trigger instruction, the processor forks a copy of the corresponding DDT. The processor siphons some register renaming bandwidth away from the master thread to inject the DDT into the execution core. The execution core naturally interleaves instructions from multiple threads.

In addition to supporting fine-grain resource allocation, DDMT's centralized organization facilitates communication between the DDT and master thread. The shared first-level cache allows a DDT to prefetch for a master thread by simply executing the appropriate loads. Beyond prefetching, the shared physical register file allows a DDT to contribute actual register values to the master thread, via a modification to register renaming called *register integration*. In register integration, the master thread can recognize and claim DDT results—physical registers allocated by DDT instructions—as its own. In addition to sparing the master thread from having to repeat pre-executed work, register integration also enforces a one-to-one correspondence between DDT and master thread instructions. We use this property to match pre-executed branch outcomes with their intended dynamic branch instances and to implement *instantaneous branch resolution*—the fixup of a mispredicted branch during the register renaming stage.

DDMT is a promising implementation of pre-execution. It requires only localized changes to a conventional superscalar processor. Its parasitic resource consumption model—i.e., there is no dedicated pre-execution bandwidth—supports high processor utilization even when pre-execution is inactive. Finally, its inclusion of register integration allows the master thread to directly leverage pre-executed work.

7.1.3 Automated DDT Selection

One important element demonstrated in this dissertation is that, regardless of its implementation, DDMT's setup component—the discovery or selection of DDTs—is automatable. Register integration restricts the structure of DDTs to dynamic backward slices of PDIs. This restriction turns the problem of constructing DDTs from scratch to the problem of *automatically* enumerating the backwards slices of PDIs observed in a program execution and *selecting* DDTs from that bounded space. We develop a framework that formalizes the notions of latency tolerance and overhead and allows the selection task to be performed rationally.

We split the selection task into three phases. First, we identify PIs whose PDIs will be the targets of DDTs. We do so by estimating the number of PDIs covered as a function of the amount of work done dynamically by DDTs. Second, we examine program traces to collect a database of the backwards slices of PDIs. We analyze the database using our formal framework to select a set of DDTs each of which covers a set of PDIs via a single (but hopefully recurring) computation. The framework optimizes a metric called *aggregate advantage*, which attempts to model the aggregate latency tolerance of all PDIs covered by the DDT minus the aggregate overhead of all DDT executions. The final phase merges partially overlapping DDTs to reduce overhead.

7.2 Summary of Results

In this section, we summarize the results of our simulation-driven performance evaluation. Our overall conclusions are that pre-execution in general, and DDMT in particular, show promise as a high performance technique that is amenable to several different setup component implementations. However, DDMT as currently formulated has several limitations that restrict its applicability and performance robustness.

7.2.1 Performance Potential of DDMT

The results of our performance evaluation, which may be described as preliminary, are encouraging. When added to an aggressive, wide, superscalar processor with large caches and a large branch predictor, DDMT produces large speedups on challenging micro-benchmarks and performance improvements in the 10% to 15% range on integer benchmarks. Correcting for the limited number of instructions targeted, these performance improvements are 20% to 35% of the "ideal" performance improvements achievable only in simulation.

DDMT's performance diagnostics are positive. It achieves its stated goals of reducing load execution latency and branch misprediction resolution latency. Also, its direct approach attacks the instructions that need attacking (PDIs) a large fraction of the time.

Register integration improves DDMT performance by saving the master thread from re-executing pre-executed instructions. Register integration's implementation of instantaneous branch resolution improves the performance of programs which exploit branch pre-execution. Finally, register integration also implements squash reuse, which achieves modest speedups in its own right and enhances the benefits of pre-execution reuse.

DDMT's execution model limits pre-execution overhead in a natural way. Small, fixed size DDTs and a triggering model that ties DDT forking to progress made by the master thread are effective ways of limiting resource consumption that can otherwise turn destructive. The arbitrarily long, but fixed, slip between a DDT and the master thread also appears to be the right model for avoiding early prefetching effects that may afflict other prefetching systems where slip between the two is unconstrained.

7.2.2 Sensitivity of DDMT

In addition to a basic performance evaluation, we performed several studies that measured DDMT's sensitivity to its own microarchitectural configuration, the parameters of the DDT selection algorithm, and the DDMT setup/runtime relationship.

Out of all of its microarchitectural components, DDMT is most sensitive to the associativity of the integration table (IT) and to the implementation of the load integration suppression predictor (LISP). IT associativity is important because, in our current formulation, its interaction with physical register tracking limits the degree of unrolling DDTs can practically exploit. The LISP is important because it can significantly reduce the number of performance degrading load mis-integrations. We have found DDMT to be largely insensitive to other pre-execution specific microarchitectural parameters, including IT and physical register size, the number of reservation station entries, the number of hardware thread contexts, and the DDT injection policy.

The DDT selection process also has an effect on DDMT performance. In our central baseline configuration, DDMT is largely insensitive to the PI definition (the thresholds used to separate PIs from non-PIs), but it is sensitive to the size, scope, and unrolling degree constraints placed on the algorithm. However, the character of the chosen DDTs does not change—and hence neither does DDMT performance—when these limits are raised above a certain level. DDT selection is particularly sensitive to the estimate of the sequencing rate of the master thread. A low estimate leads the DDT selection algorithm to believe that the master thread is highly constrained by sequencing and that short DDTs suffice to overcome this constraint. A high estimate results in long DDTs, as the selection algorithm believes that these are needed to achieve a sequencing advantage over the master thread.

The abilities to predict the right PDIs to attack and to select the right DDTs with which to attack them are largely independent of the input data set with which this knowledge was acquired. This implies that the DDT setup phase is amenable to any number of implementations, even static implementations using completely different data sets.

Our central configuration is a wide superscalar processor with large caches and a moderate memory latency. Believing that, due to increasing clock frequency, future processors will be narrower, use smaller caches, and observe relatively higher memory latencies, we investigated DDMT's sensitivity to these parameters.

DDMT's performance impact decreases slightly as we narrow processor width from 8 to 6 to 4. Although a narrower processor has a larger sequencing disadvantage vis-a-vis a DDT, it is also more highly utilized making DDMT more expensive. On the other hand, DDMT's performance impact increases relative to that of an ideal solution.

Shrinking the sizes of the caches and branch predictor tables results in more cache misses and more branch mis-predictions—i.e., more PDIs for DDMT to cover. DDMT's performance impact increases under these conditions as baseline performance drops. It is possible that, with an increase in the PDI population, sensitivity to the PI definition—which controls how many PDIs DDMT attempts to cover—will increase. If this is the case, better performance may be achieved by adjusting the thresholds of this definition.

A faster clock makes memory latency, and potentially L2 cache latency, appear relatively longer. It also potentially increases the need for pipelining, exacerbating the branch mis-prediction resolution latency. Increased PDI latencies necessitate longer, more highly unrolled DDTs and a willingness on the part of the DDT selection algorithm to accept DDTs that only partially cover latencies. Our results show that DDMT performance impact remains at the levels associated with the small cache configuration, even when the DDT selection algorithm is not tuned to these new conditions.

Proper evaluation of DDMT under different baseline microarchitectures potentially requires the PI definition, the DDT selection algorithm parameters and the DDMT-specific microarchitectural parameters to be tuned simultaneously. We did not perform such an exhaustive investigation in this dissertation.

7.2.3 Limitations of DDMT

While DDMT's micro-benchmark performance is excellent, its performance on more realistic benchmarks—which nevertheless are single-threaded and contain no operating system component—is less consistently good. We believe that two basic limitations are responsible for this: (1) the lack of an interface that sends pre-executed branch results to the master thread fetch unit, (2) the inability to exploit unoverlapped full unrolling in more scenarios. The two micro-benchmarks show DDMT in a good light because they do not exercise these two weaknesses. Both use DDMT mainly for prefetching, and their dominant computations have a structure that is one of the few for which DDMT can exploit unoverlapped full unrolling.

The lack of an interface for sending pre-executed branch outcomes to the master thread's fetch unit prevents DDMT from completely eliminating the penalty of a branch mis-prediction. Even if DDMT can fully hide the latency of a mis-predicted branch from an execution standpoint—i.e., it can compute the branch before the master thread has fetched it—the fact that rendezvous actually takes place at the register renaming stage means that a portion of the latency (equal to the number of stages between fetch and rename) remains exposed. For programs with many problem branches, this cumulative uncovered penalty is substantial. The somewhat disappointing results of our limit study with perfect branch resolution corroborate this observation.

The inability to capture and exploit cases of unoverlapped full unrolling also limits the performance of DDMT as described in this dissertation. Unoverlapped full unrolling is a powerful pre-execution idiom and the only one capable of hiding serial latencies in tight loops. DDMT's exploitation of unoverlapped full unrolling is restricted to those cases in which the program's natural statistical degree of unrolling—i.e., the average number of dynamic iterations executed—is low. Only in those cases can the automated DDT selection algorithm recognize the idiom. In addition, the current implementation of DDT sequencing does not support arbitrarily long DDTs, looping, trigger chaining, or any other mechanism that can dynamically achieve high unrolling degrees.

There are other potential limitations of DDMT as is formulated in this dissertation, including an inability to optimize DDTs. However, we have little means of quantifying—or even qualifying—their effects at this time.

7.3 Future Work

This dissertation describes one potential implementation of pre-execution. Work remains to be done before a fully functional, commercially viable system that includes pre-execution becomes a reality. The good news is that the promise of pre-execution has excited other researchers in both academia and industry and that the necessary work has been undertaken with enthusiasm.

This section summarizes my view of the future directions of research in pre-execution. Admittedly, this view is colored by my own experience and my preference for speculative data-driven multithreading as the implementation of choice. It is my belief that some work is needed in both microarchitecture and computation selection areas. Most of this work deals with the shortcomings of DDMT as described above. However, the most pressing challenge is in the architecture and system area. We must find an interface for pre-execution that is acceptable to system and application implementors and that leverages the capabilities of software while isolating the casual user and even the non-casual user from the mechanical details.

7.3.1 System-Friendly Implementation and Interface for DDMT

The runtime component of DDMT is almost entirely microarchitectural. For DDMT to succeed, a software-friendly formulation of its setup component is necessary.

As described in Chapter 3, there are three possibilities. The first is to create a purely microarchitectural implementation of the setup component, making DDMT entirely transparent to software. This is the approach taken by two of DDMT's successors, speculative pre-computation [20] and slice processors [58]. Work along these lines has already begun, although a hardware implementation of the algorithm we described will be daunting. The challenges will be in creating a formulation of the algorithm that can select DDTs with as much accuracy, while using approximations rather than full statistical analysis.

A second option is to resort to implementing DDT selection in software and to create an expanded instruction set and executable interface that can convey this information to the processor. This is the approach taken by software controlled pre-execution [54] and speculative slices [106]. The challenge here is to create an interface that is expressive enough and compatible both forward and backward. A static framework for selecting DDTs may also be necessary, as trace collection and trace driven program optimization are not common practices.

A final promising approach—and one that has not been explored yet—is to insert the DDT selection process into a virtual machine (VM), thereby executing the selection algorithms in VM software while remaining transparent to systems software above. Being dynamic, a VM implementation would have all the raw information—traces with full address annotations, knowledge of the identities of PDIs, etc.—from which to select good DDTs. The challenge here is to create fast selection algorithms whose online cost would be low.

7.3.2 Practical Register Integration

The microarchitecture required to support speculative data-driven multithreading is relatively simple. Most DDMT specific components are small, most policy changes are localized, and most internal semantic changes are both evolutionary and self-contained. However, several aspects of the microarchitecture do deserve some attention, both to improve the performance of DDMT and to simplify—and increase the probability of—an eventual implementation.

Undoubtedly, the microarchitectural aspect most in need of attention is register integration. Although it has undergone several significant simplifications since its initial conception, register integration is still admittedly complex to implement. An implementation of its current formulation may not fit within the timing constraints of current register renaming implementations and may not justify the addition of a pipeline stage. Even if increased pipelining is warranted, the critical timing path of register renaming—the passing of integration decisions from one group of instructions to a subsequent group—must still be implemented in less than a cycle. As we have shown, the complexity of this operation—even stripped to its barest form with most of the information pre-computed by previous pipeline stages—is $O(NM^2)$ where N is the superscalar width of the machine and M is the associativity of the IT. It is quite possible that a strong timing restriction will restrict M to a small number like 1 or 2. A low integration associativity will restrict performance impact especially in cases where unrolling is used. If low integration associativity is required, an organization that decouples the internal associativity of the IT from the integration associativity is required. Of course, all of these concerns are pure speculation because a circuit simulation of even the simplest register integration circuit has not yet been undertaken.

7.3.3 Branch Predictor Interface for Pre-Executed Branches

Although powerful, the instant branch resolution enabled by register integration is not as powerful as the improved branch prediction that may be possible with a mechanism that can reliably match pre-computed branch outcomes with their intended dynamic branches at the fetch stage of the processor.

The topic of such mechanisms has been the subject of much recent research [14, 36, 69, 106]. However, since these mechanisms are all based on notions of control-flow, their implementations invariably have to deal with control-flow speculation and conditional execution—i.e., missing or extraneous pre-computed outcomes and recovery from the incorrect consumption of a pre-computed outcome. These aspects are transparent to a data-flow based technique like register integration. Of course, the price for this simplicity and accuracy is the delay associated with getting the correct branch outcome. Proposed implementations of control-driven branch correlation mechanisms are either less than reliable [14], limited in their applicability [36, 69], or require some help from software [106], which necessitates an expansion of the architectural interface. Regardless, incorporating one of these mechanisms into DDMT is probably not a bad idea. If a suitable high accuracy implementation can be found, then the need for register integration itself—and subsequently the centralized organization of DDMT—may be revisited. Alternatively, a simple, lower accuracy implementation can be used in conjunction with register integration, with the latter serving as a higher-latency but perfect-accuracy backup. A satisfactory solution to this question will likely improve the performance of DDMT significantly and make the case for its adoption more compelling.

7.3.4 Unoverlapped Full Unrolling for DDTs

One of the great limitations of our current formulation of DDMT is the restricted applicability of unoverlapped full unrolling within the framework. The constraints come from two directions. First, the use of implicit data-driven sequencing mandates a small fixed DDT size, limiting the number of loop iterations that can be unrolled within a DDT. Second, the dynamic nature of the DDT selection algorithm makes recognizing instances of unoverlapped full unrolling difficult in situations in which the common dynamic iteration count of a loop is more than a few iterations. Exploiting unoverlapped full unrolling requires solving both of these issues.

We have already discussed one option for simulating loop control for DDTs in Chapter 2. Trigger chaining enables unrolling of arbitrary degrees by moving the loop control from the DDT itself to the fork mechanism. The drawbacks of trigger chaining are redundant forking and the distinct possibility of a DDT explosion. Explicit throttling mechanisms are needed to avoid these problems. A similar effect can be achieved by moving the task of unrolling the loop to the CMIS, the component responsible for injecting DDT instructions into the processor. The CMIS can simulate unrolling by injecting the same piece of DDT code into the processor repeatedly. Injector based loop control does not suffer from the redundant forking problem, since fork control is centralized.

Building an automatic DDT selection algorithm that can recognize instances of unoverlapped full unrolling is more of a challenge. Recognizing instances of unoverlapped full unrolling requires recognizing loop structures and, more importantly, finding triggers sufficiently in advance of the entire loop. Doing so dynamically is extremely difficult as long loop iteration counts mean that the likelihood of analyzing late iterations and pre-loop code within the same slicing window are slim. Statistical analysis will then dissociate these late iterations from the pre-loop code and either attack them using induction unrolling or else give up. Recognizing unoverlapped full unrolling in high dynamic iteration count cases likely requires analyzing the program *statically*.

The problem with a static analysis framework is that DDTs must be dynamic dataflow graphs, and it is more difficult to generate these by explicitly laying out static code than by mining traces. It is also unclear how our cost metrics translate to a static framework, although admittedly that is not a fundamental problem, simply one to which we have not given much thought. In any event, the subject of static analysis frameworks has already been broached, albeit in a context which does not exactly fall within our definition of pre-execution [54]. It is also possible that a hybrid static/dynamic framework could be used. Here, the dynamic portion would be responsible for generating the DDTs and computing latency tolerances and costs. However, the static framework could be consulted in special cases to advise of the presence of higher level control structures that may not be visible from within the slicing window.

7.3.5 Optimizations for DDTs

DDT optimization is potentially important. A DDT's prime objective is latency tolerance, and optimization may help in this regard. On the surface, it may appear that the opportunity for optimization within DDTs, especially those extracted from highly optimized programs, is low. However, this is not the case. In fact, there is opportunity for optimization within DDTs that does *not* exist for the corresponding computation within the context of the complete program [106]. First, DDTs are speculative, so optimizations within them are not bound by the same correctness guarantees and resource constraints as compiler optimizations on the complete program. Second, DDTs implicitly encode paths and can be optimized in ways that are only valid on those paths. Finally, DDTs transparently cross procedure boundaries, thus enabling a high degree of inter-procedural optimization not possible in the complete program context. The "register allocation" of save/restore pairs within DDTs is an example of all of these opportunities. Some save/restore pairs within the program are a result of a local shortage of logical register names. A DDT potentially encodes a much smaller computation than the local program region, and thus its context may have more free register names. Some save/restore pairs are needed to allow the load to access different values along different paths—i.e., a single restore will access saves along different paths. Since a DDT encodes a single path with respect to the load, register allocation is possible. Finally, some save/restore pairs are used to pass values across procedure calls, an artifact of separate compilation. These can be register allocated in DDTs which are monolithically "compiled."

It is impossible to marry DDT optimization with register integration in its current form. However, it may be possible to optimize *and* integrate DDTs if register integration can be modified so that it can recognize these optimizations and do the right thing. More accurately, to be able to incorporate a certain kind of optimization within DDTs, we need to find some corresponding dynamic optimization that can take place at register rename time and cast the DDT optimization as an instance of it, or alternatively allow the optimization to act upon the DDT naturally.

There are already microarchitectural analogs for two of the structured optimizations we have described. Register moves can be eliminated from the dynamic dataflow graph by a technique called *unified renaming* [48]. Memory communication can be dynamically eliminated at register renaming time using either *memory renaming* [96], *speculative memory bypassing* [59], or *dynamic load elimination* [35]. Marrying these techniques with register integration—and DDT optimization—is interesting future work.

References

[1] S.G. Abraham, R.A. Sugumar, D. Windheiser, B.R. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. In *Proc. 26th International Symposium on Microarchitecture*, pages 139–152, Dec. 1993.

[2] H. Akkary and M.A. Driscoll. A Dynamic Multithreading Processor. In *Proc. 31st International Symposium on Microarchitecture*, pages 226–236, Nov. 1998.

[3] G. Ammons and J.R. Larus. Exploiting Hardware Performance Conters with Flow and Context Sensitive Profiling. In *Proc. 1997 Conference on Programming Language Design and Implementation*, pages 85–96, Jun. 1997.

[4] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Prc. 28th International Symposium on Computer Architecture*, pages 52–61, Jul. 2001.

[5] Arvind and R.S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.

[6] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. 32nd International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[7] J.L. Baer and T.F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. 1991 Conference on Supercomputing*, pages 176–186, 1991.

[8] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Allocating Processor Resources Between Nearby and Distant ILP. In *Prc. 28th International Symposium on Computer Architecture*, pages 26–37, Jul. 2001.

[9] T. Ball and J.R. Larus. Optimally Profiling and Tracing Programs. In *Proc. 19th Anuual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, 1992.

[10] T. Ball and J.R. Larus. Efficient Path Profiling. In *Proc. 29th International Symposium on Microarchitecture*, pages 46–57, Dec. 1996.

[11] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappaport, A. Yoaz, and U. Weiser. Correlated Load Address Predictors. In *Proc. 26th International Symposium on Computer Architecture*, pages 54–63, May 1999.

[12] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[13] P-Y. Chang, E. Hao, and Y.N. Patt. Target Prediction for Indirect Jumps. In *Proc. 24th International Symposium on Computer Architecture*, pages 274–283, Jun 1997.

[14] R.S. Chappell, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. 26th International Symposium on Computer Architecture*, May 1999.

[15] S. Chatterjee, C. Weaver, and T. Austin. Efficient Checker Processor Design. In *Proc. 33rd International Symposium on Microarchitecture*, pages 87–97, Dec. 2000.

[16] T-F. Chen and J-L. Baer. A performance study of Software and Hardware Prefetching Techniques. In *Proc. 21st International Symposium on Computer Architecture*, pages 223–232, Apr. 1994.

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

[18] Y. Chou, P. Pillai, H. Schmit, and J.P. Shen. PipeRench Implementation of the Instruction Path Coprocessor. In *Proc. 33rd International Symposium on Microarchitecture*, pages 147–158, Dec. 2000.

[19] Y. Chou and J.P. Shen. Instruction Path Coprocessors. In *Proc. 27th International Symposium on Computer Architecture*, pages 270–281, May 2000.

[20] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J.P. Shen. Speculative Pre-Computation: Long Range Prefetching of Delinquent Loads. In *Prc. 28th International Symposium on Computer Architecture*, pages 14–25, Jul. 2001.

[21] D.E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proc. 15th International Symposium on Computer Architecture*, pages 141–150, May 1988.

[22] G.E. Daddis and H.C. Torng. The concurrent execution of multiple instruction streams on superscalar processors. In *Proc. International Conference on Parallel Processing*, pages 76–83, May 1991.

[23] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proc. 30th International Symposium on Microarchitecture*, pages 292–302, Dec. 1997.

[24] J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic dataflow processor. In *Proc. 2nd International Symposium on Computer Architecture*, pages 126–132, Jan. 1975.

[25] K. Diefendorf. K7 Challenges Intel. *Microprocessor Report*, 12(14), Nov. 1998.

[26] K. Diefendorf. Compaq Chooses SMT for Alpha. *Microprocessor Report*, 13(16), Dec. 1999.

[27] K. Driesen and U. Hoelzle. Accurate Indirect Branch Prediction. In *Proc. 25th International Symposium on Computer Architecture*, pages 167–178, Jun. 1998.

[28] K. Driesen and U. Hoelzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Proc. 31st International Symposium on Microarchitecture*, pages 249–258, Dec. 1998.

[29] P.K. Dubey, K. O'Brien, K.A. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading. In *Proc. 1995 Conference on Parallel Architectures and Compilation Techniques*, pages 109–121, Jun. 1995.

[30] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proc. 1997 International Conference on Supercomputing*, pages 68–75, Jun. 1997.

[31] H. Dwyer and H.C. Torng. An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts. In *Proc. 25th International Symposium on Microarchitecture*, pages 272–281, Dec. 1992.

[32] M. Edahiro, S. Matsushita, M. Yamashima, and N. Nishi. A single chip multiprocessor for smart terminals. *IEEE Micro*, pages 12–20, Jul./Aug. 2000.

[33] A.N. Eden and T. Mudge. The YAGS Branch Prediction Scheme. In *Proc. 31st International Symposium on Microarchitecture*, pages 69–77, Dec. 1998.

[34] A.N. Eden, J. Ringenberg, S. Sparrow, and T. Mudge. Hybrid Myths in Branch Prediction. In *Proc. 7th International Conference on Information Systems Analysis and Synthesis*, Jul. 2001.

[35] R. Espasa, M. Valero, and J.E. Smith. Out-of-Order Vector Architectures. In *Proc. 30th International Symposium on Microarchitecture*, pages 160–170, Dec. 1997.

[36] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. 31st International Symposium on Microarchitecture*, pages 59–68, Dec. 1998.

[37] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical Path Prediction. In *Proc. 27th Annual International Symposium on Computer Architecture*, pages 74–85, Jul. 2001.

[38] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, Madison, WI 53706, Nov. 1993.

[39] F. Gabbay and A. Mendelson. Can Program Profiling Support Value Prediction? In *Proc. 30th International Symposium on Microarchitecture*, pages 270–280, Dec. 1997.

[40] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. of the International Conference on Parallel Processing*, pages 355–364, Aug. 1991.

[41] P. Glaskowsky. Pentium 4 (Partially) Previewed. *Microprocessor Report*, 14(8), Aug. 2000.

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

[43] L. Gwenapp. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2), Feb. 1995.

[44] T.H. Heil and J.E. Smith. Selective Dual Path Execution. Technical report, University of Wisconsin-Madison, Department of Electrical and Computer Engineering, Nov. 1996.

[45] M. Horowitz, M. Martonosi, T.C. Mowry, and M.D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proc. 23rd Annual International Symposium on Computer Architecture*, pages 244–255, May 1996.

[46] R.A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15 International Symposium on Computer Architecture*, pages 131–140, May 1988.

[47] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Proc. 24th International Symposium on Computer Architecture*, pages 252–263, Jun. 1997.

[48] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification. In *Proc. 31st International Symposium on Microarchitecture*, pages 216–225, Dec. 1998.

[49] R.E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), Mar./Apr. 1999.

[50] A. Klauser, A. Paithankar, and D. Grunwald. Selective Eager Execution on the PolyPath Architecture. In *Proc. 25th International Symposium on Computer Architecture*, pages 250–259, Jun. 1998.

[51] J.K.F. Lee and A.J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 17(1):6–22, Jan. 1984.

[52] M. Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, May 1997.

[53] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value Locality and Load Value Prediction. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.

[54] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Prc. 28th International Symposium on Computer Architecture*, pages 40–51, Jul. 2001.

[55] C-K. Luk and T.C. Mowry. Compiler Based Prefetching for Recursive Data Structures. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[56] M.M.K. Martin, D.J. Sorin, H.W. Cain, M.D. Hill, and M.K. Lipasti. Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Mulitprocessing. In *Proc. 34th International Symposium on Microarchitecture (to appear)*, Dec. 2001.

[57] S. Mehrotra and W.L. Harrison. Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Program. In *Proc. 10th International Conference on Supercomputing*, pages 133–139, May 1996.

[58] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice Processors: An Implementation of Operation-Prediction. In *Proc. 2001 International Conference on Supercomputing*, Jun. 2001.

[59] A. Moshovos and G.S. Sohi. Streamlining Inter-Operation Communication via Data Dependence Prediction. In *Proc. 30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[60] A. Moshovos and G.S. Sohi. Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors. In *Proc. 6th Annual International Symposium on High-Performance Computer Architecture*, pages 301–312, Feb. 2000.

[61] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache Miss Heuristics and Preloading Techniques for General-Purpose Programs. In *Proc. 28th International Symposium on Microarchitecture*, pages 243–248, Nov. 1995.

[62] S.T. Pan, K. So, and J.T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction using Branch Correlation. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Oct. 1992.

[63] G. Papadopoulos and D. Culler. Monsoon: An Explict Token-Store Architecture. In *Proc. 17th International Symposium on Computer Architecture*, pages 82–91, Jul. 1990.

[64] S.S. Pinter and A. Yoaz. Tango: A Hardware-Based Data Prefetching Technique for Superscalar Processors. In *Proc. 29th International Symposium on Microarchitecture*, pages 214–225, Dec. 1996.

[65] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. In *Proc. 33rd International Symposium on Microarchitecture*, pages 269–280, Dec. 2000.

[66] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, Mar. 1995.

[67] E. Rotenberg, S. Bennett, and J.E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proc. 29th International Symposium on Microarchitecture*, pages 24–35, Dec. 1996.

[68] A. Roth, A. Moshovos, and G.S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[69] A. Roth, A. Moshovos, and G.S. Sohi. Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In *Proc. 1999 Internation Conference on Supercomputing*, pages 356–364, Jun. 1999.

[70] A. Roth and G.S. Sohi. Register Integration: A Simple and Efficent Implementation of Squash Re-Use. In *Proc. 33rd Annual International Symposium on Microarchitecture*, Dec. 2000.

[71] A. Roth and G.S. Sohi. Speculative Data-Driven Multithreading. Technical Report CS-TR-00-1414, University of Wisconsin, Madison, Mar. 2000.

[72] A. Roth and G.S. Sohi. Speculative Data-Driven Sequencing for Imperative Programs. Technical Report CS-TR-00-1411, University of Wisconsin, Madison, Feb. 2000.

[73] A. Roth and G.S. Sohi. Speculative Data-Driven Multithreading. In *Proc. 7th International Symposium on High-Performance Computer Architecture*, pages 37–48, Jan. 2001.

[74] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Annual International Symposium on Computer Architecture*, pages 46–53, May 1989.

[75] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based Programming for the EM-4 Hybrid Dataflow Machine. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 146–155, May 1992.

[76] Y. Sazeides and J.E. Smith. The Predictability of Data Values. In *Proc. 30th International Symposium on Microarchitecture*, pages 248–258, Dec. 1997.

[77] J.E. Smith. A Study of Branch Prediction Strategies. In *Proc. 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[78] J.E. Smith. Decoupled Access/Execute Computer Architecture. In *Proc. 9th International Symposium on Computer Architecture*, Jul. 1982.

[79] J.E. Smith and A.R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proc. 12th International Symposium on Computer Architecture*, pages xx–yy, Jun. 1985.

[80] J.E. Smith, S. Sastry, T. Heil, and T.M. Bezenek. Achieving high performance via co-designed virtual machines. In *Proc. 2nd International Workshop on Innovative Architecture*, Oct. 1998.

[81] A. Sodani. *Dynamic Instruction Reuse*. PhD thesis, University of Wisconsin-Madison, Madison, WI 53706, Apr. 2000.

[82] Avinash Sodani and Gurindar S. Sohi. Dynamic Instruction Reuse. In *Proc. 24th International Symposium on Computer Architecture*, pages 194–205, Jun 1997.

[83] G.S. Sohi, S. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proc. 22nd International Symposium on Computer Architecture*, pages 414–425, Jun. 1995.

[84] P. Song. IBM's Power3 to Replace P2SC. *Microprocessor Report*, 11(15), Nov. 1997.

[85] Y.H. Song and M. Dubois. Assisted Execution. Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California, Oct. 1998.

[86] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proc. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Jun.xx 1994.

[87] J.G. Steffan and T.C. Mowry. The Potential for Using Thread Level Data-Speculation to Facilitate Automatic Parallelization. In *Proc. 4th International Symposium on High Performance Computer Architecture*, Feb. 1998.

[88] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *Proc. 9th International Conference on Architectural Support for Programming Lanugages and Operating Systems*, pages 257–268, Oct. 2000.

[89] M. Takesue. A Unified Resource Management and Execution Control Mechanism for Data Flow Machines. In *Proc. 14th Annual International Symposium on Computer Architecture*, pages 90–97, Jun. 1987.

[90] K.B. Theobald, G.R. Gao, and L.J. Hendren. On the limits of program parallelism and its smoothability. In *Proc. 25th International Symposium on Microarchitecture*, pages 10–19, Dec. 1992.

[91] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-Driven and Demand Driven Computer Architecture. *ACM Computing Surveys*, 14(1):93–143, Mar. 1982.

[92] M. Tremblay. MAJC-5200: A VLIW Convergent MPSOC. Microprocessor Forum, Oct. 1999. http://www.sun.com/microelectronics/MAJC/documentation/docs/MPForum99-d.pdf.

[93] M. Tremblay. MAJC: An Architecture for the New Millenium. In *Proc. Hot Chips 11*, pages 275–288, Aug. 1999. http://www.sun.com/microelectronics/MAJC/documentation/docs/HC99sm.pdf.

[94] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. 22nd International Symposium on Computer Architecture*, pages 392–403, Jun. 1995.

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

[96] G. Tyson and T. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proc. 30th International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.

[97] A.K. Uht, V. Sindagi, and K. Hall. Disjoint Eager Execution: an Optimal Form of Speculative Execution. In *Proc. 28th International Symposium on Microarchitecture*, pages 313–325, Nov. 1995.

[98] A.H. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18(4):365–396, Dec. 1986.

[99] D. Wall. Limits of Instruction Level Parallelism. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[100] S. Wallace, B. Calder, and D.M. Tullsen. Threaded Multiple Path Execution. In *Proc. 25th International Symposium on Computer Architecture*, pages 238–249, Jun. 1998.

[101] D. Wilmot. Data Threaded Microarchitecture: A Radically Out-of-Order Microarchitecture for More Superscalar Performance. *Computer Architecture News*, 26(5):22–32, Dec. 1998.

[102] W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance Through Multistreaming. In *Proc. 1995 Conference on Parallel Architectures and Compilation Techniques*, Jun. 1995.

[103] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, Apr. 1996.

[104] C.B. Zilles and G.S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. 27th International Symposium on Computer Architecture*, pages 172–181, Jun. 2000.

[105] C.B. Zilles and G.S. Sohi. A Programmable Co-processor for Profiling. In *Proc. 7th International Symposium on High Performance Computer Architecture*, pages 241–252, Jan. 2001.

[106] C.B. Zilles and G.S. Sohi. Execution Based Prediction Using Speculative Slices. In *Prc. 28th International Symposium on Computer Architecture*, pages 2–13, Jul. 2001.