ENHANCING SOFTWARE RELIABILITY WITH SPECULATIVE THREADS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jeffrey Thomas Oplinger
August 2004

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Monica S. Lam
(Principal Advisor)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle A. Olukotun

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the University Committee on Graduate Studies.

Abstract

As high-end microprocessors continue to provide more and more computing power, non-performance metrics such as security, availability, reliability, and usability have become much more important. Errors and vulnerabilities in software programs have caused significant losses of data and productivity throughout the world. Software tools are available to help identify and prevent these problems, but they are often not used in practice because of large runtime overheads and limited applicability. We believe that hardware support can make these existing tools faster and more useful, as well as provide new functionality that helps programmers write safer code.

We suggest using a monitor-and-recover programming paradigm to enhance software reliability and propose an architectural design based on thread-level speculation (TLS) that makes this paradigm more efficient and easier to program. Programmers can add monitoring code, with normal sequential semantics, to examine the execution of a program. Our architecture reduces the resulting slowdown by speculatively executing the monitoring code in parallel with the main computation. To recover from errors, programmers can define fine-grain transactions whose side effects are either committed or aborted via program control. These transactions are implemented efficiently through TLS hardware support.

Our experimental results suggest that monitored execution is well-suited to this parallelization model. Applying thread-level speculation improves the running time of monitored code by a factor of 1.5. Together with a 1.9-times speedup from exploiting additional single-thread instruction-level parallelism (ILP), an overall speedup of 2.8 is obtained, effectively 6.6 instructions per cycle in performance. Through a number of real-life examples, we also show how fine-grain transactional programming can be used to detect and recover from buffer overflow exploits.

Acknowledgments

I would like to thank my advisor Monica Lam for her encouragement, guidance, enthusiasm, and exceptional last-minute paper-writing skills. On more than one occasion she has gone beyond the call of what one might normally expect from a research supervisor. I would also like to thank the other members of my reading committee, Kunle Olukotun and Mendel Rosenblum. Special thanks to Andrew Ng who graciously agreed to chair my oral defense on very short notice.

My fellow students in the SUIF group have never failed to enlighten or entertain (consciously or not) and I would like to thank all of them. I've greatly enjoyed all the officemates I've had over the, ahem, number of years I've spent at Stanford: Shih-Wei Liao, Amy Lim, Brian Schmidt, Brian Murphy, John Whaley, Costa Sapuntzakis, Ramesh Chandra, and Joel Sandin. Special thanks to David Heine, who provided exceptional assistance with my early research and, as a contemporary Ph.D. candidate, was a great person to commiserate with.

I also want to thank my friends: Pierre Louveaux, Chris Daley, Dennis Chiu, Ryan Chiao, Raj Batra, Richard Bermudez amongst others. My parents, Jim, Joanne, and Cindy, also provided an exceptional amount of support. I certainly wouldn't have completed my work without them.

This work was supported in part by DARPA contracts DABT63-95-C-0089 and MDA904-98-C-A933.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Developments in semiconductor technology and computer architecture over the past years have led to tremendous increases in computing performance. Today's high-end microprocessors offer much more computing power than the vast majority of existing software applications can fully utilize. On the other hand, non-performance metrics such as security, availability, reliability, and usability have become much more important. Errors and vulnerabilities in software programs have caused tremendous losses of data and productivity in the workplace, catastrophic mission failures and much more[39]. Researchers are beginning to look at how the abundance of computing power that is available can be leveraged to address these "quality-of-life" computing issues.

Despite much work in the past on code verification and error detection tools, it remains the case that only small amounts of critical software can be proven to be correct. As the old saying goes, "to err is human." We believe that program errors can never be fully eliminated from complex software. As a complement to code verification techniques, we advocate introducing additional code into programs to monitor whether they are behaving correctly and to recover from errors. Because code monitoring can add significant overhead to a program's execution time and error recovery code can be complex, we propose the use of computer architecture support to make this "monitor-and-recover" style of programming both more efficient and easier to write.

1.1 Contributions

In this thesis, we identify two programming idioms that can be used to enhance software reliability: the use of monitoring code to profile a program's execution in order to understand both its correct and incorrect behaviors, and the use of fine-grained transactions to safely recover from detected errors. We propose an architectural design, based on thread-level speculation, to efficiently support both of these schemes.

Thread-level speculation, or TLS, was not initially proposed with software reliability in mind; the first proposals shared a common goal of increasing the performance of traditional sequential programs via speculative parallelization[1, 17, 42, 46, 51]. In the TLS model a sequential program is divided up into threads, which may or may not be dependent upon one another. The hardware executes the threads simultaneously, but monitors the read and write operations of each thread to detect any data dependences that are violated due to the parallel execution. If a hazard is detected, the hardware aborts the thread that is supposed to execute later in the original sequential execution, eliminating all its side effects. The advantage of this approach is that the hardware can follow multiple threads of control at the same time, allowing operations that are far apart to be executed simultaneously.

Indeed, our initial interest in TLS was motivated by a desire to speed up general-purpose integer programs, but many researchers have found that broad goal to be difficult to achieve. Our first contribution is a limit study on speculative parallelism in the SPECint95 benchmarks, presented in Chapter 2. We show that single-level loops do not possess enough parallelism or coverage, in general, to allow for decent speedups. We broaden the scope of potential parallelism by considering procedure continuations as an additional target. Still, the results on an infinite machine with perfect synchronization are relatively limited.

There are still many different approaches one could take to improve the performance, such as code transformations that may expose significant additional parallelism, or the application of increasingly aggressive hardware structures. However, we instead chose to look at using the TLS hardware to tackle issues specific to software reliability, in particular by speeding up monitoring code and supporting fine-grain transactions. This leads to our further contributions as follows:

Efficient fine-grain code monitoring. To monitor the execution of a program properly, it is often necessary to invoke monitoring functions at relatively fine granularity throughout the execution of a program. Monitoring may be performed at procedural or basic block boundaries or even for all memory operations. Unfortunately, monitoring can add many more operations to a program. The run-time overhead has curbed the use of monitoring in production code, and to some extent, even in the software development and debugging cycle.

Monitoring functions need to observe the main program's state, but unless anomalies are detected, they do not have any effect on the execution of the main program. Thus, these functions can often be run in parallel with the main program as well as with each other. To keep programming simple, we propose that programmers simply identify these monitoring codes and assume that they are executed sequentially like any other functions in the program. The hardware, however, will execute these monitoring functions in parallel with the original program speculatively.

Because monitoring functions may be of potentially fine granularity, we adopt an architecture most similar to the proposed DMT machine[1]. Our proposed machine builds on the concept of simultaneous multi-threading (SMT)[55], where a dynamic scheduler allows threads to share a common pool of hardware resources. Data hazards between threads are handled by augmenting the functionality of the load-store queues. We use value prediction to minimize rollbacks due to typical data hazards found in monitoring code.

Fine-grain transactional programming. Writing error recovery code is difficult because the code must reverse any side effects that are no longer desired. Borrowing the concept of transactions from databases, we propose that portions of program execution be structured as *transactions*. Each transaction is a unit of computation whose side effects, which include all changes made to registers and memory, can be committed or discarded as a unit. Error detection and recovery can be achieved by simply applying an end-to-end check of the integrity of the computation after it is complete and rolling back the transaction to its initial state if necessary.

To support transactional programming, we introduce a number of instructions with which the software determines when to start a transaction and ultimately whether to commit or abort. The TLS machine executes the transaction as a speculative thread; the side effects are saved in the speculative buffers, which can then be committed or discarded. The size of these buffers limits the amount of side effects that can be allowed in a transaction, so we suggest using the processor data cache to buffer larger amounts of transactional state. We also present a fall-back software mechanism than can be used should the limited size or associativity of the cache prevent a transaction from completing.

Empirical validation. We have evaluated our proposed architecture through a number of case studies. Our experiments with four different examples of execution monitoring show that the hardware reduces the overhead of monitoring overall by a factor of 2.8, and even provides a respectable factor of 1.5 speedup once single-thread instruction-level parallelism (ILP) benefits are factored out. We also show that the concept of fine-grain transactional programming is useful in catching buffer overrun exploits through a number of real-life examples.

Evolution of new programming paradigms. This thesis tries to look ahead to the future and proposes architectural concepts that promote new and more effective programming paradigms that are considered impractical on today's architectures. The more traditional approach to computer architecture research has been to tune our architectures according to the characteristics of existing applications. If we only optimize for today's applications, the machines will be ill-suited to new programming paradigms; on the other hand, without an efficient implementation, new paradigms are hard to get established. Our approach represents an attempt to break this cyclic dependency; we hope this will lead to better combinations of programming paradigms and architectures that together improves our ability to create reliable software.

In the past, so much energy has been devoted to squeezing the last drops of performance out of existing applications that we are seeing diminishing returns from new architectural features that are proposed. Programs written in new paradigms have different characteristics, and thus provide opportunities for new architectural development. In particular, achieving significant performance gains by applying thread-level speculation to existing general-purpose programs has proven difficult. We show in this research that the self-monitoring programs are quite suitable for thread-level speculation.

1.2 Thesis Organization

First, we examine some of the limits of applying thread-level speculation to general-purpose integer code, as well as related work by other researchers, in Chapter 2. Chapter 3 describes our Monitor-and-Recover paradigm, with Section 3.1 discussing the concept of execution monitoring: its uses, associated programming tools, and the characteristics of the monitoring functions. Section 3.2 then motivates the need for transactional programming and proposes some concrete high-level syntax for expressing transactions. Chapter 4 describes our machine architecture. We present experimental evaluations of the Monitor-and-Recover paradigm, as well as related work, in Chapter 5. Finally we conclude with Chapter 6.

Chapter 2

General Purpose Thread-level Speculation

2.1 Introduction

This chapter describes the basics of thread-level speculation and investigates how much speculative thread-level parallelism can be found in integer programs depending on where those threads are created. Our studies show that speculating only on loops does not yield much parallelism. We evaluate the use of speculative procedure execution as a means to increase the available parallelism beyond that available in loops, as well as the potential for data value prediction improve the performance of speculative execution.

For a number of years, researchers have been exploring the use of speculative threads to harness more of the parallelism in general-purpose programs [1, 17, 24, 31, 35, 46, 51]. In these proposed architectures, threads are extracted from a sequential program and are run in parallel. If a speculative thread executes incorrectly, a recovery mechanism is used to restore the machine state. While a superscalar processor can only extract parallelism from a group of adjacent instructions fitting in a single hardware instruction window, a thread-based machine can find parallelism among many larger, non-sequential regions of a program's execution. Speculative threads can also exploit more parallelism than is possible with conventional multiprocessors that lack a recovery mechanism. Speculative threads are thus not limited by the programmer's or the compiler's ability to find guaranteed parallel threads. Furthermore, speculative threads have the potential to outperform even perfect static parallelization by exploiting dynamic parallelism, unlike a multiprocessor which requires conservative synchronization to preserve correct program semantics.

Several hardware designs have been proposed for this thread-level speculation (TLS) model, but many of the speedups reported for large, general-purpose integer code have been limited[1, 17, 24, 31, 35, 46, 51]. However, it is important to note that these experiments evaluated not only the proposed hardware, but also the choices made by the researcher or the compiler as to where to apply speculative execution. The decision on where to speculate can make a large difference in the resulting performance. If the performance is poor, we gain little insight on why it does not work, or whether it is the parallelization scheme or machine model (or both) that should be improved. As a consequence, poor results may not reflect any inherent limitations of the TLS model, but rather the way it was applied.

The goal of this chapter is to evaluate different sources of speculative parallelism running on the same machine configuration to see how they compare. To search through a large space of parallelization schemes effectively, we work with simple machine models and a relatively simple trace-driven simulation tool.

We define an *optimal* TLS machine that incurs no overhead in executing the parallel threads and can delay the computation of a thread perfectly to avoid the need to rollback any of the computation. To keep the experiments simple and relatively fast, we assume a base in-order machine that executes one instruction per cycle. Many different synchronization schemes have previously been proposed to minimize rollbacks, such as adding synchronization operations to the code statically or inserting them dynamically as rollbacks are detected, for example[10, 33]. We can use the optimal machine to derive an upper bound on the performance achievable using any possible synchronization optimizations. The optimal machine serves as an effective tool for filtering out inadequate parallelization techniques, since techniques that do not work well on this machine will not work well on a real machine of a similar design. We vary the resources available on our optimal TLS machine in the experiment, supporting 4, 8, or an infinite number of concurrent threads.

We also define a *base* TLS machine that is similar to the optimal version but makes no attempt to eliminate rollbacks through synchronization. The machine simply executes the instructions of each thread in sequence; if the data used is found to be stale, and the value was not correctly predicted, the machine restarts the thread. The performance of a particular synchronization scheme should thus fall between the bounds established by the optimal and base machines.

To explore the potential of various parallelization schemes in an efficient manner, we have created a trace-driven simulation tool that can simultaneously evaluate multiple parallelization choices. We also use this tool to collect statistical data describing the parallel behavior of the program.

Our experiments have led to the following contributions:

- We found that it is inadequate to exploit only loop-level parallelism, the form of parallelism that is used almost exclusively in many prior studies. Our tool simultaneously evaluates all possible choices of which level in a loop nest to speculate on. Even with optimal loop-level choices for speculation and optimal data dependence synchronization, the speedup obtained is low. This is due to a combination of the limited parallelism of loops in non-numeric code and the time spent in sequentially executed code found outside of the parallelized loops.

- We found that procedures can provide a useful source of speculative parallelism. In procedural speculation, another thread speculatively executes the code following the return of the procedure at the same time the procedure body is itself being executed. Procedure boundaries often separate fairly independent computations, and there are many of them in most programs. Thus, procedure calls provide important opportunities for parallelism and can complement loops as a source of thread-level parallelism.

- We also evaluated the potential of using simple value prediction to improve speculative parallelism. In particular, predicting the value returned by a procedure (return value prediction) can greatly reduce the data dependences between the procedure execution and the following computation. Value prediction can also eliminate important data dependence constraints across iterations of speculatively executed loops.

This work shows some of the limitations of certain parallelization schemes, such as parallelizing only loops, and evaluates the potential benefits of procedural speculation and value prediction. As the related work in Section 2.8 discusses, further TLS proposals have investigated additional sources of parallelism from more arbitrary code regions as well as the benefits of more aggressive value prediction. Many of these schemes require significant additional hardware in order to be successful, however.

The rest of the chapter is organized as follows. In Section 2.2 we describe the TLS machine model in more detail, followed by the simulation methodology in Section 2.3. In Section 2.4 we present results on using the optimal TLS model to exploit loop-level parallelism. We investigate the use of procedural speculation in Section 2.5. Section 2.6 contains the results of combining loop-level and procedure-level speculation. In Section 2.7 we present the performance results of both the optimal and base machines with finite resources. Related work is discussed in Section 2.8 and we conclude in Section 2.9.

2.2 The TLS Machine Model

In the TLS machine model, a normally sequential execution is explicitly divided into multiple threads which are run in parallel. This process is shown in Figure 2.1, where the sequential execution in (A) is divided into four threads for concurrent execution in (B). Note that the first thread in the machine, thread 1, comes earliest in the original sequential execution. It is not dependent on any other thread in the machine, so we know its execution will be correct and thus it does not execute speculatively. There is always just one such thread in the machine at a time, so we refer to such a thread as the *sequential* or *head* thread. Threads 2, 3, and 4 in the example are each executing speculatively. As the current head thread retires, each will successively become the new head thread in turn, as long as their execution is validated.

To preserve correct sequential execution semantics, the side effects of each speculative thread are saved in a separate speculative state. Each thread observes all write

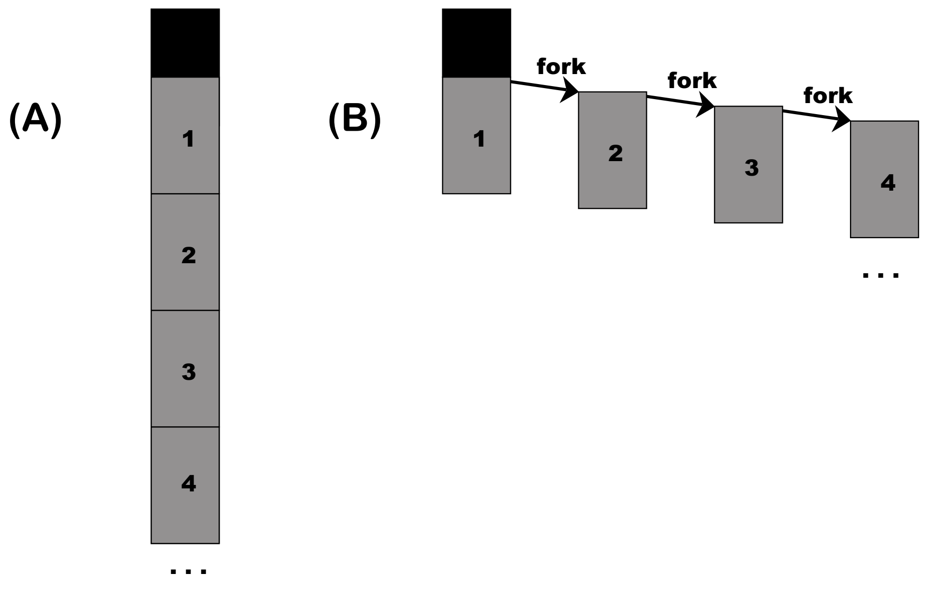


Figure 2.1: Normal Sequential Execution and the TLS Counterpart

operations from threads that occur earlier in the sequential execution sequence, using the most recent values seen at each point, and detects dependence violations once they occur. There are two forms of dependence violation:

- (true) data dependence. A thread detects a dependence violation if it discovers that an earlier thread has generated a value that it needs after it has already speculatively computed with an outdated value. A memory data dependence violation is shown in Figure 2.2(A). Conflicts due to true dependences are the only form of data dependence violation observed. Anti-dependences (or storage dependences) do not cause a violation. That is, write operations by a later thread do not affect the values read by an earlier thread, and therefore these operations need not be ordered. Similarly, actions of earlier threads have no effect on a later thread that first writes then reads the same location.

- control dependence. A thread detects a control dependence violation if the control flow of an earlier thread causes the subsequent thread not to be executed at all. For example, an early exit taken by an iteration of a loop would render

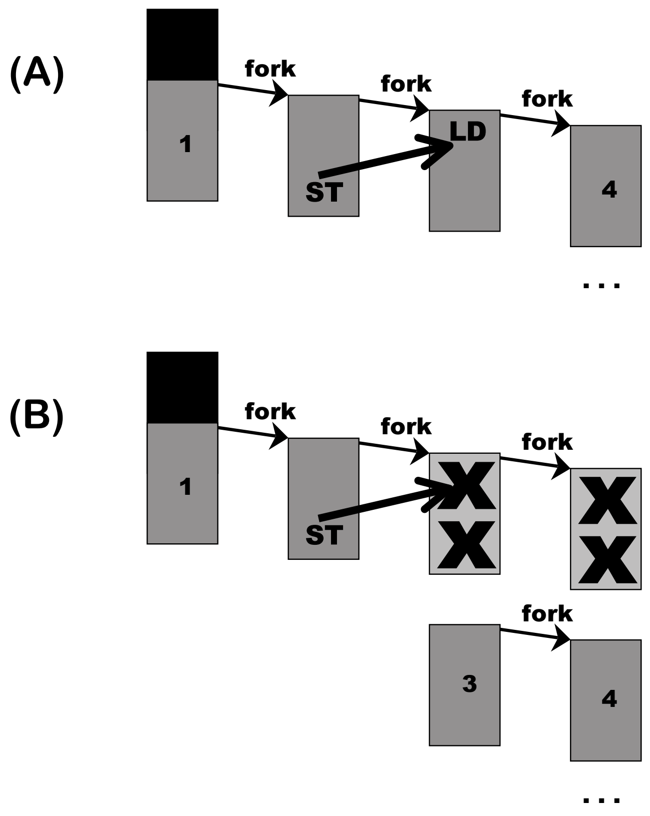


Figure 2.2: TLS Re-execution Following a Data Dependence Violation

the speculative execution of all subsequent iterations invalid.

If a violation occurs, the processor throws away the speculative state and, in the case of a data dependence violation, restarts the thread, as shown in Figure 2.2(B). If there are no violations, the speculative state is committed when all threads representing earlier segments of the sequential execution have committed.

Based on this ordering of the threads by their expected occurrence in the sequential execution, we sometimes refer to the threads as *more-* or *less-speculative*, or *lower-* or *higher-priority*. Referring to the preceding figures, thread 1 would be considered *least-speculative* (it is in fact non-speculative) and *highest-priority* as well, as its execution is guaranteed to be correct. Thread 4 represents speculation furthest out in the execution stream from what we know to be correct; it may be forced to restart due to a data or control dependence violation with any of the other threads. Thus we would refer to thread 4 as the *most-speculative* or *lowest-priority* thread in the machine.

Clearly data dependences can significantly limit the thread-level parallelism that is available. Value prediction is particularly relevant to TLS as it enables a program to run faster than the dataflow limit determined by data dependences[27]. We examine two simple schemes of value prediction. The last-value prediction (LVP) scheme predicts that the loaded value will be the same as the value obtained from the last execution of that load instruction. The stride-value prediction (SVP) scheme predicts that the loaded value will be the same as the last loaded value plus the difference between the last two loaded values. By using value prediction, a speculative thread need not be rolled back upon detecting a true data dependence violation if the predicted value matches the newly produced value. To find the upper bound of the impact of value prediction on TLS, we assume that the machine automatically uses value prediction whenever it attempts to read a value that has not yet been produced. As this is a limit study, we also assume that the machine has a buffer large enough to hold all predicted values needed by any thread.

2.3 Simulation Methodology

In this section we describe our experimental setup, specifically the simulation tool that was developed as well as the benchmarks used to evaluate thread-level speculation.

2.3.1 Simulation Tool

We use a trace-driven simulation tool to evaluate the performance of different speculative execution schemes under different machine models. For the sake of simplicity, we assume that each processor takes one cycle to execute each instruction. Also, to focus on the potential of the different parallelization schemes, we assume that the system has a perfect memory hierarchy. All memory accesses can be performed in a single clock and stored data is immediately available to all processing elements in the

Region(s)	Description
loops	a single loop in each nest is chosen for speculation
multi-level loops	all loops in each nest are chosen for speculation
procedures	procedure bodies execute in parallel with the code following them
loops and procedures	speculation on a single loop in each nest as well as on procedures

Table 2.1: TLS Parallelization Schemes

next cycle. There is no overhead in the creation of threads and no additional cycles are needed to restart a thread once a violation is detected.

To explore the TLS design space, we vary our simulations based upon the types of regions speculation is applied to (single-level loops, multi-level loops, procedures, loops and procedures), whether or not dependence synchronization is employed, the maximum number of concurrent threads allowed to execute, and the value prediction policy for memory loads and register reads. The parameters to our simulator are summarized in Table 2.1 and Table 2.2.

We use the ATOM tool[48] to augment the optimized program binaries and generate a user-level trace that includes events of interest: loads and stores, procedure entries and exits, and loop entries, exits, and iteration advances. (System calls are not captured in the trace.) The simulation clock normally advances one cycle per instruction. Procedure and loop entries signal the potential forking of a speculative thread, depending on the parallelization scheme used. When a speculative thread fork is encountered, the simulation time is stored and the first thread executes. When the later thread(s) from the fork begin to execute, the simulation time is set back to the time of the fork. When a store or register write occurs, the current simulation time is recorded for that memory/register location. Execution continues as normal, except threads are delayed or squashed if they try to read a value that was written at a time greater than the current simulation time. Delays are avoided in the models employing value prediction whenever the values are correctly predicted. Value prediction is implemented by keeping an array of the last two loaded values for each load and register use in the program, and by keeping another array of the last two returned values for

Parameter	Description
Synchronization Policy	
optimal	all operations delayed optimally to avoid rollback
base	no operations are ever delayed, threads are rolled back as soon as a violation is detected
Resources	
Infinite	no limit on the number of concurrent threads
8-way	resources to execute 8 concurrent threads
4-way	resources to execute 4 concurrent threads
Return Value Prediction	
none	no return value prediction
LRVP	last value prediction of return values whenever procedural speculation is used
LVP	last value prediction of return values whenever procedural speculation is used
SVP	stride value prediction of return values whenever procedural speculation is used
Value Prediction for Memory Loads and Register Reads	
none	no value prediction
LRVP	no value prediction (except for return values as above)
LVP	last value prediction
SVP	stride value prediction

Table 2.2: TLS Machine Model Parameters

each procedure in the program. The value prediction employed is rather idealistic, in that we presume perfect "confidence estimation"–when the prediction is incorrect, the machine simply behaves as if prediction were not employed at all, so misprediction never causes any performance penalty. Additionally, because the simulation is based on the sequential program trace, there is no notion of the value predictor being updated "out of order" as would undoubtedly happen in a real speculative machine. A prediction is always based on the last one or two instances of that instruction in the sequential trace. Thus the performance results with prediction should be considered as an upper bound on the benefit that prediction could provide.

In simulations of finite-processor models, a resource table is used to track usage of the individual processors. A thread must obtain execution cycles from a resource table before it is allowed to execute. Threads closer to the sequential execution are given priority and may preempt lower priority threads in progress if resources are exhausted. Preempted threads are delayed until the next available cycle and are re-executed from the start.

When the actual number of loop iterations is not known a priori, a real TLS machine would end up wasting some resources executing beyond the end of the loop. To account for the wasted cycles on mispredicted iterations, which do not show up in our sequential trace, our simulator treats the end of a loop as a barrier to parallelization in simulations of finite machines. That is, the machine is not allowed to speculatively execute computation that logically follows the loop in the original sequential execution. No such barrier exists in the procedural case because only one new thread is created at each procedure call.

One final consideration is that the code generated by the compiler for a uniprocessor machine includes many "artificial" dependences. For example, the multiple threads in a TLS machine operate on multiple stack frames at the same time, so they need not obey the dependences due to global and stack pointer updates in the sequential program. Similarly, since the threads have separate register files, they need not be constrained by the dependences introduced by a called function's register save and restore operations. Figure 2.3 shows a pseudocode example, where the " `<-` " notation represents assignment. The original uniprocessor code representing a register write, procedure call, and subsequent register read, together with the definition of the called procedure, is shown in the leftmost column. In the two right columns, we show a sample two-thread execution utilizing procedural speculation, which is described in Section 2.5. Here, thread 2 speculatively executes the code that comes after the call to `foo()`. In a true TLS machine, thread 2 would have it's own copy of the `r0` register from before the call to `foo()`. Thus, our simulator ignores dependences originating from a register restore operation (e.g. the dashed arrow in Figure 2.3), and instead observes only the true dataflow dependence (e.g. the solid arrow in Figure 2.3) that the save/restore code is designed to preserve in a uniprocessor execution.

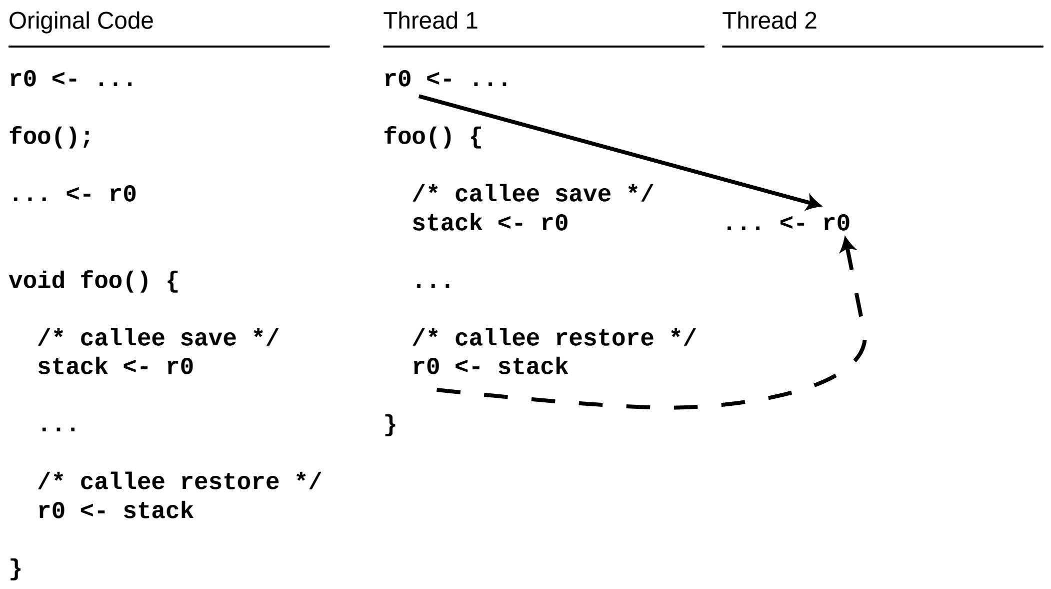


Figure 2.3: Dependences Induced by Callee-Saved Register Operations

2.3.2 Benchmarks

We evaluate the performance of our various speculative thread models using the 8 benchmarks from the SPECint95 benchmark suite. Table 2.3 lists the programs, input sets, and execution characteristics. Throughout the chapter, speedups are calculated relative to a single cycle per instruction sequential execution of the program.

All the programs were compiled using the HP/Compaq Alpha `cc` compiler with

Program	Lines of Code	Input Set	Dynamic Instr	Description
compress	1K	train	47M	File compression
gcc	192K	train	286M	The GNU compiler
go	29K	train	54M	Game of go
ijpeg	28K	train	183M	Image compression
li	7K	train	136M	Lisp interpreter
m88ksim	18K	test	134M	Processor simulator
perl	23K	train	5M	Perl language interpreter
vortex	52K	train	963M	Database

Table 2.3: Benchmarks Executed

optimizations using the `-O2` flag. To perform the simulation, we use ATOM to annotate the binaries with information such as the entry and exit points of loops as well as locations for `setjmp` and `longjmp` calls. The annotation tool analyzes the binary code directly, extracts control flow graphs from the code and calculates the singly entry and potentially multiple exits of the loops using standard compiler algorithms. Recognizing all induction variables in the binary, however, would require an interprocedural analysis that we have not implemented, so induction variables are not recognized. Note that machines employing stride value prediction will effectively recognize the induction variables and ignore most of their dependences.[1] For machines with last value or no value prediction, the loop iteration counting code generated by a typical uniprocessor compiler will result in at least one data dependence across iterations that needs to be synchronized, even if the loop is otherwise parallel. Thus the performance that one might obtain by using a compiler that better handled induction variables should be roughly bounded by the results utilizing stride value prediction and the results without value prediction.

2.4 Speculative Loop Parallelism

Loop iterations are the traditional target of parallelization and an obvious candidate for thread-level speculation. Each iteration of a loop can be turned into a speculative thread that runs in parallel with the other iterations of that loop. The only form of control dependences shared between iterations are loop termination conditions, and the outcomes are highly skewed in favor of continuation. The remainder of the control flow in each iteration is independent; thus failure to predict a branch within an iteration does not affect other threads. The degree of parallelism available in a loop is governed by the presence of data dependences that cross loop iteration boundaries. If the iterations operate on disjoint sets of data, the degree of parallelism can be equal to the number of iterations of the loop. In the following, we first focus on the common model of applying speculation to one loop at a time, beginning with a discussion of

[1]The stride predictor we use requires two iterations in most cases to learn a new stride value, but after that all predictions will be correct if the stride amount does not change.

how we select such loops. We then look at the performance and hardware implications of allowing multiple loops to speculatively execute in parallel.

2.4.1 Choices for Single-Level Loop Speculation

When we restrict speculation to a single loop in a nest, the critical decision is which loop in the nest to speculate on. There are two factors that need to be considered when selecting the best loop.

- *Degree of Parallelism* : there must be sufficient data independence between the iterations to achieve parallelism. If the iterations are totally independent (a DoAll loop), then the potential degree of parallelism is equal to that of the number of iterations. If there are dependences across iterations (a DoAcross loop), the degree of parallelism is dependent upon the ratio of the length of the recurrence cycle to the length of the iteration.

- *Parallelism Coverage* : If we parallelize an inner loop, then all the code outside will run sequentially. Thus, it may be desirable to choose an outer DoAcross loop with less parallelism over an inner DoAll loop if speculation can only be applied to one loop at a time. We refer to the percentage of code executed under speculative execution as the parallelism coverage. By Amdahl's Law, low parallelism coverage necessarily results in poor performance.

To select the best loop, we developed a separate trace-driven tool called MemDeps[35]. This tool is quite similar to our main simulator, described in Section 2.3, configured to speculate on loops and using optimal synchronization. MemDeps, however, simultaneously evaluates *all* loop levels in a given dynamic loop nest to determine which single level is best to speculatively parallelize.[2] The complexity of the simulation algorithm lies in evaluating these loop choices to while utilizing just a single run through a program execution trace. The MemDeps tool keeps separate simulation times for each currently active loop as if that were the only loop to be speculating. For each loop that an executed instruction is dynamically nested

[2]Note that loops in a dynamic nest need not be defined in the same procedure.

in, the algorithm tracks the current iteration number and individual simulation time assuming that only that single loop was executing speculatively. This per-currently-active-loop information is recorded for each location stored to in the simulation. When a load operation is encountered, the tool compares the current iteration counts with those of the last store to that location. The loop level that *carries* the dependence is the first common loop level where the iteration counts *differ* between the load and store. If the time of the store in that loop level is later than the current time, we advance the simulation time of the current iteration in that loop (containing the load) by the minimum execution delay. Note that a dependence can only be carried by one loop level, and only restricts single-level TLS execution for that specific loop in the nest. If we were to speculate only on a non-carrying level of loop, the read in question would automatically be executed after the relevant store operation even without synchronization.

When an innermost loop terminates, it passes to its surrounding loop the length of its computation and the degree of parallelism it would have achieved had it been speculatively executed. By collecting all such information from its inner loops, an outer loop can determine whether it is more profitable for it speculate or to instead suppress speculation in order to allow its inner loops to speculate. It chooses the option providing the best performance and, upon termination, informs its surrounding loop of the best performance available had speculation been applied to either itself or its inner loops. Eventually this information propagates back to the outermost loop level and determines the best overall loop selection for single-level speculation. Our algorithm handles recursion by assuming that speculation can be applied only to the outermost execution of a loop that recursively invokes itself again.

Note that different dynamic executions of a certain loop nest may result in different runtime loop choices by MemDeps. At the end of the MemDeps simulation, we calculate the overall frequency with which each loop was dynamically chosen as the best loop. These overall frequencies are then used to make static choices for loops in our simulations of the TLS machine models employing single-level loop speculation.

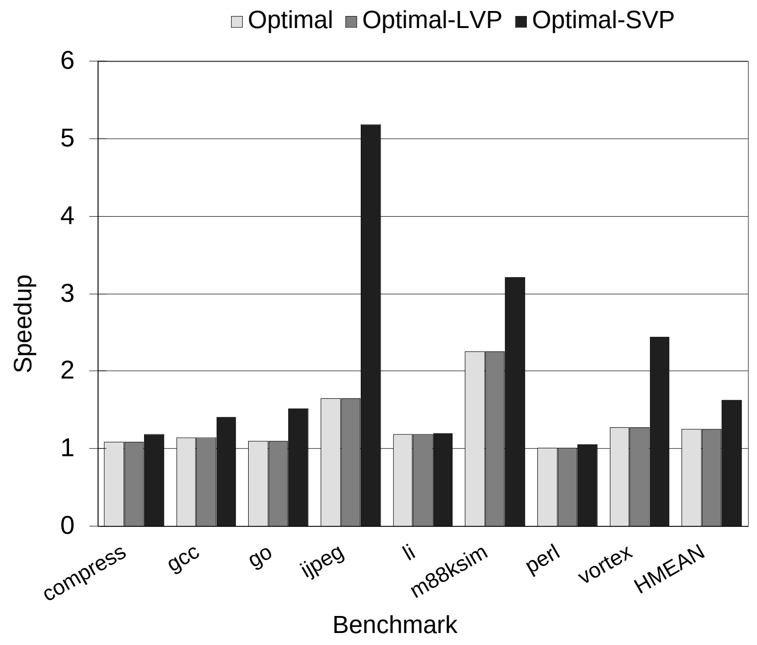


Figure 2.4: Optimal TLS Speculation on Single-Level Loops

2.4.2 Single-Level Loop Speculation

We evaluate the performance of the one-level loop speculation model, using loops selected as in the previous section, on several variants of the optimal TLS machine using the SPECint95 benchmark suite. Figure 2.4 presents the experimental results of this study. Machines are denoted by their synchronization policy and memory load prediction scheme, if any, as described in Table 2.2.

The largest speedup of 5.2 is achieved by `ijpeg`, an image compression program, with stride prediction enabled (Optimal-SVP). The significant performance improvement seen with stride prediction is due to the elimination of induction variable dependences across iterations. (Last-value prediction has no effect on these variables.) Had the code been compiled for a TLS machine explicitly, the compiler would recognize many of these induction variables and would eliminate their dependences from the program. If a TLS machine used induction variable recognition but not general value prediction, its performance would be bounded by the Optimal and Optimal-SVP results.

It is not surprising that `ijpeg` performs well as its algorithm is very parallel. `M88ksim` and `vortex` are the only other programs with speedups over 2, with the rest of the benchmarks performing only between 1 and 1.6 times better than sequential execution. Note that `li` and `perl` are relatively unaffected by value prediction.

Despite the optimal TLS machine's ability to speculate loops perfectly, the overall harmonic mean of speedup achieved across the benchmark suite is only 1.6. With the exception of `ijpeg`, the results are rather disappointing especially when considering that the optimal TLS machine uses an unbounded number of processors, delays every operation optimally, and has zero-communication cost. Moreover, the loop choices are made by analyzing the execution of the program with the same input set. Unless large changes are made to the code, speculating at only one level in each loop nesting will not yield significant speedup on a realistic TLS machine. Different code generation or instruction scheduling could provide a potentially higher limit, however.

To gain more insight into the performance results, we instrumented the code and the simulator to collect various characteristics of the individually speculatively parallelized loops[35]. We determine the computation time of the loops speculated on, whether the loops are innermost or outer loops, and whether the loops are DoAcross or DoAll loops. We summarize the findings of those experiments here.

- *Poor overall performance due to poor parallelism coverage.* Many of the programs have loops that show fairly impressive speedups when speculatively executed. However, the lack of parallel coverage, shown in Table 2.4, results in overall performance that is much lower. For example, `m88ksim` with stride prediction achieves an impressive 34.4-times speedup on 71% of the program, but the serialization in the other 29% of the computation drags down the overall performance. To illustrate the importance of coverage, we also present the Amdahl's Law limit on the overall speedups in Table 2.4, which assumes infinite speedup of the covered portions of the programs and sequential execution elsewhere. Even if all the parallelized code was executed in a single cycle, most programs would still show relatively modest overall speedups.

- *Trade-off between speedup and coverage.* In many cases, choosing the best loops

Program	% Parallelized	Amdahl's Law Max Speedup
compress	27%	1.4
gcc	77%	4.4
go	67%	3.0
ijpeg	99%	100.0
li	45%	1.8
m88ksim	71%	3.5
perl	85%	6.7
vortex	93%	14.3

Table 2.4: Coverage and Maximum Theoretical Speedup for Single-Level Loops

for overall program speedup presented a distinct trade-off. Inner loops tended to have higher speedups but lower parallel coverage, while outer loops covered more of the program but had lower speedups.

- *Lack of DoAll loops.* Only the two best performing programs (`ijpeg` and `m88ksim`) spend more than 10% of their execution time in DoAll loops. Most of the integer programs have only DoAcross loops, which tend to have a lower degree of parallelism.

The analysis of the above suggests that large performance gains will require additional sources of speculative parallelism. More sources will not only increase parallelism coverage, but will also enable the machine to exploit parallel inner loops more effectively.

2.4.3 Multi-Level Loop Speculation

Speculatively executing multiple loops in a nest seems to be an obvious approach to improving single-level loop performance, but there are many difficulties in practice. First, the relatively small number of processors we are targeting (4 or 8 in our later experiments) make it difficult to assign them to multiple loops at a time. In addition, because the number of iterations in a loop is not always known a priori, some loops would occupy the entire machine with "potential" iterations. Finally, our thread

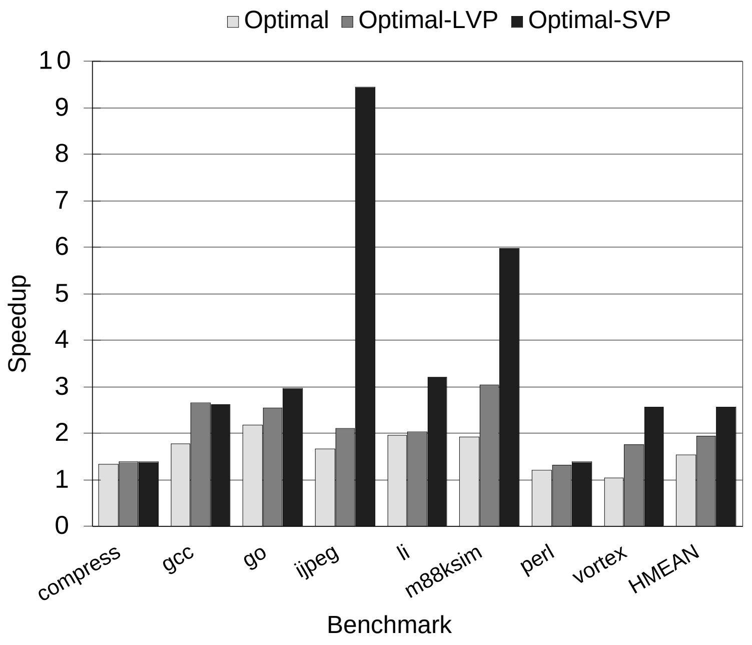


Figure 2.5: Optimal TLS Speculation on Multi-Level Loops

prioritization favors speculative threads that are closest to the current head thread in the original sequential execution. Thus, inner loop threads (closer to the head thread) would tend to force outer loop threads out of the machine. Some researchers have in fact implemented multi-level loop speculation in specific cases where an inner loop is only occasionally (conditionally) executed[4][40]. Despite the feasibility issues with multi-level loop speculation, we wish to find a bound for this approach. In this experiment, we consider the extreme case where the optimal TLS machine uses an unbounded number of processors to simultaneously execute all the iterations of each loop in a nest that it encounters. The results are shown in Figure 2.5.

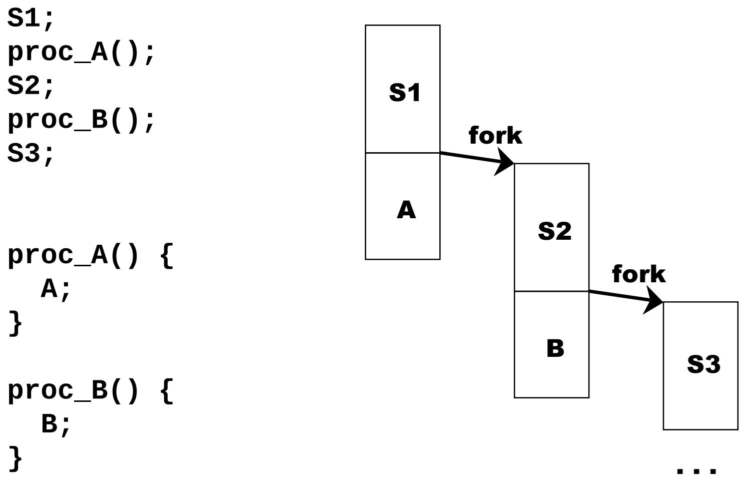
Even with all its idealistic characteristics and its use of a very aggressive speculation model, the optimal TLS machine's performance is still relatively poor. The harmonic mean improves only modestly from 1.6 to 2.6 (with stride prediction) when speculatively executing all loops simultaneously, while the machine design becomes much more difficult. Algorithms based on nested-loop computations, such as the two-dimensional image processing in `ijpeg` or the processor simulation that spans both time (processor cycles) and space (arrays of hardware structures) in `m88ksim` lead them to benefit most from the nested speculation opportunities. The less loop-oriented and more arbitrary computations in `gcc` and `perl` show relatively little improvement.

2.4.4 Summary of Speculative Loop Level Parallelism

Our results show that speculatively executing one loop at a time will not yield significant speedup under the TLS model. Moreover the performance is highly sensitive to the way the code is written. Most integer codes have DoAcross loops which have limited parallelism. The ability to speculate on just one loop in each nest limits the parallel coverage which produces a lower overall speedup. Parallelizing multiple loops simultaneously increases the coverage and the overall performance, but would be very difficult to effectively support in a real machine. Overall, the performance on a very idealistic system is still modest. This result strongly suggests that loop-level speculation needs to be complemented with other sources of parallelism.

2.5 Procedure Level Speculation

Procedures are the programmer's abstraction tool for creating loosely-coupled units of computation. This suggests that it may be possible to overlap the execution of a procedure with the code normally executed upon the return of the procedure. A characteristic that makes speculative procedure execution particularly attractive is the lack of control dependence between the sequential and speculative threads. Procedures are expected to return under normal execution, and thus it is seldom necessary to discard the speculative work because of control flow. The only exceptions are when the procedure raises an exception or uses unconventional control flow operations such as `longjmp`. These unusual circumstances can easily be handled by simply aborting the speculative threads and discarding their speculative states.

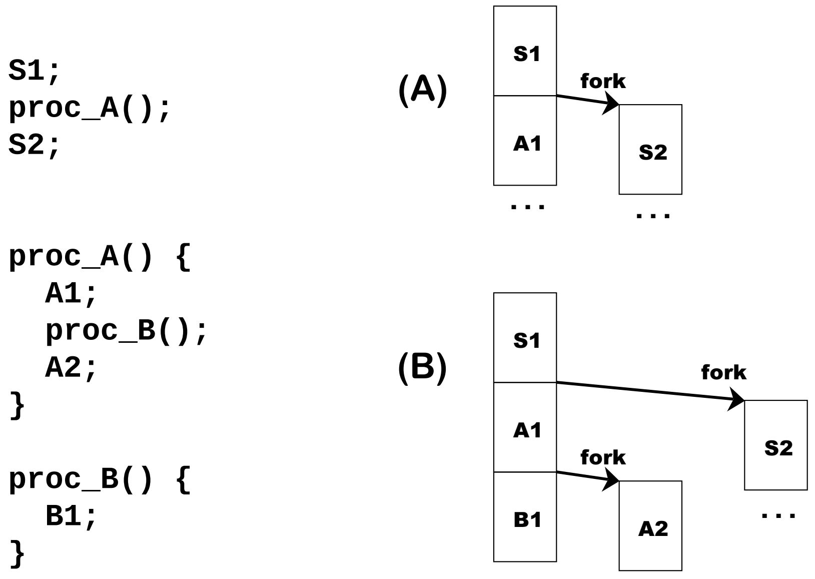


```
S1;
proc_A();
S2;
proc_B();
S3;


proc_A() {
  A;
}

proc_B() {
  B;
}
```

Figure 2.6: Procedural Speculation

Unlike loops, procedures have not traditionally been a popular target of parallelization. They have generally been used in more functional programming environments where there are fewer memory side effects in procedures and recursion is more common[16, 23]. In typical imperative programming environments, procedures tend to share more data dependences with their callees. Also, as recursion is less predominant in imperative programs, the available parallelism is not scalable. These limitations, however, are much less important to the TLS model. The speculative execution hardware can handle memory dependences that might exist across procedures. Furthermore, a real TLS machine is likely to have hardware support for only a small number of concurrent threads. The prevalence of procedure calls throughout programs provides a potentially effective source of parallelism that might complement loop-level parallelism.

To speculate at the procedure level in the TLS model, we concurrently execute the called procedure with the code following the return of the procedure, as shown in Figure 2.6. Notice that it is the latter that executes speculatively. A data dependence violation occurs if the code following the return reads a location before the callee thread writes to that location. The same mechanism that is used for loop-level parallelism can be used to ensure that the data dependences are satisfied. By customizing the procedure calling convention to support speculation, the overhead



```
S1;
proc_A();
S2;


proc_A() {
  A1;
  proc_B();
  A2;
}

proc_B() {
  B1;
}
```

Figure 2.7: Nested Procedural Speculation

of creating a new thread could be minimized. For example, if threads have their own private registers, then register saves and restores at procedure boundaries could potentially be eliminated. This limit study does not take advantage of this optimization opportunity, but we do investigate it for monitoring code optimization as later described in Section 4.5.

While loop level speculation can occupy an arbitrarily large number of processors by assigning a new iteration to each processor, each instance of procedural speculation creates work for only one additional thread. To create more opportunities for parallelism, these speculative threads from procedure calls can be created in a nested fashion. This is shown in Figure 2.7. When we first encounter the call to `proc_A`, we fork the computation as shown in (A). When we arrive at the nested call to `proc_B`, we create another new thread that is inserted in the thread ordering shown in (B). Thus the order of thread creation is not the same as the sequential order that the threads will retire in. The sequential ordering of the recursively created threads is determined as follows. If thread T_i creates a speculative thread T_j at a call site, then T_j comes after T_i in the ordering, and inherits from T_i all of its sequential ordering relationships with existing threads.

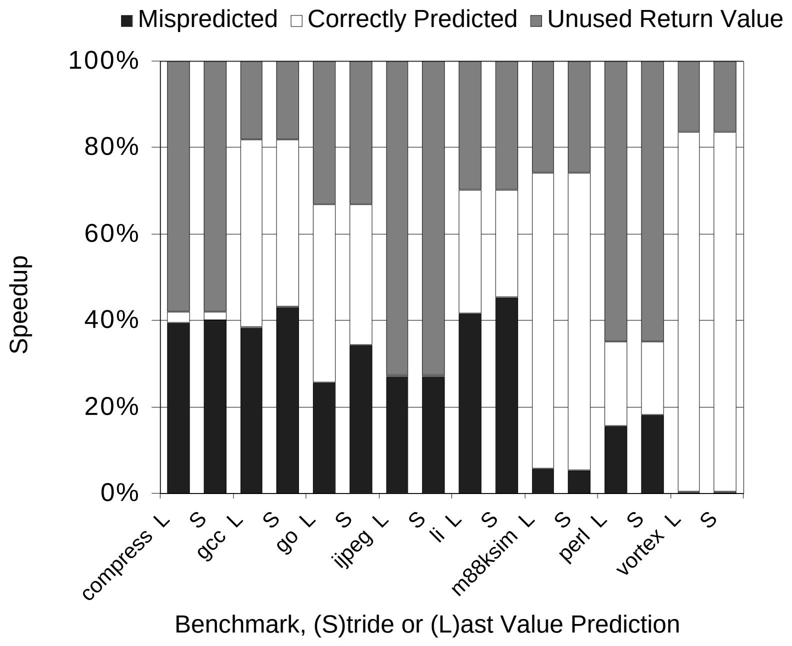


Figure 2.8: Predictability of Procedure Return Values

Because the return value is often used immediately upon the return of a procedure, speculatively executing the code following the procedure body could result in a large number of rollbacks. To avoid these rollbacks, we propose predicting the value that will be returned. Return value prediction is implemented by keeping a cache of past values returned by each procedure, if they exist. The caller thread continues to execute the code following the procedure call using the predicted return value. When the callee thread returns, the actual return value is compared to the predicted value. If the values are different, the machine would generate a data violation signal, discard the speculative state, and restart the thread.

2.5.1 Predictability of Return Values

To verify that speculation with return value prediction has potential, we first look at the predictability of those values. We experimented with two simple schemes of prediction: last-value prediction and stride-value prediction. The results are shown in Figure 2.8. We classify procedure returns into three categories: (1) those that have either no return values or whose return values are not used, (2) those whose return values are used and are correctly predicted, (3) those whose return values are used and are mispredicted. For each program, we show two sets of data, one that uses last-value prediction labeled "L" and one that uses stride-value prediction labeled "S".

First, we observe that both last-value and stride-value prediction give similar results, with those of last-value prediction being slightly better for half of the programs. Misprediction of return values occurs less than 50% of the time for all programs, with `vortex` and `m88ksim` having almost no mispredictions. The benchmarks where return value prediction most often fails typically return pointers or other memory/storage related values. For example, `compress` makes many calls to a hash function whose results are highly unpredictable. Those that are extremely predictable tend to return quantities like status/error conditions, as in `vortex` for example. Finally, note that just because a return value is correctly predicted does not imply that much of the callee and caller computation will be overlapped; if the procedure modifies global variables or reference parameters after the caller has speculatively read such data, and the values read are not predictable, then the caller thread will be rolled back.

2.5.2 Evaluation of Procedural Speculation

Our next experiment evaluates the speedups of the procedural speculation model on optimal TLS machines with different value prediction policies.

The results are shown in Figure 2.9. First, by comparing the performance of Optimal and Optimal-LRVP we observe that result value prediction has a significant positive effect on the performance of procedural speculation. As expected, programs with highest numbers of used and correctly predicted return values (`vortex`, `m88ksim`, as well as `go` and `gcc` to a lesser extent) benefit significantly. Conversely, `compress`, whose return values are not predictable, shows almost no improvement. `ijpeg` shows essentially no improvement with return value prediction because its most frequent routines (discrete cosine transforms) do not return any values.

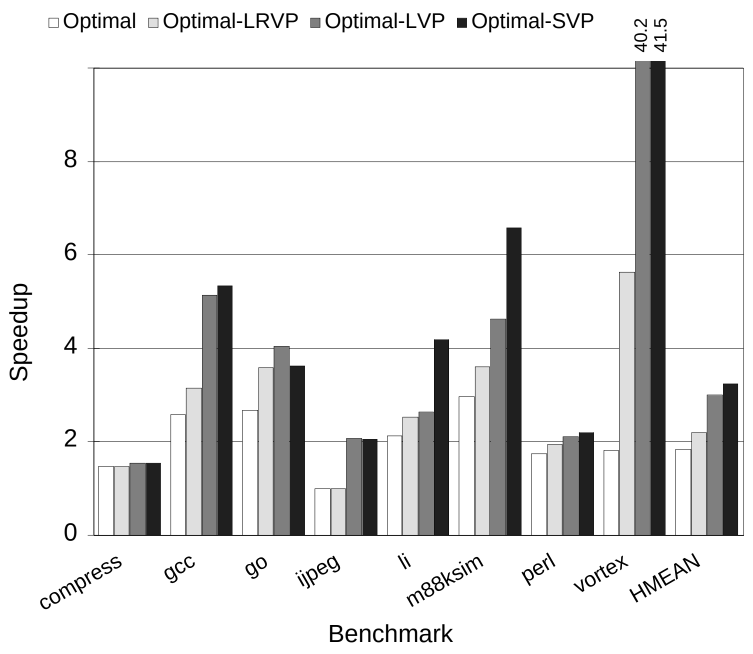


Figure 2.9: Optimal TLS Speculation on Procedures

Value prediction on regular data accesses is useful for almost all the programs, and can sometimes make a dramatic difference to the performance as in the case of `vortex` and to a lesser extent `m88ksim` and `gcc`. To gain some insight on this issue, we analyzed the code for `m88ksim`, a program that simulates a microprocessor. We found that the load instruction that benefits most from stride value prediction is a load of the simulation's program counter (at the beginning of the `datapath()` procedure). Since the program counter typically increments by 4 each time `datapath()` is called, stride value prediction is quite effective in eliminating this dependence.

We also investigated the behavior of `vortex`, which has abundant parallelism when prediction is enabled. The dominant procedure in `vortex`, `Chunk_ChkGetChunk`, accounts for about 18% of the total execution time. The procedure verifies that a memory chunk access was valid and sets a call-by-reference parameter, *status*, to indicate the type of error if any. The return value is a boolean version of status. Given that the error conditions rarely occur, this is an excellent procedure for speculation. Note that prediction of both the return value and the call-by-reference out parameter

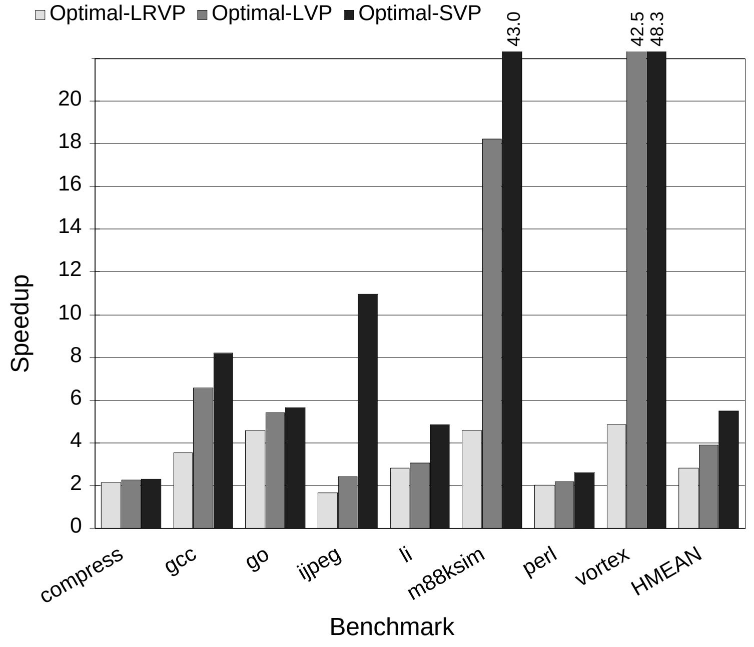


Figure 2.10: Optimal TLS Speculation on Loops and Procedures

is needed to make the threads completely parallel.

Overall, the experiments suggest that procedures are a reasonable source of speculative parallelism in many programs. With the use of return value prediction, speculating at procedural boundaries delivers a performance comparable to that of executing all loops speculatively on the various Optimal TLS models. Value prediction of regular memory accesses improves the overall speedup for almost all programs and has a major impact on specific programs.

2.6 Speculating at Both Procedural and Loop Boundaries

We next investigate the effect of combining both procedure and loop-level speculation. The experimental results for the various optimal TLS models are shown in Figure 2.10, and they are much more encouraging. Most of the programs improve significantly over speculation on loops or procedures only, showing benefits from both forms of speculation. All but `compress` and `perl` have at least a 4.5-times speedup under the Optimal-SVP model. This includes programs, such as `gcc`, which have been very difficult to parallelize previously. As noted before, value prediction can have a significant effect on specific programs.

2.7 Experimenting with More Realistic Models

Having shown that speculation at all loop and procedure boundaries exposes a reasonable amount of parallelism in an optimal TLS machine, we now experiment with this software model on more realistic machine models. In the following sections, we first evaluate the speculative scheme on an optimal TLS machine with a finite number of processors, and then on the base TLS machines that may require rollbacks.

2.7.1 A Finite Number of Processors

An optimal TLS machine with an unbounded number of processors favors the creation of as many threads as possible. In the degenerate case where every single instruction is a thread of its own, the results would be identical to those reported by previous oracle studies where each operation is issued as soon as its operands become available[58] . Speculating at all procedure and loop boundaries can easily generate more threads than a reasonable implementation could maintain.

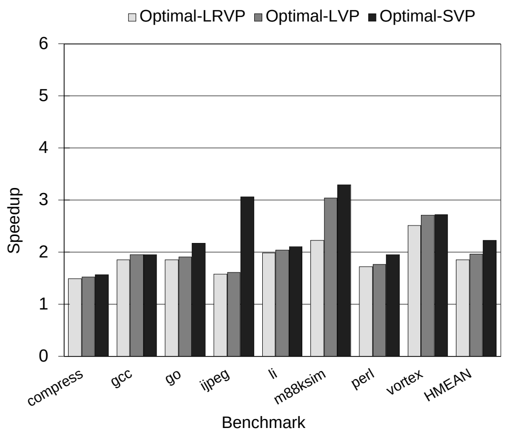
On a machine with support for only a finite number of threads, we must have a strategy to prioritize among the available threads. We adopt the simple strategy of prioritizing the threads according to their sequential execution order; a thread earlier in the sequential execution order has a higher likelihood of success and is thus given higher priority. In the presence of recursive procedural speculation, a newer thread may have a higher priority than an existing thread. When that happens, the machine frees up a processor for this thread by terminating the speculative execution of the thread with the lowest priority and discarding its speculative state. When the machine has some free resources, it will (re)start the execution of the thread with the highest priority. With this strategy, speculation on inner loops can occupy all available resources and thus prevent any speculative execution progress in the outer loops–effectively resulting in single-level innermost loop speculation. In cases where the coverage or parallelism of an outer loop is more compelling, allowing inner loop speculation to "preempt" outer loop speculation would not be desirable. To address this, we suppress the speculation of any inner loop if an outer enclosing loop was determined by the MemDeps simulator described in Section 2.4.2 to provide a better overall speedup under single-level speculation.

Figure 2.11 shows the performance achieved with 4-way and 8-way TLS machines. As expected, the speedups are lower than those found with infinite processors. `M88ksim`, `vortex`, and `ijpeg` perform quite well, delivering over a 5-times speedup on an 8-way machine and roughly 3-times speedup on a 4-way machine (both with stride prediction). Value prediction continues to benefit the same programs that saw improvement in the infinite processor case, but the gains are much more realistic and limited. The program `compress` suffers little degradation, but its performance with infinite processors was quite low to begin with. Overall, the harmonic mean of the speedups is 3.2 for 8 processors, and 2.3 for 4 processors.

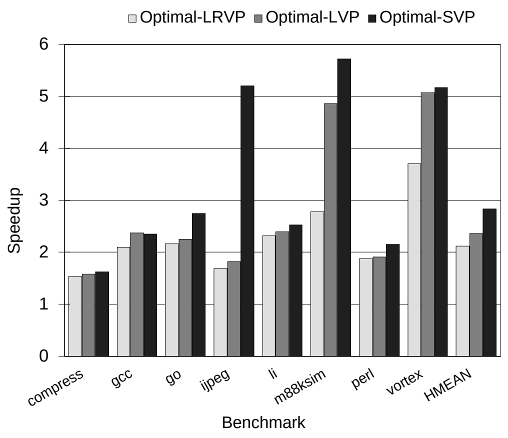
2.7.2 Machines with Rollbacks

The most unrealistic aspect of the optimal TLS machine is that it automatically delays every operation by the optimal amount, guaranteeing that there are no dependence violations to cause rollbacks. In this section, we present an experiment where we remove this fundamental assumption. We evaluate 4-way and 8-way machines that insert no delays into their executions, and upon detection of a dependence violation, must squash the thread and roll back the computation. This causes a performance degradation when the machine speculates on threads that try to read data before it is written and are unable to predict the value correctly.

In Figure 2.12, we show the results for 4-way and 8-way Base TLS machines. As expected, the performance of each Base machine is lower than that of the corresponding Optimal machine. Harmonic mean speedups range from 1.7 to 2.1 for a 4-way

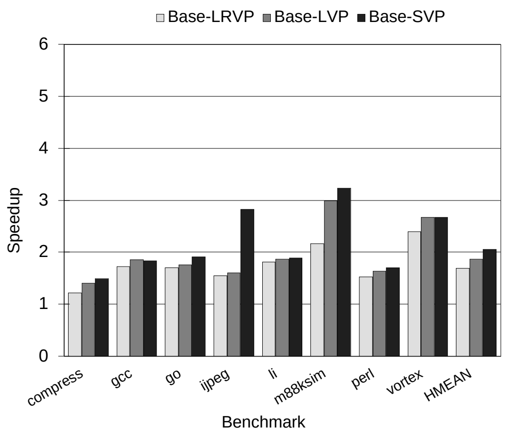


4-way

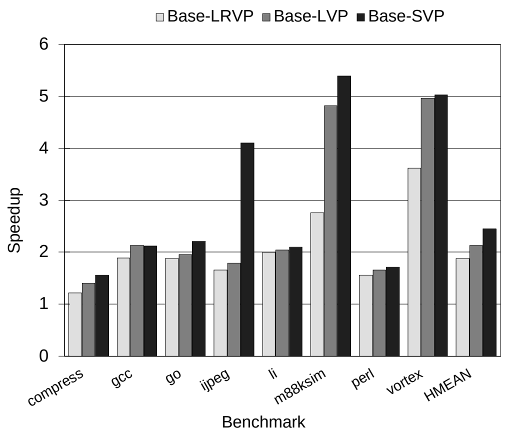


8-way

Figure 2.11: Loop and Procedure Speculation on 4-way and 8-way Optimal TLS Machines



4-way



8-way

Figure 2.12: Loop and Procedure Speculation on 4-way and 8-way TLS Machines with Rollback

base machine depending on the value prediction employed.

The finite-processor Base TLS machines are obviously still quite idealized. For example, the communication between threads incurs no overhead and thread creation and rollback is instantaneous. These investigations were largely intended to quickly evaluate the amount of inherent loop or procedural speculative parallelism in off-the-shelf integer code. In the following section, we describe both prior and further work that has been done on general-purpose thread-level speculation.

2.8 Related Work

The TLS model is based on the Multiscalar paradigm which was the first speculative thread architecture[14, 46]. The Multiscalar has a number of processing units organized in a ring-based topology for executing speculative threads. It has register and memory forwarding mechanisms and a mechanism for detecting memory data dependence violations called the address resolution buffer (ARB). The Multiscalar research group has evaluated the base processor and extensions to the processor that avoid unnecessary thread restarts using a number of integer applications, showing moderate speedups[33]. The Multiscalar group and other researchers have augmented the cache-coherency mechanisms of a single chip multiprocessor to support speculative threads [15, 33, 46]. The goal of these approaches is to achieve lower hardware overheads and more flexibility than the ARB approach originally proposed in the Multiscalar processor. To select tasks for the Multiscalar, a compiler pass examines the control-flow graph of the program and uses heuristics to partition the basic blocks into threads[57]. Speculative tasks are supposed to immediately follow the spawning task, so there is no nested task creation, but prediction of successor tasks is more difficult.

Steffan and Mowry evaluated the performance potential of a multiprocessor-based approach and showed reasonably good performance on a sampling of integer applications.[3] The reported performance was quite dependent upon rather idealistic

[3]Benchmarks included `compress`, `gcc`, `espresso`, and `sc` from SPECint92, `m88ksim`, `ijpeg`, `perl` and `go` from SPECint95, as well as `buk` from the NAS Parallel benchmark suite.

(and potentially unrealizable) dynamic instruction scheduling however; an average speedup of 1.5 dropped to 1.2 without such scheduling[51].[4] Later work by Steffan used a practical scheduling implementation, and evaluated potential benefits of value prediction, silent store elimination, hardware dependence synchronization, and modifications to the instruction issue logic to prioritize instructions on critical paths[50]. Results were mixed, with the authors concluding that their implementations of dynamic synchronization and issue logic prioritization were not helpful overall. Further work by Zhai et al. presented a complex compiler algorithm for aggressive scalar value communication between threads, allowing scheduling across potential control and data dependences[60]. Unfortunately, all these optimizations operate within a single-level loop speculation paradigm; thus parallel coverage, which averaged 51% via hand-selection in Steffan's work, is only 20% in Zhai's evaluation after poor regions are manually pruned out from the 35% of the program that was automatically targeted by the compiler. As a result, overall program speedup averages about 1.1. The lack parallelism coverage severely limits the utility of these optimizations.

The Trace processor is a concrete machine proposal that can exploit similar parallelism found in multiple flows of control[42]. In the Trace processor the program is dynamically broken into a sequence of instructions, each of which can be executed speculatively as a separate thread by an individual processing element (or PE). If an instruction violates a data dependence, only that instruction and the instructions dependent on it will be re-executed. The ability to selectively re-execute only those instructions that are affected mimics the ability of an oracle that can execute every instruction optimally whenever its operands are ready. Unfortunately, this ability comes with an implementation costs that increases greatly with trace length, as the processor must keep track of enough information to recover from all combinations of mispredictions. This constrains the size of each thread–the proposed maximum thread length is sixteen instructions. This limitation prevents the system from exploiting parallelism with a larger granularity. In comparison, the TLS machine model can realistically allow longer speculative threads than that of the Trace processor

[4]Code was scheduled using perfect runtime data dependence information in order to minimize cross-iteration dependence lengths.

because there is only one speculative state per thread. It can exploit parallelism between instructions that are farther apart, and can follow more independent flows of control because threads are explicit. However, our TLS thread restarts discard much more work than the Trace processor's selective recovery, so it is more important to minimize dependence violations.

To compare the performance of a Trace processor to an "equivalent" superscalar, the authors suggest evaluating machines of similar issue complexity[5] They evaluate a four-way issue superscalar with a sixteen instruction window versus a 4-, 8-, or 16-PE Trace processor with that same base superscalar as the processing element, arguing that since each element has identical and independent issue complexity, chip cycle time should be roughly similar. As the authors state, in this case "the only penalty for having more than one PE is the extra cycle to bypass values between PEs." In this scenario, they achieve speedups of 1.6, 1.8 and 2.0, for four, eight, and sixteen PEs, respectively, on `gcc`. However, this equivalence measure is a bit extreme. The Trace processor has many more functional units and instruction buffers distributed across the multiple PEs, and also utilizes a 64K-entry trace predictor in order to execute effectively. Also consider that the superscalar in this example has only two cache buses, while the trace processors use four, four, and eight buses for the four, eight, and sixteen PE configurations, respectively. If we compare those same Trace processors against a 4-way superscalar with a 64-instruction window, or an 8-way superscalar with a 32-instruction window, the performance gains are more modest: in both cases, roughly 1.1, 1.3, and 1.4 for Trace processors with four, eight, and sixteen PEs respectively.

The Dynamic Multithreading (DMT) processor[1] combines features of the Simultaneous Multithreading machine[55] and the Trace processor while also supporting procedural and some loop speculation. It executes threads in a tightly-coupled SMT-based environment. Selective recovery is performed from a trace buffer like the trace processor, but the machine does not compact normally executed code into traces. The speculative thread size is limited by the need to keep all of the thread's dynamic

[5]Superscalar issue complexity is generally equal to the product of instruction window size and instruction issue width.

instructions in the trace buffer; in their simulations, threads can be at most 500 instructions long. They chose loop and procedure continuations as their targets for speculative execution. Inner loop speculation is not supported in order to work with unmodified, preexisting binaries.[6] While our studies suggest that speculating only on single-level (mostly inner) loops in integer codes is insufficient, inner loops are still a valuable contributor to performance in certain programs. The DMT machine also does not perform explicit return value prediction. The only prediction employed is for registers, and it is limited to predicting that the initial register values for a child thread are the same as the parent thread's register values at the time the child is spawned. The DMT processor does have a fast forwarding mechanism for when register prediction fails, though there is no facility for memory load prediction or synchronization. Overall, the DMT achieves an average 1.5-times speedup on integer programs when executing with eight threads.

Hammond, Willey, and Olukotun presented a sample machine design implementing many of the ideas suggested in this paper and elsewhere, but the results reported initially were not encouraging[17]. In order to lessen the impact of TLS support on the base chip-multiprocessor hardware design, thread management is done largely in software by handlers that take significant time to operate. For example, starting a loop, committing a loop iteration, and ending a loop all take on the order of 70 to 80 instructions, which is fairly close to the average thread size that we observed. Additionally, while return value prediction is implemented for procedures, neither value prediction nor synchronization is implemented to help cope with the other data dependences across speculative threads. A subsequent paper by the group showed improved results when overheads were reduced and software was manually updated to introduce synchronization and better code scheduling[34]. The reduced overheads were achieved by eliminating support for procedural speculation in favor of more efficient support for loops, however. Overall, they achieved a 1.7x speedup a number of general purpose integer programs as a result of the lower-overhead handlers and

[6]Hardware only detects the end of a loop (when it sees a backwards branch instruction) as opposed to being alerted to the start of a loop. Thus DMT parallelizes the code coming after a loop body ("loop continuations") which would include any enclosing outer loop iterations.

software optimizations.[7] Performance increased to 1.8x when hardware checkpointing was added to reduce the number of instructions that needed to be re-executed when a thread is squashed.

Further work by Chen presented an effective system for speculation in Java programs using the Hydra CMP[4]. Thanks in part to the dynamic nature of Java, a profiler was developed to analyze loops at runtime to determine the amount and lengths of dependences between loop iterations, as well as the amount of speculative buffering needed. The runtime system then selects appropriate loops for speculation and generates optimized code. Chen did find that some of the integer programs benefited further from hand optimizations that could not be automated. Overall speedups reported ranged from 1.5x to 2.5x for integer programs, 2x to 3x on multimedia codes, and 3x to 4x for floating-point applications.[8]

Prabhu found that manual transformations by the programmer could significantly improve TLS performance on the Hydra CMP[40]. The transformations included complex value prediction, targeting non-loop parallelism via “speculative pipelining”, and various algorithm changes to expose parallelism. An average of eighty programmer hours was applied to three integer and four floating-point benchmarks, achieving an average 1.7x integer speedup and 2.1x floating-point speedup.

Other researchers have also examined the performance of various choices for thread boundaries. Codrescu compared the performance of loops, procedures, fixed-sized blocks of sixteen instructions (like the Trace processor), and finally a scheme devised by the author (called MEM-slicing) where memory instructions are picked as thread boundaries, subject to a minimum thread length of sixteen instructions[6]. The speedups for these four schemes were 1.7, 1.9, 2.5 and 3.4 respectively, executing on eight in-order CMP processors. The machine had some rather aggressive hardware, however, such as trace buffers for selective recovery of dependence violations, as well as a 128KB combined value/control predictor. Codrescu noted that procedures or loops alone did not seem to have sufficient coverage or load balance to provide good

[7] General-purpose integer programs evaluated were `compress` and `m88ksim` from SPECint95, `eqntott` from SPECint92, and the UNIX utilities `grep` and `wc`.

[8] Integer benchmarks included an assortment of Java programs from SPECjvm98, Java Grande, as well as Java applications publicly available on the Internet.

performance, though loops and procedures were not evaluated together. The MEM-slicing proposal yields great coverage and load balance; however, it must use a large predictor to "learn" the locations of thread boundaries, as well as provide needed value predictions. In addition, without selective thread recovery the performance with MEM-slicing drops from 3.4x to 1.8x.

Studies by Marcuello and González also evaluated thread-spawning schemes on a tightly-coupled multiprocessor with 4-way out-of-order cores[29]. Initial studies compared loop iterations, loop continuations, and procedure continuations (but not combinations thereof). They found that when the machine is limited to in-order thread creation (as opposed to the nested creation described in Section 2.5), the best performing scheme is loop iterations, with a 1.7x speedup on 16 processors. Note that this includes perfect value prediction of all register and memory dependences. When nested thread creation is allowed, a speedup of 5.7x is obtained, again with perfect prediction. They also investigate a realistic register value predictor, but one with an idealistic value misprediction penalty of a single-cycle over perfect synchronization, as well as perfect synchronization for memory. The best performance is again with loop iterations, an average 2.8x speedup.

In later work, Marcuello and González suggested a more general source of threads–simple sequences of basic blocks[30]. A profiler is used to determine which sequences are relatively control independent, sufficiently long, and do not have many external data dependences. With a 16KB register value predictor, the profiled basic block scheme performs 13% better than the combined loop and procedure speculation that they also implemented.

Finally, one of the most recent TLS machine proposals, the Implicitly-Multithreaded processor (IMT) has been proposed by Park[37]. This is an eight-wide SMT-based design, using the Multiscalar task selection compiler described previously, but with an execution preference for loop iterations. Squashed threads must fully re-execute (no selective recovery) and there is no value prediction. Dynamic dependence synchronization hardware is included in the design, however. Overall, Park found that an eight-thread TLS execution typically slowed down most integer benchmarks. He

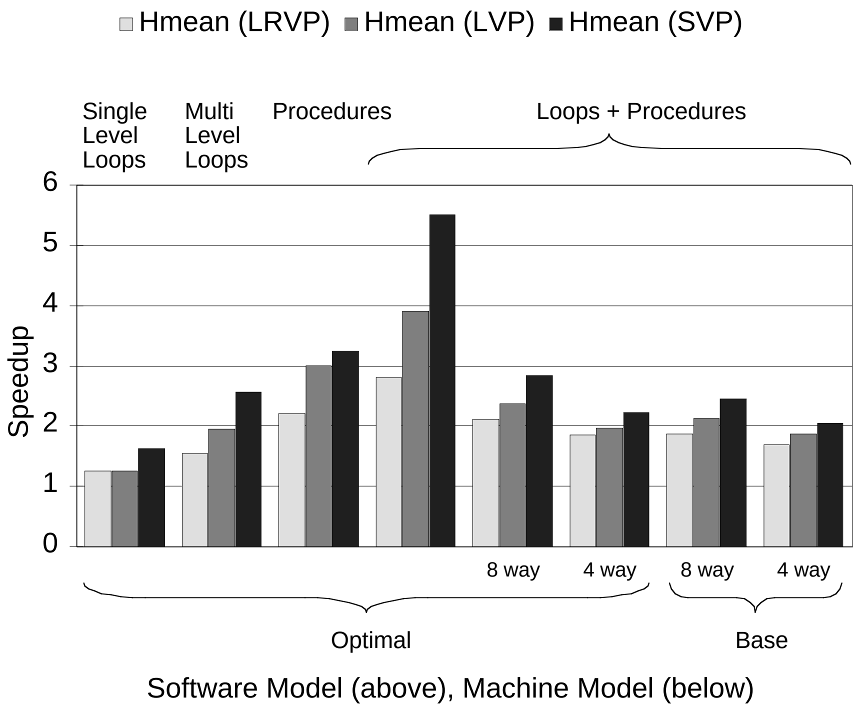


Figure 2.13: Summary of the Harmonic Mean Speedups

implemented additional optimizations, such as context multiplexing to allow adjacent threads to share a single execution context, a resource-aware thread fetch policy, and lower-overhead thread creation. With these additional optimizations, speedups averaging 1.2 for integer programs and 1.3 for floating-point applications are obtained.

General motivation for using multiple flows of control to increase sequential application performance was presented by Lam and Wilson[25]. While the value prediction we employ could result in performance beyond the dataflow limit observed in Lam and Wilson's experiments, our multiple flows of control (loop iterations and procedures) are much more restricted in nature.

2.9 Summary and Conclusions

We summarize our study of speculative parallelism with Figure 2.13, which shows the harmonic means of the performance results of the different experiments we performed. We started our exploration by assuming an optimal TLS machine with an infinite number of processors that completely avoids rollbacks. We experimented with different speculation schemes: speculating only one loop at a time, speculating at all loop levels, speculating at all procedural boundaries, and finally to speculating at both loop and procedure boundaries. We found that the last scheme delivers decent performance on the optimal TLS machine. Having found such a scheme, we then considered more realistic machine models. We first refined the parallelization scheme to reduce the number of threads created, and evaluated the performance of the programs on optimal TLS machines with 8 and 4 processors. Finally, we experimented with base machines that roll back speculative threads whenever dependence violations are detected.

The methodology used in this paper enabled us to analyze programs effectively and find potential sources of speculative parallelism. The relaxed machine model (Optimal) allowed us to quickly identify the fundamental limitations of loop level speculation. We were able to develop a variety of analysis and simulation tools that isolated parallelism coverage as an important factor in the lack of performance. This result led us to locate alternative sources of speculative parallelism, namely procedural speculation. The gradual refinement of the machine models from the optimal TLS machine, first with an infinite number of processors, then to a finite number of processors, and finally to machines with rollbacks increased our understanding of the different factors that affect performance.

Still, the results are for a relatively idealistic machine that does not account for the additional demands that speculation puts on memory bandwidth, for example, nor does the study address the impacts of thread operation overheads. Conversely, we do not address the potential performance improvements that could come from instruction and dependence scheduling, or beneficial thread selection or suppression criteria (in the finite processor models only). Given the complexities of thread selection, wide variety of potential hardware support options, and number of different software optimizations that could be applied, it is difficult if not impossible to provide a meaningful bound on the potential general-purpose TLS performance. However, we feel our study does show that single-level loop speculation will not provide significant speedups across a broad array of integer programs, and suggests that good TLS performance will be difficult to achieve, particularly without value prediction or significant changes to the code.

Devising an hardware-efficient scheme where thread-level speculation can be effective for a wide variety of general-purpose programs has proven to be a very challenging task for many researchers. As discussed in Section 2.8, recent TLS proposals have had significantly narrower focuses, such as scientific or multimedia codes, Java programs, or code that is made more suitable for TLS by manual programmer transformations. We have identified another class of programs that can significantly benefit from TLS–programs with verification or analysis code that can be parallelized, as well as programs where the fine-grained recovery available with TLS can be used directly by the programmer. By allowing a choice in how TLS is employed, one could use it to target pure performance via traditional thread-level speculation if the application was suitable, or TLS could be used to provide enhanced reliability if those qualities were deemed more important by the user.

Having explored some of the limits of general-purpose TLS, we now turn to look at these "reliability-based" programming paradigms that TLS can help support effectively.

Chapter 3

The Monitor-and-Recover Paradigm

In this chapter, we look at the two components of our proposed programming paradigm: monitoring program execution and recovery via fine-grain transactions.

3.1 Execution Monitoring

The concept of execution monitoring has been used in the past for a variety of purposes: architectural evaluations, off-line performance tuning, on-line optimization, program debugging tools, and dynamic program verification. Execution monitoring is so common that new programming languages as well as binary and bytecode rewrite tools have been developed to facilitate adding instrumentation to programs. Here, we advocate additional support for execution monitoring at the architectural level in order to make this paradigm more efficient.

3.1.1 Uses of Execution Monitoring

Practitioners often use profiling to understand the bottlenecks in their system and tune the performance of programs. Dynamic languages like Java have been shown to benefit greatly from dynamic optimizations. These optimizations rely on profile data gathered by instrumenting the program to identify the frequently executed code regions or frequently used parameters. Execution monitoring is also widely used in computer architecture research to determine how programs behave under different architectural models.

Apart from performance analysis and optimization, execution monitoring has also been used in a variety of ways to improve software reliability. Application-specific assertions are added to the code by programmers, and safe languages add checks to ensure, for example, that there are no null dereferences or out-of-bound array accesses during execution.

A number of execution monitoring tools have been developed to watch for common errors in programs. StackGuard is an example of a tool that stops buffer overrun attacks, which have been a major cause of recent software vulnerabilities[9]. StackGuard inserts checks into the code to ensure that the run-time stack has not been inappropriately tampered with. Execution monitoring tools that help find memory management errors are widely used. For example, Purify monitors memory accesses to detect memory leaks, duplicate frees and illegal access errors such as reading past the end of heap-allocated objects or from uninitialized memory[19].

While StackGuard and Purify are designed to find specific types of programming errors, DIDUCE is a more general tool that helps programmers find all kinds of application-specific bugs[18]. DIDUCE instruments Java bytecode to watch the data values accessed at various code points in the program. It creates a model of the correct behavior as the program runs, reports a potential error whenever it encounters a violation of the model extracted and then relaxes the model. By doing so, DIDUCE usually alerts the programmers of errors as they first appear, rather than waiting for the corrupted data to cause program exceptions or failure. This technique has been demonstrated to help programmers quickly diagnose bugs that result from algorithmic errors in handling corner cases, errors in expected inputs, and developers' misconceptions about the APIs of software components they use. DIDUCE also helps programmers find hidden errors, errors that can silently compromise the integrity of the result and cause even greater damage than obvious errors that crash a program.

Today, these execution monitoring tools are used mainly during the debugging phase of software development. In fact, there are also many advantages to including execution monitoring codes in production software. By detecting run time errors in the software, execution monitoring can prevent errors from silently compromising the integrity of the system and data. In addition, it may be possible for software to capture error information and send it back to the developers for analysis, alert the user to take precautions before a harder failure occurs, or even attempt to recover from errors. However, the large overhead introduced by execution monitoring has not only made execution monitoring infeasible in production software, but also limited its usage in software development.

3.1.2 Tools for Execution Monitoring

Extensive monitoring code is usually inserted automatically by a program. For example, a compiler might insert relevant safety checks at all appropriate code points. ATOM[48] and EEL[26] are examples of binary rewrite tools that help programmers instrument binaries. BCEL is an example of a Java bytecode rewrite tool[11]. These tools allow users to insert calls to user-supplied routines at specified code points, such as before or after basic blocks, procedure calls, or data accesses.

Support for execution monitoring is one of the motivations behind the development of aspect-oriented programming[22]. To enhance modularity of software, aspect-oriented programming allows programmers to create an *aspect* that groups together behavior that cuts across typical divisions of responsibility in a given program. For example, monitoring code to be inserted at the beginning and at the end of every function would be organized as an aspect. The aspect compiler would automatically weave these functions into the main computation. In this way, all the source code related to monitoring is written in one place, making it easy for the programmer to write and maintain the software.

3.1.3 Speeding up Monitoring Functions

Monitoring functions represent pure overhead if the program is correct. In the rare event that an error is caught, it is important that the monitoring function communicate the failed check to the original program. This is usually achieved via a raised exception, an error return code, or by just terminating the main program. Production software programs, especially server programs where availability is paramount, often check for unexpected inputs or conditions and then needs to recover without terminating the program.

To enhance the performance of these checks, we propose that the software, typically an instrumentation tool, convey to the architecture which functions are monitoring functions. The programmer writes code following the simple semantics that the monitoring functions and the normal computation are executing sequentially. In fact, on a TLS machine, each call of a monitoring function may spawn a new thread; the original thread executes the monitoring function and the new thread executes the code after the call *speculatively* in parallel. The TLS architecture ensures that the sequential semantics are preserved. The monitoring function only sees the program state at the beginning of the call and the subsequent changes that it itself makes; the state of the speculative computation is buffered and not observed. If a data or control hazard is detected, that is if the speculative thread uses data that is later produced by the monitoring function call, or if the speculative thread was not supposed to execute at all, it is rolled back and not allowed to commit.

Typically the monitoring function reads the state of the main computation and operates on its own data structures. It may communicate with the main function by returning an error code that indicates success or failure. These kinds of results can be very accurately predicted because the program is expected to be correct. Thus, the speculative thread should not typically need to be rolled back.

It is possible that multiple monitoring functions might be invoked in rapid succession. If they are pure functions that simply return a predictable result, then parallel execution is possible. However, some of these monitoring codes may update internal data structures that are then read by later monitor calls. The TLS hardware will guarantee the sequential semantics even if data hazards occur between the calls. We expect these dependences to be rather sparse, especially if the monitoring routines are spaced out with the main computation.

3.1.4 Summary

In our proposed model, programmers can create monitoring functions simply by assuming that these functions will be called and executed sequentially. A code instrumentation tool inserts calls to the monitoring functions and marks these routines as speculative. The machine then exploits the fact that monitoring functions are mostly independent from the main computation and uses thread-level speculation to overlap their execution. When the program is correct, the monitoring functions run faster because they are overlapped with the main computation; if the monitoring functions find an error, the hardware allows the error be processed as if the program had executed sequentially.

3.2 Fine-Grain Transactional Programming

Once an error is detected, it is important for many programs to recover from the error and not just terminate the computation. For example, the software may be providing services that cannot be interrupted, or the program may have some volatile state that is difficult or impossible to reconstruct. Even for software that can easily be restarted, the overhead involved in restarting a process may represent a denial-of-service vulnerability.

Clearly, it is preferable that software check its inputs and not make any erroneous updates to its state. However, despite all precautions, programmers still make mistakes. It is often easier to introduce end-to-end checks that examine the integrity of a computational state, but that means the program must now recover from a compromised state.

Transactions are a powerful abstraction that can be used to address this difficulty. We envision that programs in the future be built as a composition of transactions. Each transaction is a unit of computation whose side effects, which include all changes to registers and memory, can be committed or discarded as a unit. If checking code detects a problem, the transaction is aborted and the pre-transaction state restored. This provides the programmer with an extremely simple approach to error recovery.

Transactions have been extensively used in database applications; these are rather heavy-weight transactions that typically involve writing to permanent storage. The concept of transactions has also been applied at the operating system level, where changes of virtual memories and even system calls can be "undone"[20, 28, 43].

Transactions used for the sake of error containment and recovery can be very fine-grained, on the order of tens and hundreds of instructions. As such, we must reduce the overheads involved in implementing these transactions. By keeping transaction state in the speculative buffers of a TLS machine, we can provide extremely low-overhead transaction support.

3.2.1 Examples of Transactional Programming

Let us motivate the use of end-to-end checks with the single most common source of security vulnerabilities: buffer overruns. Despite the large number of man-years devoted to solving this problem, additional security holes based on overruns continue to be discovered. Many vulnerable programs are written in C; because C is unsafe, these programs can often be manipulated with carefully crafted inputs to write past the intended buffers. Manual inspection has proven to be inadequate, because vulnerabilities have been found even in codes that have already been audited for security holes[9]. It is in general impossible to determine if a program may overrun its buffer statically. And dynamic techniques that insert array bound checks into C are too slow to use in practice[21].

The need for end-to-end checks has prompted the development of tools like StackGuard[9], which verifies that portions of the C run-time stack have not been improperly overwritten. Unfortunately, by the time a problem is detected, damage has already been done.

Using transactional programming, the programmer only has to identify routines that are potentially problematic, as opposed to finding the exact errors. For example, many servers have been found with security vulnerabilities in input parsing functions. One approach is to make the input handling routine a transaction and monitor its computation. If any errors are found, the entire transaction is aborted and that particular input can be skipped. The input handling routines can create necessary data structures as normal, knowing that all side effects will be wiped out cleanly, without any corruption of data structures, if the transaction is eventually aborted.

Transactional programming also makes it convenient to write programs that may need to "undo" portions of the computation. An example can be found in network protocol processing code. In the implementation of HTTP proxy caches, for example, there can be a considerable amount of effort dedicated to ensuring that HTTP inputs are reasonable. Unreasonable requests can result in crashes or the allocation of excessive or unauthorized resources. When an unreasonable request is detected, perhaps midway through processing, any partially constructed data structures that have been created must be cleaned up. By treating the processing of the request as a separate thread with its own private memory state, the model for such validation is simplified considerably. Input validation is considered a separate transaction that can be aborted, and when aborted the resources are reclaimed. This compares favorably to the conventional model of error handling "cleanup" code that is infrequently exercised or tested, and can contain subtle bugs or resource leaks.

To emphasize this point, we found a concrete example of problematic "cleanup" code in the first open-source reference release of the User Direct Access Programming Library (or uDAPL) by the DAT Collaborative[8]. The library is designed to provide a common API for programmers to easily use a variety of underlying networking protocols that each supports remote direct memory access or RDMA. The bug is contained in the `dapls_evd_alloc()` function that allocates an event descriptor (EVD); a distillation of that code is shown in Figure 3.1. The EVD is a structure that queues networking events for a consumer, so it has a number of private fields, as well as two circular buffers to hold free events and pending events. There are numerous allocations and initializations made in this allocation routine, and if an error is detected at certain points along the way, the routine jumps to a "`bail`" label. Unfortunately, that code simply calls `dapls_evd_dealloc()`, a routine that expects a fully initialized

```
dapls_evd_alloc(...) {
    /* Allocate EVD */
    evd_ptr = dapl_os_alloc(...);
    if (!evd_ptr) goto bail;

    /* Initialize */
    evd_ptr->header.provider = ...
    ...

    /* Allocate EVENTs */
    evd_ptr->events  = dapl_os_alloc(...);
    if (!evd_ptr->events) goto bail;

    /* allocate free event queue */
    dat_status = dapls_rbuf_alloc(&evd_ptr->free_event_queue, ...);
    if (dat_status != DAT_SUCCESS) goto bail;

    /* allocate pending event queue */
    dat_status = dapls_rbuf_alloc(&evd_ptr->pending_event_queue, ...);
    if (dat_status != DAT_SUCCESS) goto bail;

    /* add events to free event queue */
    for (i = 0; i < evd_ptr->qlen; i++)
      dapls_rbuf_add(&evd_ptr->free_event_queue, ...);
    return(evd_ptr);

bail:
    if (evd_ptr) dapls_evd_dealloc(evd_ptr);
    return(NULL);
}

dapls_evd_dealloc(...) {
    ...
    // FIXME-there is a problem here if rbufs failed during alloc
    dapls_rbuf_destroy (&evd_ptr->free_event_queue);
    dapls_rbuf_destroy (&evd_ptr->pending_event_queue);
    ...
}
```

Figure 3.1: Cleanup-code Error Example

```
TRY {
  ... the original code ...
  if (error-detected())
     ABORT;
} CATCH {
  return an error code;
}
return ok;
```

Figure 3.2: Transaction Syntax

and consistent EVD structure to deallocate. The programmer seems to have noted this problem in the comments, but perhaps for reasons of expediency, or a belief that the error-handling code is unlikely to be executed in any case, this corner-case error has been left in. It should be noted that this incorrect cleanup code was fixed in later code revisions to the library, but certainly there are similar situations that remain uncorrected in code that is widely distributed and used. We feel this is an excellent example of where a simple, fine-grained transaction could be used to effectively clean up data structures in situations where programmers might be tempted to defer writing complicated manual recovery with a "FIXME" note placeholder instead.

3.2.2 Programming Constructs

To make the concept of transactional programming more concrete, we propose that the programmer writes transactions using the syntax shown in Figure 3.2.

Although the `try..catch` syntax we have adopted above resembles that of traditionally exception handling constructs, the semantics is quite different. In conventional exception handling, the stack is unwound and the control flow returns to the `catch` block when an exception is signaled, but side effects made to all other data structures are not removed. The exception handler must be careful in restoring all the data structures to a clean state. In our case, the hardware automatically buffers up all the memory writes and discards them when an `abort` is issued, guaranteeing

```
    TRY <L1>;
    ... the original code ...
    if (error-detected())
       ABORT;
    COMMIT;
    return ok;
L1: return an error code;
```

Figure 3.3: Transaction Machine-level Pseudocode

that the memory is restored to its state just before the `try` statement. It is this ability that allows the program to recover from attacks that overwrite data structures beyond those expected.

To support this operation, we propose three machine instructions:

TRY ⟨ADDR⟩. This instruction indicates the start of a transaction. In TLS terminology, the thread at the point becomes speculative and all the side effects are buffered. If the transaction is aborted, the speculative state is discarded and the flow of control branches to the given ⟨ADDR⟩ address.

ABORT. This instruction indicates that the transaction is to be aborted. The hardware discards the speculative state, and the program counter is set to the address specified by the last TRY instruction.

COMMIT. This instruction indicates that the transaction is to be committed. The machine commits the speculative state, and the execution continues as usual.

Compiling the high-level construct discussed above to machine instructions yields the pseudocode shown in Figure 3.3.

3.2.3 Discussion

Different granularities of transactions require different implementation techniques. Architectural support is necessary to minimize the overhead of transactions of the finest granularity. Conversely, hardware support is unsuitable for implementing coarser-grain transactions. Statements nested under these architectural-supported transactions cannot execute any system calls unless operating system support is provided. If the code is run in a multi-threaded environment, threads can read shared state, but if a thread wishes to write into any shared state, then locks must be held until the speculative state is committed.

Ideally, we want to provide the programmer a system that automatically adapts and chooses a combination of architectural and operating system techniques to implement transactions of any granularity. One possibility is for the machine to raise an exception when the state of the transaction overflows the available hardware, and have the system switch to an operating-system based technique to support the coarse granularity.

Chapter 4

Machine Architecture

There have been quite a number of architectures proposed for thread-level speculation. They all provide three basic functions:

- multiple simultaneous threads of control,
- buffering of speculative state so that side effects can be discarded and false dependences can be eliminated, and
- detection of true data dependence violations between threads and the ability to discard or commit the speculative state to maintain sequential semantics.

The various architectural proposals vary widely in the degree of coupling between the threads in the system. At one extreme, speculative thread parallelism has been proposed for large-scale shared memory multiprocessors[5, 13, 49]. More common proposals, such as those based upon chip multiprocessors, are fairly loosely coupled and typically provide inter-thread communication through the second-level cache[7, 17, 51]. Some propose relatively novel architectures, such as the ring of processing elements found in the Multiscalar[46]. The most tightly-coupled implementations include the DMT machine[1], which adds speculative thread support to a simultaneous multithreaded (SMT)[55] version of a traditional superscalar processor. In the DMT, threads communicate through forwarded registers and an expanded load-store queue.

Our objective is to support the speculative execution of both execution monitoring code as well as fine-grained transactions. Both of these scenarios generate relatively fine-grained threads, which are likely to perform poorly on a loosely-coupled TLS machine where thread operations (like creation and commit) would have higher overheads. Thus, our proposed architecture is based on a tightly-coupled SMT machine, and is quite similar to the DMT machine proposal.

TLS machine proposals also vary in the way they divide the computation up into speculative threads. Parallelizing loop iterations is the most common technique[17, 51, 54]. Some researchers have proposed speculating on procedure continuations, that is, the codes that immediately follow a procedure call[1, 36]. Other techniques include using a compiler analysis to divide the static program into threads[57], using fixed-interval chunks of instruction traces[42], or applying a predetermined heuristic to dynamically partition the program[6].

While our architecture can support all the above options, the work presented here focuses on support for execution monitoring and transactions. The TLS scheme we propose for execution monitoring is a special case of speculative procedure continuations–the main difference is that the software is responsible for selecting which procedure continuations to execute speculatively. We can exploit higher-level program knowledge to restrict TLS to a set of functions, in this case monitoring calls, where speculation is more likely to succeed. Support for transactions, as discussed in Section 3.2, requires the addition of a few instructions so that at run time the software can control whether transactions are committed or aborted. These relatively small additions give the software the necessary hooks to make better use of the basic TLS hardware.

We now give an overview of the architecture, followed by the details of how the speculative threads are controlled, and the addition of value prediction to the scheme. Next, we describe an optimization to the calling convention that makes procedural speculation more efficient. Finally, we discuss how fine-grained transactions are supported.

4.1 Overview

Our architecture is an extension of a simultaneous multi-threaded machine (SMT)[55], which in turn is based on a superscalar architecture. Figure 4.1 shows a block diagram of our machine. The main features of our base superscalar machine include a standard 5-stage pipeline:

1. Fetch instructions from the instruction cache into the instruction queues.
2. Decode the instruction and rename registers.
3. Issue ready instructions and execute.
4. Write back register results.
5. Commit instructions in the original program order.

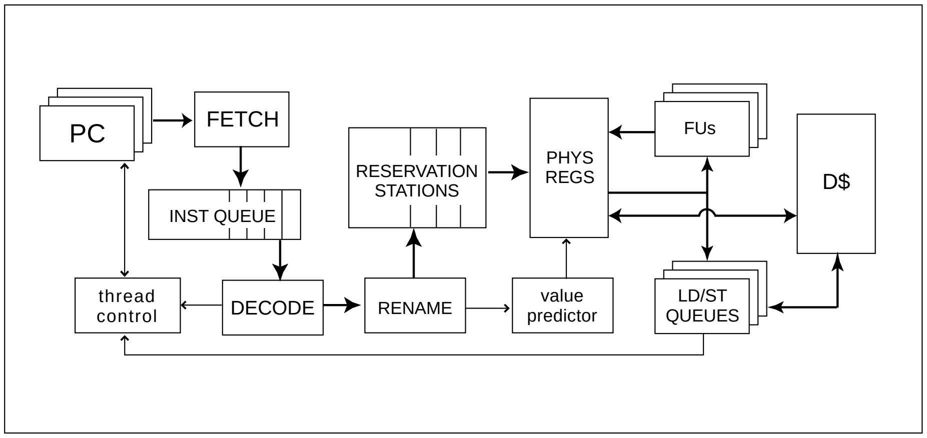


Figure 4.1: Machine Architecture

Simultaneous multi-threading extends the superscalar processor by allowing the machine to execute instructions from multiple threads on the same set of functional units, using primarily the same hardware scheduling mechanism as the superscalar. To store the contexts associated with the multiple threads, architectural registers (including program counters), the register renaming table, and the branch predictor's return-address stack are all duplicated for each thread. The fetch unit is modified to arbitrate instruction fetch between the threads, while branch misprediction recovery as well as instruction retirement are modified to operate on a per-thread basis. Having

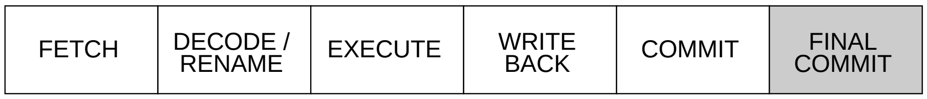


Figure 4.2: Machine Pipeline

a common pool of physical registers and functional units, SMT allows the threads to share the hardware resources in a more fluid manner.

The following is a high-level description of the features added to support thread-level speculation. More details are provided in Sections 4.2 through 4.6.

Thread control logic. This unit initiates threads, detects when threads stop, and either validates the speculation or restarts the speculative thread. It keeps track of the priorities of speculative threads, with higher priorities being given to those that would have occurred earlier had the code been executed sequentially. Our base proposal allows for an arbitrary ordering between threads, though we discuss the performance impact of a simpler implementation in Section 5.2.4.

Speculative memory state and data hazard detection. The load-store queue (LSQ) is replicated per thread to buffer the speculative memory state. This fine-grain support allows stores by a thread be observed by a later thread within two clocks. The queues are augmented with address comparators and snooping logic to detect memory data dependence violations between the threads. This expanded snooping logic will likely require a longer latency than traditional LSQ operation, a consequence we evaluate in Section 5.2.7. The original five-stage pipeline is augmented with a final commit stage, shown in Figure 4.2, that issues the buffered speculative stores buffered to memory when the thread commits.

Additional architectural register sets per thread. We add a set of "input" registers per thread, representing the starting state of each thread. We refer to the per-thread architectural registers that are updated during thread execution (known as the "retirement" register file in a superscalar) as the "output" registers. We presume a register file organization very similar to the DMT's, where single-cycle copies of the registers from thread to thread are possible. Note that in TLS, the output registers are no longer known to be correct and final, as the speculative thread could potentially be restarted. A single "final" register set is updated when threads commit and are no longer speculative; no recovery is possible once final registers are updated.

Modified arbitration policies. As in typical SMT implementations[56], our machine can fetch from a maximum of 2 threads per cycle. With TLS, however, all threads are not created equal. Only the sequential thread is guaranteed to make forward progress, while all the other threads are speculative and may be rolled back. Thus our policy is to fetch from the two highest priority threads whose fetches are not blocked. We also prioritize the arbitration for shared resources each cycle to favor the higher-priority threads.

A value predictor. To improve the success of speculation, a value predictor is used to predict the results of procedure return values that would otherwise cause thread squashes.

Buffered transaction state in the cache. To support coarser-grain transactions, each cache line is augmented with a single bit to mark whether the data is speculative and must be held in the cache, or is nonspeculative and free to be stored further out in the memory hierarchy.

4.2 Procedural Speculation

Procedural speculation is supported with a design very similar to that in the DMT architecture. The design supports nested procedural speculation as well as exception handling.

In procedural speculation, it is the code following the procedure that is being executed speculatively, since the called function executes first according to the original sequential semantics. A thread's current context is used to execute the called function, and a new thread context speculatively executes the computation after the call.

A speculative thread may encounter three kinds of potential hazards: memory data hazards, register data hazards, and control hazards. Memory data hazards are detected as a thread runs, whereas register data hazards are checked only when a thread is ready to commit. A thread is rolled back whenever a data hazard is detected; when that happens, all the lower-priority threads that have read from other speculative threads must also be rolled back. A control hazard occurs if the procedure call preceding a speculative thread generates an exception and never returns to the call site. In that case the speculative thread stays in the system until it is evicted from the machine.

The lowest-priority thread is evicted whenever a higher-priority thread wishes to spawn but there are no thread resources available. This ensures that resources are applied to the most important and least speculative threads. Useless threads wind up being evicted from the machine, but poor load balance or control mispredictions can cause useful threads to be evicted as well.

A thread executes until it reaches the end of the program, or it *meets* the speculative thread started on its behalf, or it gets evicted from the system. Its state is committed when it becomes the *head thread*, that is, the thread with the highest priority in the system.

To facilitate the fast creation and validation of threads, we propose a register file design, shown in Figure 4.3, that is essentially the same as the one proposed for the DMT processor[1]. All copies of a particular architectural register are kept close together to facilitate fast copying during thread creation, and fast validation during thread meet.

We describe each of the thread operations in more detail below.

Creating a new thread. To create a new thread, an empty thread context is selected if available. If all contexts are full, a new context is made available by discarding the lowest-priority thread. This requires invalidating all its state in the reservation stations, execution units, load-store queues, rename tables and fetch structures.

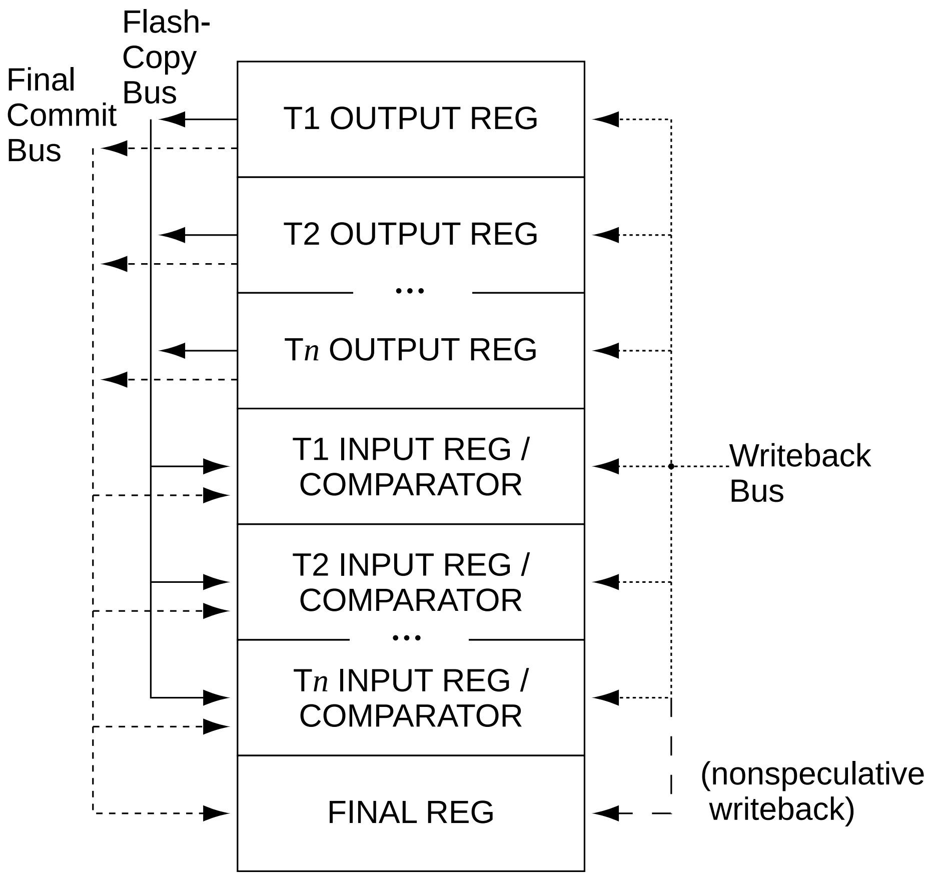

Final Commit Bus
Flash-Copy Bus
T1 OUTPUT REG
T2 OUTPUT REG
Tn OUTPUT REG
T1 INPUT REG / COMPARATOR
T2 INPUT REG / COMPARATOR
Tn INPUT REG / COMPARATOR
FINAL REG
Writeback Bus
(nonspeculative writeback)

Figure 4.3: Register File Organization

The "output" register state of the current thread is flash-copied in a single cycle into the "input" register set of the new thread. If the output register has a tag indicating that an in-flight instruction has an update of that register pending, the tag representing that write will be copied into the input register in lieu of a register value. When the in-flight instruction completes, the input register will be set with the register value arriving on the writeback bus. In the fetch unit, the new thread's program counter is set to the instruction after the call (the return address) and it begins fetching from there. In addition, the return address stack of the current thread is copied to the new thread. Our system allows only one new thread to be created per cycle, again favoring higher-priority threads, and there is no additional cost to starting a thread, apart from the pipeline startup delays associated with redirecting fetch.

Stopping a thread. During the execution of a thread, the thread control logic watches the fetch addresses of all the threads. If any thread T_i attempts to fetch from the same address as the start PC of the next speculative thread T_{i+1} in the priority list, instruction fetch for thread T_i is blocked and T_i waits to become highest priority. We refer to this as a thread *meet*. Note that a stopped thread may be restarted if data hazards arise before the thread can be committed.

Committing a thread. When a speculative thread becomes the head thread T_1, the thread is no longer speculative and the final commit stage becomes active. In that stage, any buffered speculative stores are issued to memory, up to the commit width of the machine and subject to the availability of store ports. T_1's speculatively written output registers are also committed to the final register file, again subject to the commit width.

Note that T_1 can continue to fetch and execute while the thread is committing, as long as fetch has not been stopped by a meet with the next thread T_2. If T_1 finishes committing and has not met with T_2, it continues executing as the sequential thread and thus does not buffer any speculative state. The store addresses produced by T_1 will still be snooped by lower-priority threads to detect potential hazards.

Memory data hazards. A lower-priority thread must watch the writes of higher-priority threads to ensure that there are no data hazards. These hazards are detected in the load-store queues. All loads and stores of a speculative thread are buffered in program order in a single queue, and these queue entries are not released until the speculative thread commits. Store entries contain both the address and the data for the store. Load entries contain the address as well as a bit indicating whether the load was satisfied by an earlier store entry in the same queue.

When the address of a store in the LSQ resolves, it is checked against all loads with resolved addresses in the queues of lower-priority threads. The following two conditions are checked:

1. Do the addresses match or overlap?
2. Was the load satisfied within its own thread by a earlier store in the same queue?

If condition 1 is true and condition 2 is false, then there is in fact a data hazard, and the lower-priority thread (with the load) is restarted (using its saved "input" register state) so that the memory dependence will be observed. In addition, any other lower-priority threads that have read any speculative data from other threads are restarted as well, as they may have read corrupt data from the thread we are restarting. Note that we do not restart when both conditions 1 and 2 are met. This policy, together with the prioritized load access described in Section 4.3, effectively implements "memory renaming" between the threads and prevents unnecessary restarts due to false dependences.

Register data hazards. Register validation takes place as the head thread T_1 goes through its commit process, writing back modified output register values to the final register file. We must ensure that any input register of T_2 (the second-highest priority thread that immediately follows T_1) matches the final value coming from T_1 if that value was in fact used by T_2. This is accomplished by having T_2's input registers watch the register values that T_1 commits to the final registers and then verifying that they are the same. If no output register value is committed by T_1 for a particular register, then T_2's input register must be compared with the final register value instead.

When all of T_1's stores have been issued to memory and all modified output registers have been committed, we know that any memory dependence violations would have been detected and handled. In addition, any incorrect input register values used by T_2 would be detected by this point. If one or more of T_2's input registers indicate that T_2 used an incorrect input value, T_2 is restarted with correct values from the final register file. If the input registers all used correct values, T_1's context is freed, T_2 becomes the head thread, and then begins its own commit process.

Performance optimizations. If a thread is restarted due to a data hazard, any threads it has created in the past are now unneeded *orphan* threads, as the restarted thread will create any desired child threads again as it re-executes. Because of nested thread creation, however, there may be valid threads, representing speculation at outer procedure levels, that are lower in the priority list than these orphans. Because of that lower priority, they would be targeted for eviction sooner than the orphaned threads. To prevent this, the control logic for each thread watches the status of its parent thread. If a parent is restarted or deleted, the newly orphaned thread will stop execution as soon as access to the thread priority list is available. Transitively, any children of the orphaned threads will be deleted in later cycles.

4.3 The Load-store Queues

As described in the previous section, the replicated load-store queues are modified to watch store addresses of previous threads to detect data dependence violations and signal the need for a restart. In addition, we must modify the lookup procedure when a load instruction is issued. Figure 4.4 shows how load lookups are performed in an example with four threads. The load address is sent to all load-store queues, which determine whether they contain a store entry with a matching address. If so, a hit is signaled, and the data from the latest matching store is forwarded. We then must use the ordering of the threads in the sequential program execution to select the most recent store value from among all lower-priority threads with hits. This thread ordering is provided by the thread control logic, and combined with the hit information selects the correct queue. If the load is satisfied by a store in its own queue, we set a bit in the load entry noting that it is intrathread, so any subsequent stores in higher-priority threads do not cause an unnecessary restart.

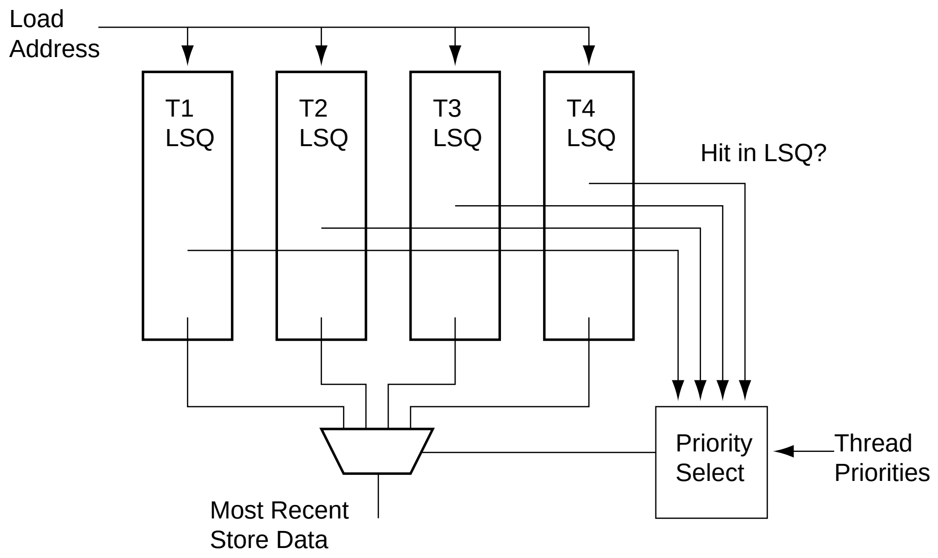


Figure 4.4: Load Lookup Operation with Four Load-store Queues

There is also the possibility that the most recent address-matching store may not yet have its store data available. In this case, we put the load to "sleep" instead of issuing it, as executing with any other data would most likely result in a data dependence violation. In addition to suppressing the load for the time being, we mark the matching store to indicate that it should issue a "wake" signal when its data value is ready. Once such a wake signal is received by the sleeping load, it attempts to reissue as usual.

4.4 Value Prediction

Whenever we create a new thread, it receives an initial copy of the spawning thread's register state at creation time as its predicted starting state. We refer to this primitive form of value prediction as "spawn prediction". For procedural speculation, this prediction tends to work very well, with the exception of the return value register. The speculative threads would have restored caller-saved registers before using them, and callee-saved registers would be restored by the callee before it returns if it changes

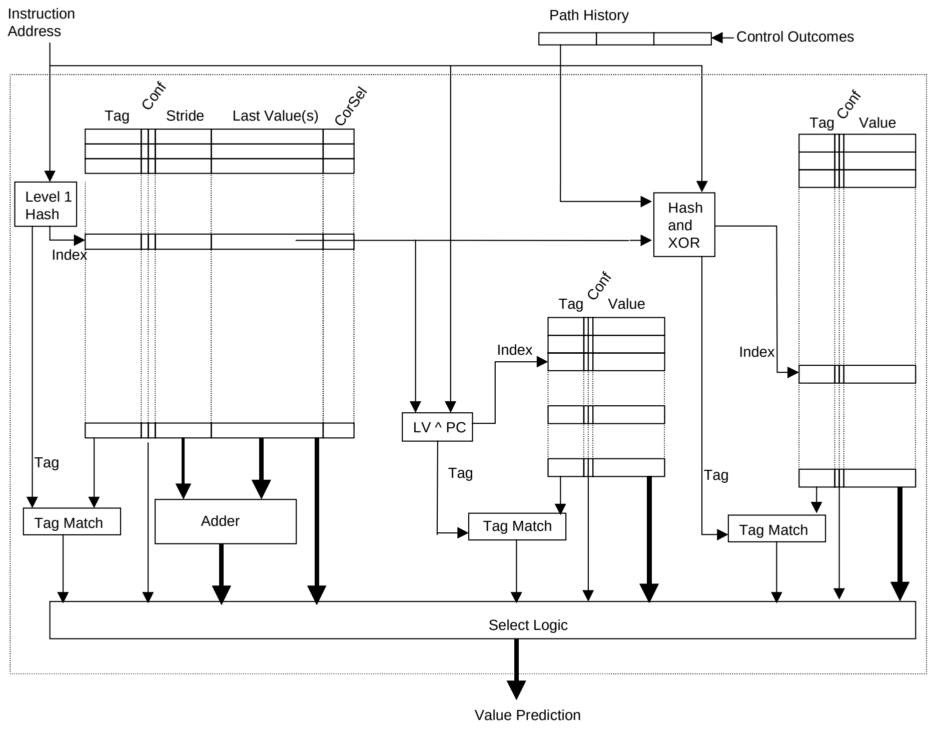


Figure 4.5: ATLAS Hybrid Local/Global Predictor

their values. Thus, no data hazards are expected to be observed for any of the register values except for the return value. In addition, data hazards may exist through updates to memory (e.g. pointer arguments or global variables).

To improve the success of speculation, we have implemented value prediction for procedure return values only. The predictor can also be used to predict values for load instructions that frequently cause thread restarts, but for the monitoring codes we examined it did not prove to be necessary or helpful. Most instrumentations did not need load value prediction as they experienced very few memory violations, while the instrumentations that did have significant memory violations did not have enough value locality to make load value prediction effective. Our description of the value predictor thus focuses only on its use for return value prediction.

Entries in the value predictor are indexed by the thread-starting address. When a thread is initiated, the predictor is checked for an entry. If there is a hit, the predicted value will be installed in the return value register for procedure threads.

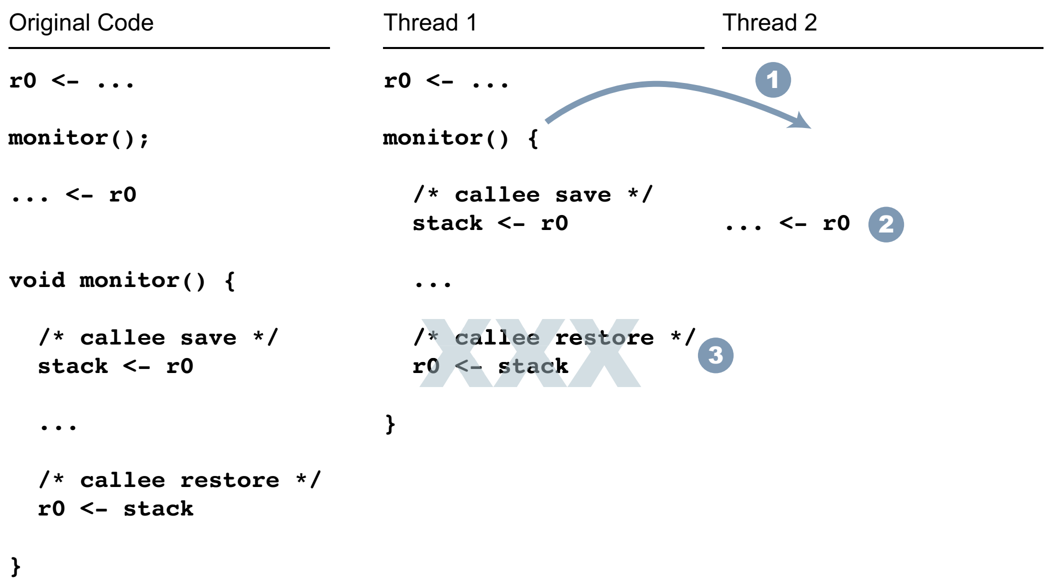
The value predictor used is the Atlas Hybrid Local/Global predictor that was described and implemented by Codrescu[7]. We show a diagram of it in Figure 4.5. It has a first-level table containing local value history that is used for last value and stride value predictions; this table is also used to index into two second-level context predictor tables as shown. One of the second-level tables is indexed in a shallow fashion: using only the instruction address as well as the last predicted value. The other second-level table combines a deeper value history together with recent branch outcomes and the instruction address to form its index. Each predictor entry in the three tables has a saturating confidence counter, so the predictor with the greatest confidence at the time is chosen to provide the predicted value.

Value predictions are verified and updated as follows. When head thread T_1 meets with the next thread T_2, and a return value prediction had been made for T_2, the standard register validation process (described in Section 4.2) is used to check the values and restart T_2 if necessary. In addition, the value predictor is accessed and updated with the correct return value from T_1. The saturating confidence counters for the different predictor entries (in the first-level and two second-level tables) are also updated in this process.

Entries in the value predictor are introduced on demand whenever a thread is restarted due to an incorrect return value register, that is, after simple "spawn" value prediction has failed.

4.5 Calling Convention Optimization

When examining the instrumentation code that we are targeting for TLS execution, we saw that many of these routines spend significant time saving and restoring the register state of the main computation. In the case of binary instrumentation, where the instrumentation calls are directly inserted into the binary, the instrumentation must preserve any temporary or argument registers that it may overwrite. (Under normal calling conventions, it would be the responsibility of the caller to save these registers.) Also, if the instrumentation routine is not a leaf procedure, the register usage of its callees may not be known, so again the instrumentation must be


Original Code

```
r0 <- ...

monitor();

... <- r0


void monitor() {

  /* callee save */
  stack <- r0

  ...

  /* callee restore */
  r0 <- stack

}
```

Thread 1

```
r0 <- ...

monitor() {

  /* callee save */
  stack <- r0

  ...

  /* callee restore */
  r0 <- stack

}
```

Thread 2

```
... <- r0
```

Figure 4.6: Calling Convention Optimization

conservative and preserve them.

When executing the instrumentation code using TLS, however, we already have a copy of the register state at procedure call boundaries—in the input registers of the next thread. Thus it is possible to eliminate the register restore operations if we instead rely on the input registers to preserve the values. We show how this scheme works in Figure 4.6, quite similar to the code in Figure 2.3, except here the procedure being called is a monitoring function.

When the monitoring function is called, we fork a new thread (step 1 in the figure) and copy the current register state into the input registers of the new thread. As the new thread executes, it will the use input register values as needed (step 2). Meanwhile, when the end of the monitoring function is reached, the register restore operations we would normally execute are not necessary as long as we can guarantee that the input register state of the next thread (thread 2) is correct. If we are confident of that fact, we can simply suppress the restore operations (step 3), thread 1 then retires, and thread 2 continues executing.

To take advantage of this opportunity, we have implemented an optimization that dynamically suppresses the execution of these register restore operations when possible. We use a simple static analysis to identify the register restore instructions in the binary. These instructions are identified out-of-band to our simulator, but in a real-world implementation the instructions would need to be marked. This could be accomplished either with an annotation on the instruction itself, such as an additional bit or special opcode, or the entire restore sequence could be demarked by placing special instructions at the beginning and end of each sequence. We only mark the restore operations inside of procedures that may execute with speculative continuations; thus only restore operations inside monitoring functions are marked for potential elimination.

Before performing this optimization, we need to be sure that the next thread indeed holds correct register value. Typically it should, but there are a number reasons why it may not, namely:

1. *A thread fork operation may have been suppressed.*[1] If so, we obviously do not want to eliminate restore operations, as no corresponding thread was created with the needed register values.

2. *Lower-priority threads may have been evicted from the machine after creation.* In the process, the input register state would also be discarded. Thus any operations restoring those values should in fact execute instead of being eliminated.

3. *A* `longjmp` *operation may have executed.* The location of the corresponding `setjmp` indicates which thread, if it exists, now has the appropriate input register values. This is too complicated to accurately track, so restore operations should not be eliminated.

4. *A thread may have been created under branch misprediction, i.e. during wrong-path execution.* A thread created during wrong-path execution does not have a valid input register state; thus no restore operations can be eliminated if they would rely on correct values being available in the input registers of that thread.

[1] In our scheme only a single new thread can be created (machine-wide) each cycle, so additional requests to fork a thread that same cycle would be suppressed.

We handle most of these cases by maintaining a boolean value, *OptimizeOK*, in each thread context indicating whether it is safe to eliminate any marked register restore operations are encountered. This boolean is initialized to `true` in the parent thread when a new speculative thread is spawned. To prevent incorrect execution in the cases outlined above, we set *OptimizeOK* for thread T_i to `false` when any of the following occur:

1. T_i attempts to fork a new thread but the operation is suppressed

2. T_i becomes the lowest-priority ("last") thread in a full machine where all available thread contexts are occupied

3. T_i executes a `longjmp` operation

4. Upon branch misprediction recovery, T_i detects that it had forked a thread during wrong-path execution

Thus, after any of these events occur, T_i will not be allowed to eliminate restore operations. However, the situation changes if T_i subsequently forks a new child thread T_{i+1}. Now we want to allow T_i to optimize restore operations with respect to the newly forked procedure continuation, but any conditions that were *previously* preventing T_i from optimizing should now restrict optimization in the newly-created child thread instead. Thus, whenever thread T_i forks a new thread T_{i+1}, we set the values of *OptimizeOK* for the two threads as follows:

1. $\mathit{OptimizeOK}[T_{i+1}] = \mathit{OptimizeOK}[T_i]$

2. $\mathit{OptimizeOK}[T_i] =$ `true`

An additional safety concern arises when the machine is fully occupied with threads and a new thread (of non-lowest priority) is to be created. We wish to evict the last, most speculative thread T_n to make room the new thread, but the second-to-last thread T_{n-1} may have already optimized away some register restore operations. If so, blindly discarding T_n's state would lose the correct register values.

In this case we can simply squash and restart T_{n-1} after evicting T_n so that it will correctly execute the needed restore operations.

The register validation scheme described in Section 4.2 must also be modified to handle this optimization. When threads T_1 and T_2 meet, and T_1 had optimized some restore operations away, the stale final register value from T_1 may not match the correct value in T_2's input register. These discrepancies are ignored in the validation process; if all other "non-optimized" registers match, then T_2 becomes the head thread just as before. On the other hand, if any of the "non-optimized" register values do **not** match, then the correct register state that T_2 must start with consists of "non-optimized" register values from the final register file together with "optimized" register values in T_2's input registers.

4.6 Speculative Transaction Implementation

4.6.1 Basic Transaction Functionality

For transactions, all the important decisions are made by the software; programs use the TRY⟨ADDR⟩ , COMMIT, ABORT instructions to dictate when the speculation starts, when to commit the speculative state, when to abort the speculation. Unlike the case of procedural speculation, the hardware does not have to detect data hazards nor initiate roll backs on its own.

Let us first consider the simple case when transactions cannot be nested. When a TRY⟨ADDR⟩ instruction is issued, the machine does not need to start another thread, it only needs to save the current state in case an abort instruction is issued. This is achieved on our hardware by flash-copying the current register state into its input register state. Register mappings are also flash-copied, and any outstanding register writebacks (resulting from instructions issued before the transaction began) must update both the current and the input registers. In addition, the address supplied in the TRY statement is also stored as the target should an abort be issued. From this point on, any stores that are issued are part of the speculative state.

If the program executes an ABORT instruction, the speculative memory state is discarded, the registers are restored to the saved input register values, the program counter is set to the address the user specified with TRY, and the thread proceeds with normal non-speculative execution. If COMMIT is executed instead, the speculative memory state is committed, and the thread resumes normal execution.

While architecture-supported transactions are very efficient, they are limited by the amount of speculative state that can be stored in the hardware. The load/store queues, which hold the speculative state for procedural speculation, are expensive because of the logic needed to support data hazard detection. (In our base architecture, we assume that every thread has a load/store queue holding only 64 entries.) With traditional TLS, we can always discard a speculative thread if the speculative state storage is exhausted, or the thread can wait to become sequential at which point its state no longer needs to be buffered. Full support for transactions, however, requires that an arbitrary amount of state be buffered until the program or programmer decides to commit or abort.

To support larger-sized transactions, it is highly desirable to increase the amount of speculative buffering as much as is practical. We observe that data hazard detection is not needed in general for the implementation of transactions in a single-threaded environment. The only requirement is that speculative memory state must be prevented from overwriting the sequential state. We can expand our speculative storage with a simple extension to the first-level data cache.

All data written into the cache during speculative execution is labeled "speculative". If a transaction is commits, all the speculative bits in the cache are cleared and the state is deemed permanent. If a transaction is aborted, however, all cache lines marked speculative are invalidated. A single "speculative" bit per cache line is the only additional storage that is required for single-threaded transactional programs. Supporting transactions under multi-threaded execution, assuming the programmer has ensured there is no sharing of transactional data between threads, would require one additional bit per thread. Similarly, nesting of transactions within a single thread could be supported by introducing more bits per thread.

The size and associativity limits of first-level caches obviously presents a problem for buffering large transactions. A cache line that is marked speculative cannot be written back to the memory hierarchy. If such a line must be evicted from the cache due to conflict or capacity issues, the default behavior is to raise an exception indicating that there are insufficient architectural resources to handle the transaction. If the system does not handle the exception automatically and instead passes it to the program, this places a burden on the programmer to determine how to appropriately handle oversized transactions. Before we present our proposal for handling an overflow exception, we first outline a few mechanisms that could be used to automatically handle overflows:

Abort the transaction. To be safe, we could simply discard any transaction that was too large. While perhaps appropriate in cases when an oversized transaction might be a clue that an ABORT is desired (e.g. large buffer-overrun attacks), this solution presents a denial-of-service opportunity for large but valid transactions, or could result in an application being unable to make forward progress if the transaction is essential.

Commit the transaction. If it is essential that all valid transactions be able to proceed, we could simply commit transactions without validation when they get too large. While such transactions could be noted in a log for later review, a forced commit would negate much of the safety benefits gained by offering transactions in the first place.

Increase the amount of cache buffering. Buffering could be extended by adding speculative line bits to the second or third level caches, though this would result in increased cache design complexity. The larger cache buffers would still be insufficient for some transactions, of course, especially if associativity is limited.

Utilize coarser-granularity transactional support. If the operating system offers coarser-granularity transactions (e.g. Recovery-Oriented Computing [2]) the system could be designed to fall back to this option when transactions are too large.

We can see that the first three options each have significant limitations to their effectiveness, while the last option simply shifts the burden to a different transactional system. Because buffering in limited associativity caches presents significant variability the amounts of transactional state that can be handled, and given that our overall goal is to help make code more reliable, we feel that a robust overflow solution, even if potentially slow, is necessary. The next section describes our proposal, which utilizes an interrupt handler in the operating system to deal with transaction overflows in the cache.

4.6.2 Fallback Support for Larger Transactions

Our proposed scheme for handling larger transactions utilizes a "copy-on-write" policy whenever data overflows a particular set in the cache. The general idea of buffering is to prevent speculative data from overwriting the nonspeculative versions of that same data. When we have an overflow, however, what we can do is make a *duplicate* copy of the valid, nonspeculative data and then allow the speculative data to overwrite the *original* valid data and be stored in memory outside of the cache. Any valid, nonspeculative data that is in danger of being overwritten by speculative overflow will be preserved in a separate memory buffer, referred to as NS below.

We now outline the steps taken to preserve nonspeculative data from being overwritten. When a store of transactional data D occurs, and any candidate locations where it could be placed in the L1 are already filled with speculative ("pinned") transaction data, we invoke an operating system handler. Assume that $A = [A_1, ..., A_n]$ represents the set of addresses of all the speculative data items in the n-way associative cache set that is currently full and where we would like to store D.

1. Copy the speculative data from the cache set into memory, if necessary.

2. Load the nonspeculative data from the addresses in A.

3. For each address $a \in A$, store a, as well as its nonspeculative data, in NS if a has not previously been stored in NS.

4. Return the speculative data copied in Step 1 to the cache.
5. Clear the "speculative" bits in the cache set so the data is no longer "pinned" in the cache.

Note that we must be careful not to destroy either the speculative or nonspeculative versions of the data in this process. Thus steps 1 and 4 above are needed if the architecture does not have a "non-cacheable" load instruction that would allow the speculative data to remain in the cache while we load the nonspeculative versions of that data (in Step 2) directly into registers from memory, leaving the cache untouched.

After we allow speculative data to spill into memory, we must restore such memory if the transaction is not to commit. Thus the transaction abort procedure is modified to restore all the nonspeculative data buffered in NS, taking care to ensure that any speculative data in the cache is either overwritten with the correct values from NS or is invalidated if data NS is written only to memory. The commit procedure can operate much as usual, since the NS data is rendered stale after a transaction commit and can simply be discarded.

We must also be careful how the handler is implemented in order to correctly preserve the speculative and nonspeculative state–for example, the handler cannot rely on the ability to bring arbitrary data into the L1 data cache in order to operate. As such, the handler should exclusively use data pages that are mapped as non-cacheable in the virtual memory system in order to execute correctly. The page(s) that hold NS should neither be cached nor paged out to disk for the duration of the transaction. Overall, this handler will obviously execute quite slowly, but it is only designed as a fallback mechanism to allow for the correct completion of unusually large transactions. If the handler is invoked frequently, there should be some feedback path to the programmer suggesting that the transactions need to be modified to be of finer granularity, or that coarser-granularity transaction support from a different system would be more appropriate.

Chapter 5

Monitor-and-Recover Evaluation

This chapter presents an experimental evaluation of our proposals for using thread-level speculation to speed up monitoring code execution and support fine-grained transactions. Section 5.1 describes the baseline machines and evaluates the performance of monitoring code examples. Section 5.2 goes further into detail regarding the performance effects of various design decisions in the architecture. Finally, in Section 5.3 we investigate the use of fine-grained transactions to recover from some buffer-overrun examples.

5.1 Baseline Monitoring Code Simulation Results

We have implemented a simulator of our architecture based on the SimpleScalar 3.0c out-of-order simulator for the Alpha instruction set[3]. We extended the simulator to first support simultaneous multithreading and then to support thread-level speculation as discussed in Chapter 4. The main functionality our simulator represents about 20,000 lines of code, compared to roughly 5,000 for SimpleScalar as it is distributed. Validation functionality was also added to compare the final retirement instruction stream with that of a reference uniprocessor simulator to ensure that both were identical.

As in the base SimpleScalar simulator, system calls are executed by trapping through to the host operating system and cannot be executed speculatively. Thus

Parameter	SMT1	SMT2	SMT4
fetch width	4	8	16
issue width	4	8	16
commit width	4	8	16
reservation stations	128	256	512
lsq entries	64 p.t.	64 p.t.	64 p.t.
int ALUs	4	8	16
FP ALUs	4	8	16
mem. ports	2	2	4
lsq ports	2	4	8
fetch ports	2 ports, priority scheduled		
branch predictor	8k/8k entry gshare / bimodal, 8k meta predictor		
value predictor	AMA[7], L1: 128 entry 2-way SA, L2 (each): 2048 entry 4-way SA		
L1 Dcache	32K, 4-way SA (32B lines), 1 cycle latency		
L1 Icache	32K, 2-way SA (32B lines), 1 cycle latency		
L2 cache	4M, 4-way SA (64B lines), 6-cycle latency		
memory	100 cycles latency		

Figure 5.1: Parameters of the Simulation

speculative threads must be delayed until they become non-speculative if they attempt to make a system call. System calls are not allowed inside transactions.

Figure 5.1 shows the simulator parameters. Parameters labeled "p.t." represent per-thread values for non-shared resources, so an SMT2 processor with four threads would have 64 load-store queue entries per thread, or $64 \times 4 = 256$ entries across the whole machine. We simulate three sample SMT configurations, with the larger configurations typically having two or four times the overall base SMT1 resources. Memory interfaces are more limited, however.

As discussed in Section 4.1, the fetch unit services a maximum of two threads per cycle. For the data cache ports, we generally increase them more slowly than other resources. For example, the SMT4 configuration still has four memory ports, but we allow eight load-store queue (LSQ) ports to enable more loads to execute–either

Base Program	Monitoring	Inst Count	Inst Increase
Vortex	*(none)*	3.1B	
	Pixie	19.5B	6.2x
	Third	55.0B	17.4x
	DIDUCE.1	10.8B	3.4x
	DIDUCE	75.7B	24.0x
Perl	*(none)*	2.9B	
	Pixie	18.4B	6.3x
	Third	61.2B	21.0x
	DIDUCE.1	7.6B	2.6x
	DIDUCE	53.4B	18.3x

Figure 5.2: Dynamic Instruction Count Overheads of Monitored Programs

via store forwarding between threads or lookups for disambiguation. Note that in the text, we use the shorthand SMTm/tn notation to refer to an SMTm machine running with n thread contexts available.

Benchmarks we tested were all compiled using gcc 2.91.66 with `-O2` optimization. The version of ATOM used was 2.17d. Complete instruction counts were obtained using a fast functional simulator, while performance statistics were gathered by simulating one billion instructions of each execution after skipping well past the initialization phases of the base programs[45].

5.1.1 Monitoring Code Benchmarks

Our first set of experiments evaluate the performance of applications after adding instrumentation with the help of an automated tool. We experimented with three execution monitoring tools. The first two, Pixie and Third Degree, come as prepackaged tools with Compaq's ATOM. Pixie counts basic block execution frequencies and can be useful for profile-based optimizations. Third Degree, like Purify, finds potential memory access errors in programs. The third tool we experimented with is a simple version of DIDUCE. As discussed in Section 3.1.1, DIDUCE monitors data accessed by instructions to look for anomalies in programs and help locate algorithmic errors.

[1]Uninstrumented base program IPCs (italicized) are not included in the harmonic mean figures

Program	Monitoring Slowdown			Speedups			IPC		
	SMT1 /t1	SMT4 /t1	SMT4 /t8	4/1 vs. 1/1	4/8 vs. 1/1	4/8 vs. 4/1	SMT1 /t1	SMT4 /t1	SMT4 /t8
Vortex	–	–	–	–	–	–	*2.3*	*4.0*	–
Pixie	5.8	3.2	2.2	1.8	2.7	1.5	2.5	4.6	6.8
Third	18.3	10.2	6.4	1.8	2.8	1.6	2.3	4.1	6.5
DIDUCE.1	3.4	1.6	1.0	2.1	3.3	1.6	2.4	5.1	8.0
DIDUCE	24.4	12.0	7.2	2.0	3.4	1.7	2.3	4.8	7.9
Perl	–	–	–	–	–	–	*2.1*	*2.9*	–
Pixie	5.3	3.0	2.0	1.8	2.7	1.5	2.4	4.2	6.5
Third	18.5	12.6	8.8	1.5	2.1	1.4	2.3	3.4	4.8
DIDUCE.1	2.3	1.2	0.8	2.0	2.8	1.4	2.3	4.5	6.2
DIDUCE	15.8	7.6	4.6	2.1	3.4	1.6	2.3	4.8	8.0
Harmonic Mean[1]				1.9	2.8	1.5	2.3	4.4	6.6

Figure 5.3: Summary of Vortex and Perl Monitoring

DIDUCE was originally implemented for Java, but we created a simple version with ATOM that tracks the values generated by load instructions. DIDUCE allows users to adjust the level of instrumentation according to their needs. To simulate different degrees of instrumentation, we tested two configurations: a heavy-weight version where every static load instruction in the program is instrumented, and a more lightweight version where only every 10th static load in the binary is instrumented. We will refer to the former experiment as DIDUCE and the latter as DIDUCE.1.

We applied these instrumentations to two base programs: Vortex, an object-oriented database, and Perl, the PERL language interpreter, both from the SPECint benchmark suite. We used the training sets from CINT95 (specifically jumble.pl for Perl) for inputs. The dynamic instruction counts of the base programs as well as their monitored executions are shown in Figure 5.2. The final column lists the factor by which instruction count increased when instrumentation was added to the base program. That factor ranged from 2.6x when DIDUCE.1 is applied to Perl up to 24x when DIDUCE is applied to Vortex.

5.1.2 Baseline Performance of Monitoring Code

Figure 5.3 shows an overview of how our monitoring examples performed. First we ran the both the original base programs as well as the instrumented versions on the SMT1/t1 machine in order to determine the slowdown factors[2] caused by monitored execution. These factors range from 2.3x to 24.4x—similar to the range of factors by which dynamic instruction count increased. When we increase the resources of the machine to the more generous 16-wide SMT configuration but still execute with only one thread (SMT4/t1), the computations speed up by an average factor of 1.9 due to the additional ILP exploited. But by adding TLS features supporting eight thread contexts to the machine (SMT4/t8), we are able to obtain an *additional* 1.5x performance gain over the single-threaded SMT4 that has comparable resources. Overall, performance goes up by an average of 2.8x relative to SMT1/t1.

The lightest instrumentation, DIDUCE.1, originally slows the program by factors of 3.4 and 2.3 for Vortex and Perl, respectively. However, the SMT4/t8 configuration is able to reduce the 3.4x slowdown to 1.02x—just a two percent overhead—and the 2.3x overhead is more than eliminated. In the latter case the monitored execution actually runs faster than the original program thanks to exploiting both instruction-level and thread-level parallelism. The heaviest instrumentation, DIDUCE on Vortex, takes only 7.2 times as long with SMT4/t8, down from more than 24 times. Figure 5.3 also notes the committed IPC of the executions, which on average have gone up from 2.3 to 6.6.

Also of interest is how close we can get to SMT4/t8's achieved IPC with fewer SMT resources and a smaller number of thread contexts available. The overall IPCs achieved with the SMT2 and SMT4 configurations executing with 1, 2, 4 or 8 threads are shown in Figure 5.4. The base IPC with SMT1/t1 is also plotted as a single data point. While many of the programs show significant performance improvement going from four to eight available threads, the Pixie instrumentations benefit particularly in this case, as the threads are relatively short and can utilize more contexts. Overall, providing eight thread seems worthwhile—for both SMT2 and SMT4, only about 75%

[2]Slowdown factor is calculated by dividing the execution time of the instrumented program by the execution time of the uninstrumented (base) program

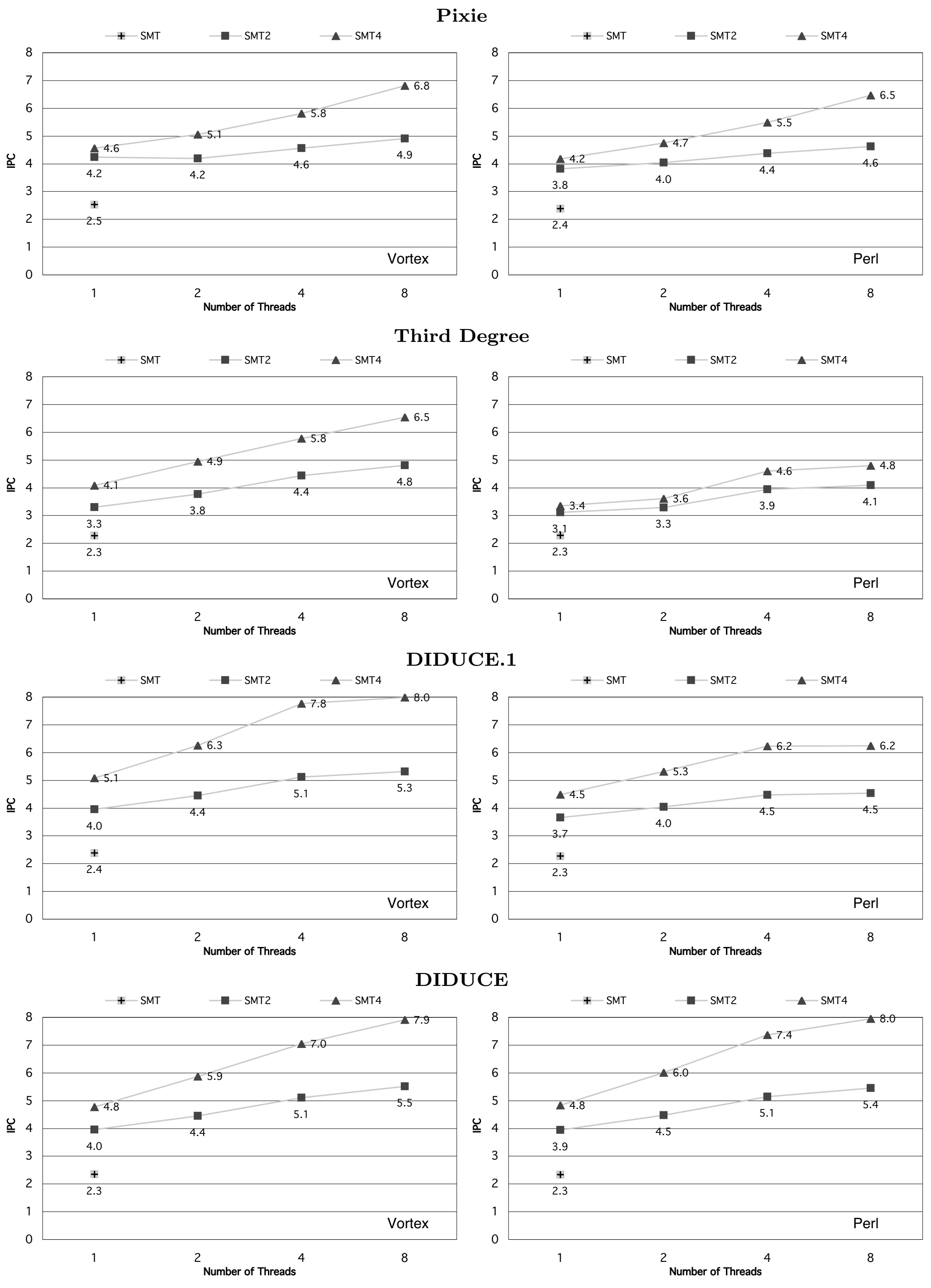


Figure 5.4: Performance of Execution Monitoring on the Proposed Architectures

of the IPC that is gained by using eight threads can be achieved when a maximum of four threads is allowed.

To further understand the performance of programs on our proposed architecture, we show in Figure 5.5 some internal statistics from the SMT4/t8 runs. At the top, the speedup from TLS execution is included for reference. We show the average number of threads active each cycle, as well as what ultimately becomes of those threads: whether they are committed, evicted to make room for higher priority threads, or orphaned and deleted. Of the threads that do retire, we show what percentage had been restarted due to memory or register data violations before commit, as well as the number of instructions they executed and committed. For the cycle-based metrics, we distinguish between machine cycles and "thread cycles", where n threads active during one machine cycle represents n thread cycles. Thus the LSQ full percentage expresses a likelihood that an active thread will find its own load-store queues full in any given cycle.

Programs that achieved the best TLS speedups, such the full DIDUCE instrumentations of both Vortex and Perl, tend to have low violation rates and threads of consistent and moderate length. Third Degree applied to Perl, on the other hand, results in widely varying thread lengths. Because threads must wait to commit in program order, large variations in thread lengths can leave short threads sitting idle while waiting for longer (higher-priority) threads to commit first. In addition, widely varying thread lengths generally indicate rather "bursty" thread creation. In some periods there may be an excessive number of very short threads created, which are inefficient on their own in terms of relatively constant per-thread overheads (like validating registers) but may also be evicting longer, more suitable outer-level threads. At the other extreme, long-running sections of code without thread creation would tend to lower the number of threads occupying the machine, limiting the parallel speedup that could be achieved. We refer to these overall difficulties caused by widely-varying thread lengths as "load imbalance". As an example of the difficulties with Third Degree applied to Perl experiences, we note that this execution ultimately commits just 51% of the threads that are forked. Relatively poor branch prediction in this benchmark also contributes to the number of orphaned and deleted threads. Third

Statistics	Pixie		Third		DIDUCE.1		DIDUCE	
	Vortex	Perl	Vortex	Perl	Vortex	Perl	Vortex	Perl
TLS speedup	1.5	1.5	1.6	1.4	1.6	1.4	1.7	1.6
average # of active threads	4.2	4.3	3.2	4.7	2.5	2.1	3.4	3.2
% of threads								
committed	99	91	80	51	98	83	99	93
evicted	0	0	1	10	0	0	0	0
deleted	1	9	19	39	2	17	1	7
% of committed threads with								
memory violation	0.0	0.0	5.5	1.7	0.3	0.5	0.1	0.0
register violation	0.0	0.0	0.4	8.3	0.0	0.0	0.0	0.0
% of committed threads with								
0-10 instructions	0	0	9	12	0	0	0	0
11-25 instructions	10	8	1	11	0	0	0	0
26-50 instructions	89	91	9	15	0	0	0	0
51-100 instructions	1	1	55	26	68	62	96	97
101-200 instructions	0	0	17	7	32	30	3	2
201-500 instructions	0	0	9	29	1	8	1	1
500+ instructions	0	0	1	0	0	0	0	0
% of thread cycles with								
LSQ full	0	0	3	2	1	2	1	0
% of machine cycles with								
0 fetches	30	28	20	12	21	17	28	26
1 fetch	28	30	28	24	31	44	25	23
2 fetches	42	43	52	64	48	40	46	51
% Icache miss rate	0.64	0.50	0.59	0.10	0.41	0.19	0.64	0.54
% Dcache miss rate	0.21	0.23	0.40	2.26	0.24	0.11	0.19	0.20
% L2 miss rate	0.27	0.89	0.23	4.21	1.44	4.36	0.25	0.42
% Branch misprediction rate	0.14	0.51	5.53	7.06	0.45	1.66	0.90	0.95

Figure 5.5: SMT4/t8 Statistics

Degree applied to Perl also has a significant number of register violations, the bulk of which are return values that were not correctly predicted.

The statistics also indicate that this rather demanding instrumentation, Third Degree applied to Perl, motivates the substantial 4-megabyte level-two cache that we assume. All other benchmarks performed about the same with a significantly smaller level-two cache.

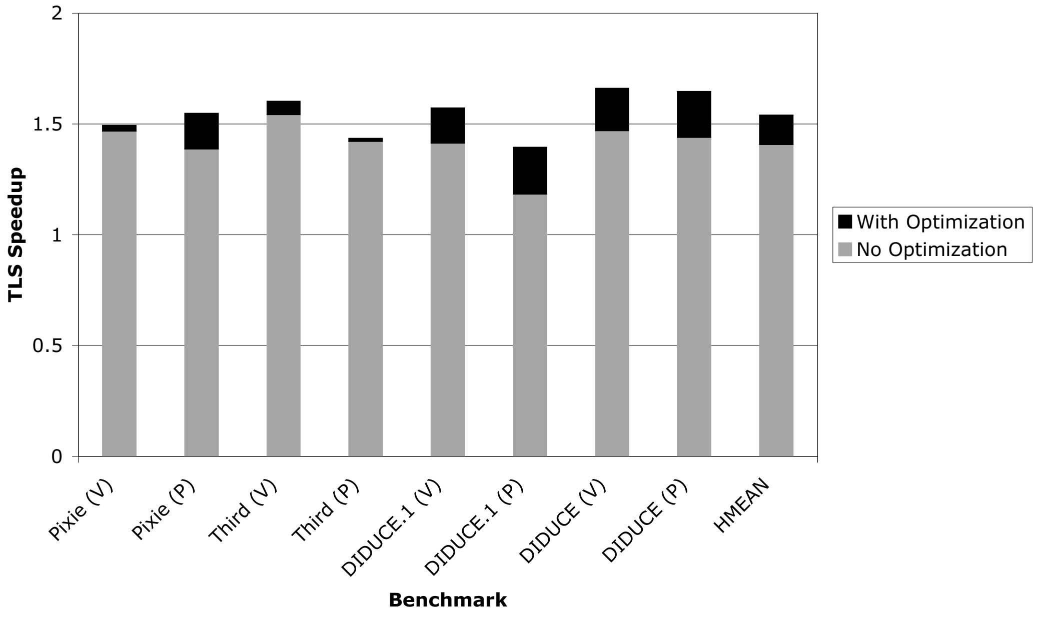


Figure 5.6: Effect of Calling Convention Optimization

5.2 Design Decisions and Monitoring Performance

The previous section showed how our monitoring code examples performed based on the overall size of the machine and number of threads available. In this section we explore how different microarchitectural design decisions and optimizations affect the speedup we obtain from thread-level speculation. The graphs in this section show changes to the baseline SMT4/t8 versus SMT4/t1 (or TLS) speedup resulting from different design decisions. If the change made to the machine is equally applicable to both single-threaded and TLS execution, then the change is of course made to both machines before the new TLS speedup is determined.

5.2.1 Effect of Calling Convention Optimization

In Section 4.5, we described a calling convention optimization that dynamically skips unneeded register restore operations in monitoring code procedures. The effects of

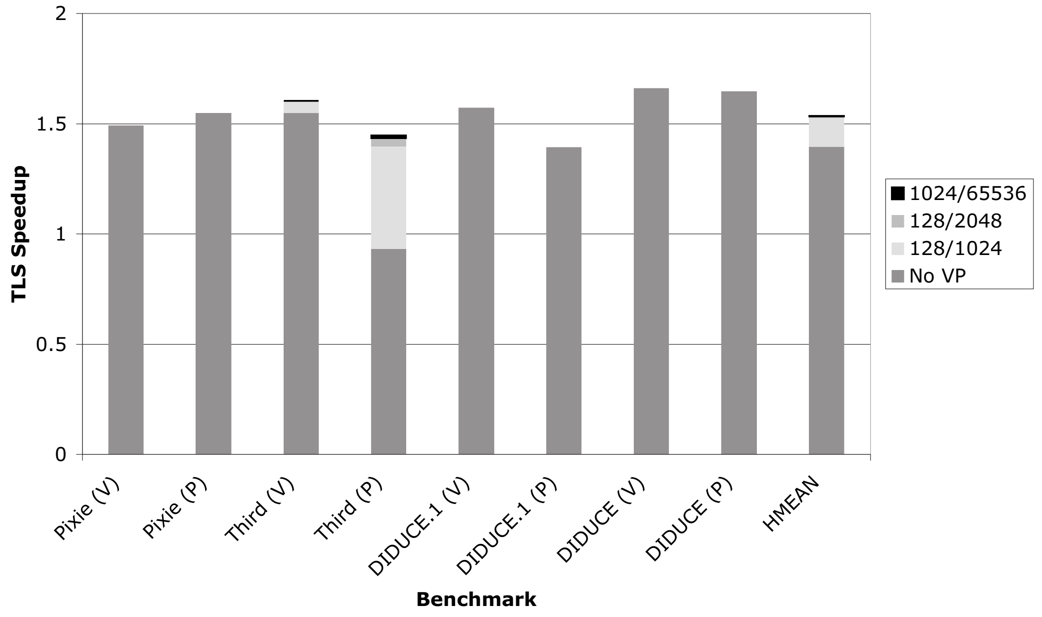


Figure 5.7: Effect of L1/L2 Value Predictor Table Sizes

this optimization are included in the baseline results presented in Section 5.1. Figure 5.6 shows just how much this optimization improved our TLS performance. On average, the TLS speedup is about 9% better than it would have been without the optimization.

5.2.2 Effect of Return Value Prediction

Here we examine the effect that return value prediction has on the performance of our benchmarks. Section 4.4 describes the two-level value predictor we use, and Figure 5.7 shows the performance results for predictors of various sizes. Of the benchmarks we examined, only Third Degree benefited from return value prediction. In fact, Third Degree applied to Perl requires value prediction to achieve any speedup at all. The results show that a 128-entry first-level predictor table and 1024-entry second level tables achieve most of the speedup. Doubling the sizes of the two second-level tables to the default configuration of 2048 entries increases the performance modestly, while an extremely large predictor provides almost no further benefit.

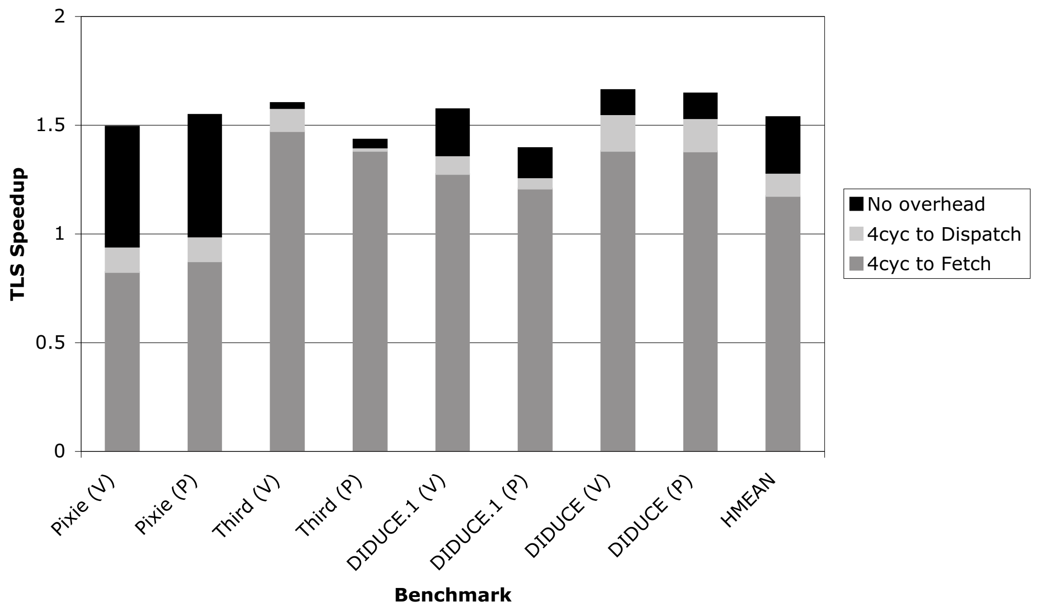


Figure 5.8: Effect of Thread Initiation Overheads

We should note that simple last-value or stride-value predictors were insufficient for the value predictions needed by Third Degree.

5.2.3 Effect of Thread Initiation Overheads

In our base design for thread speculation, we assume single-cycle flash copy of register resulting in no overheads in starting new speculative threads. Here we look at the performance achievable with less aggressive implementations. First, we assume a four cycle delay from the time a new thread is created until it is allowed to begin fetching. This reduces performance significantly, especially for Pixie monitoring, as the monitoring routine is so small (on the order of ten to twenty instructions) that any substantial thread creation overhead negates the benefits of TLS.

A second implementation option is made possible by recognizing that copying the register state is the most costly part of creating a new thread. The starting fetch address for the new thread is just a single value that is known immediately at creation time, so a new thread could actually begin fetching while its input register state is

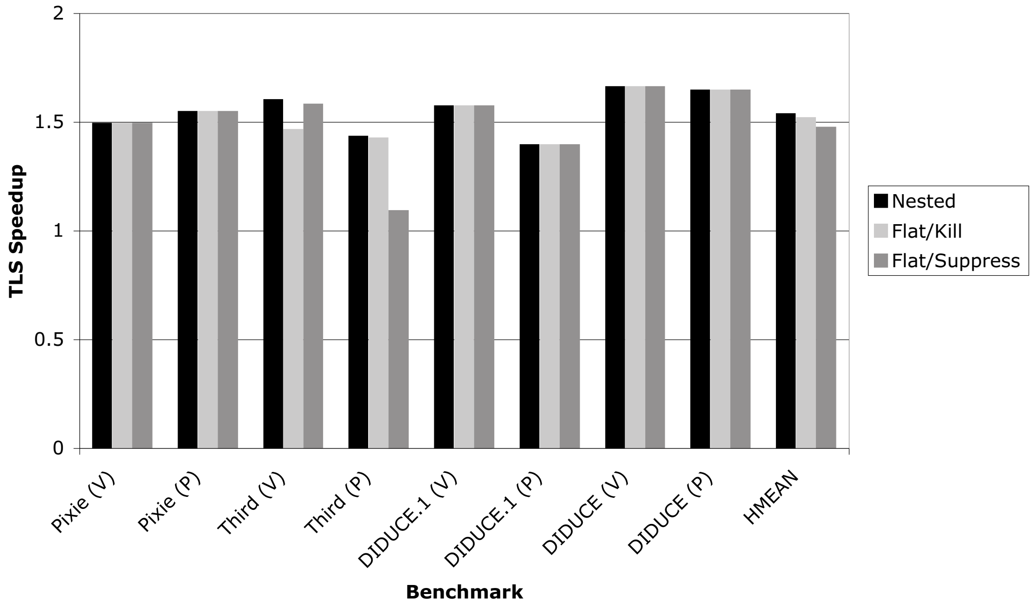


Figure 5.9: Effect of Allowing Nested Speculation

being copied. Thus the "4cyc to dispatch" bar in Figure 5.8 shows the performance assuming a new thread can begin fetching in the cycle immediately after creation, but still must wait four cycles before its dispatch/rename pipeline stage is allowed execute. This design is still not sufficient to execute Pixie well, but the DIDUCE instrumentation in particular benefits from not needlessly delaying fetch.

5.2.4 Effect of Nested Speculation

General-purpose procedural speculation typically allows threads to be created in a nested fashion, out of program order, as described in Section 2.5. However, some of the more practical TLS proposals have found that the richer support for nested procedural speculation can result in significant overheads for thread operations, particularly if those operations are implemented in software[34]. In addition, support for nested speculation requires that some per-thread hardware structures, such as the load-store queues in our proposal, be able to reflect the dynamic thread ordering created at runtime. This implies full connectivity between the per-thread elements, as a crossbar or bus-based connection topology would provide. Out-of-order nested thread creation would not be compatible with a static and linear hardware-based ordering that a ring-based topology, for example, would give.

Speculating on procedures can still be supported with simpler hardware and/or simpler software if we restrict nested speculation at runtime. Figure 5.9 shows performance results of two variations on this approach. With the first, labeled "Flat/Kill", threads speculating on outer procedure calls are automatically killed when a higher-priority (less speculative) thread wishes to fork. The second, labeled "Flat/Suppress" only allows forking by the *most* speculative thread currently in the machine; any attempted forks by higher-priority threads are suppressed. Of course, a single executing thread is both highest and lowest priority, so it would be able to fork in both variations.

The Pixie, DIDUCE, and DIDUCE.1 instrumentations show no change in performance because the speculated instrumentation routines are leaf procedures; no nested thread creation is even attempted. Third Degree does create nested threads, and while performance degrades in all cases, neither of the two schemes is preferable across the board. An analysis of thread priority at the time of attempted forks suggests why. In the baseline case where we allow nesting, 52% of threads created with Perl are nested (not of lowest priority when created), while with Vortex the figure is only 19%. Thus the programs have an overall preference as to where they create threads. Still, the overall point to be made is that our performance results are not highly dependent on full support for nested thread creation.

5.2.5 Effect of Front-end Pipeline Stages

Figure 5.10 shows the effects of adding extra pipeline stages to the front-end of the six-stage pipeline shown in Figure 4.2. The graph labels are of the form $(+m/+n)$, where m is the number of extra stages inserted between fetch and decode/rename ("fetch stages"), while n is the number of extra stages inserted between decode/rename and issue ("issue stages"). Only additional fetch stages effectively delay the creation of new threads, while both types lengthen the overall pipeline and increasing the branch

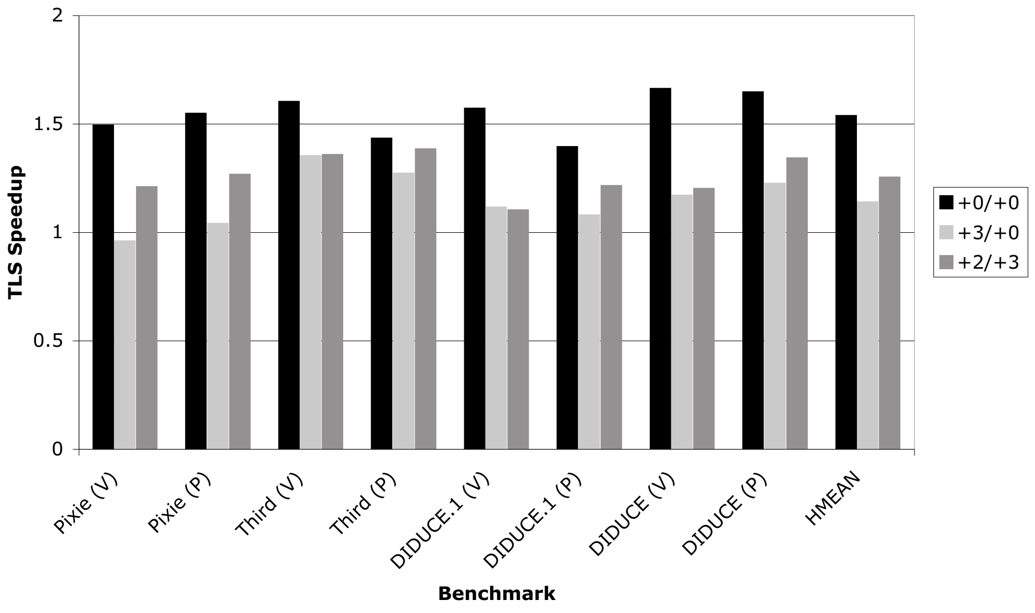


Figure 5.10: Effect of Additional (Fetch/Issue) Front-end Pipeline Stages

misprediction recovery time. First we examine adding three additional fetch stages only. This makes our TLS execution of the Pixie instrumentation impractical, as the monitoring routine is quite small (about fifteen instructions). The other benchmarks also suffer significant performance loss; the mean TLS speedup for them is only 1.2. Some of the performance could potentially be gained back by initiating new thread fetch at an earlier stage, much as was described with thread overheads in Section 5.2.3. As we see with the next set of bars, performance is significantly less sensitive to extra issue stages. If we instead add just two fetch stages but then add three extra stages before issue, the results are almost uniformly better than we obtained with three fetch stages, particularly for the Pixie instrumentations. Clearly these threads need to be started early in a long pipeline if they are to be effective.

5.2.6 Effect of Number of LSQ Entries

Figure 5.11 shows the TLS performance when each thread is given a load-store queue with 32, 64 (the default), or 128 entries. The effects are fairly minimal and somewhat

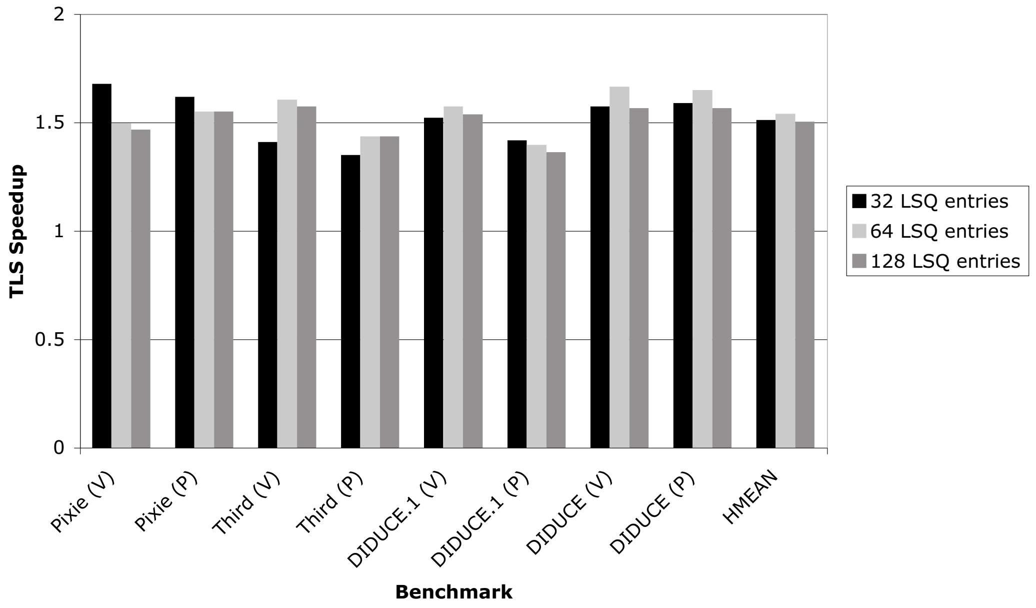


Figure 5.11: Effect of Number of LSQ Entries

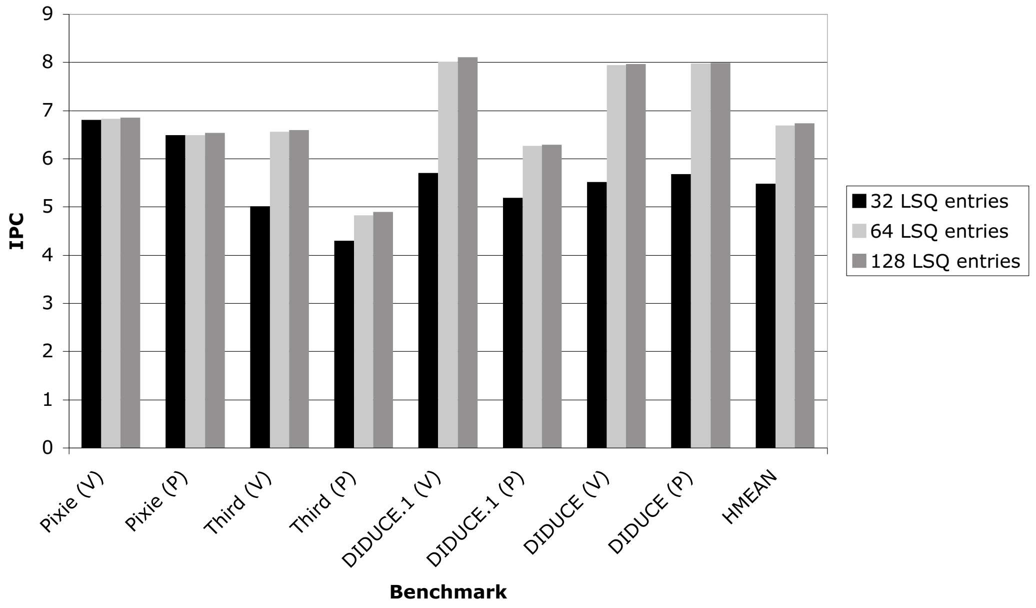


Figure 5.12: Effect of Number of LSQ Entries on IPC

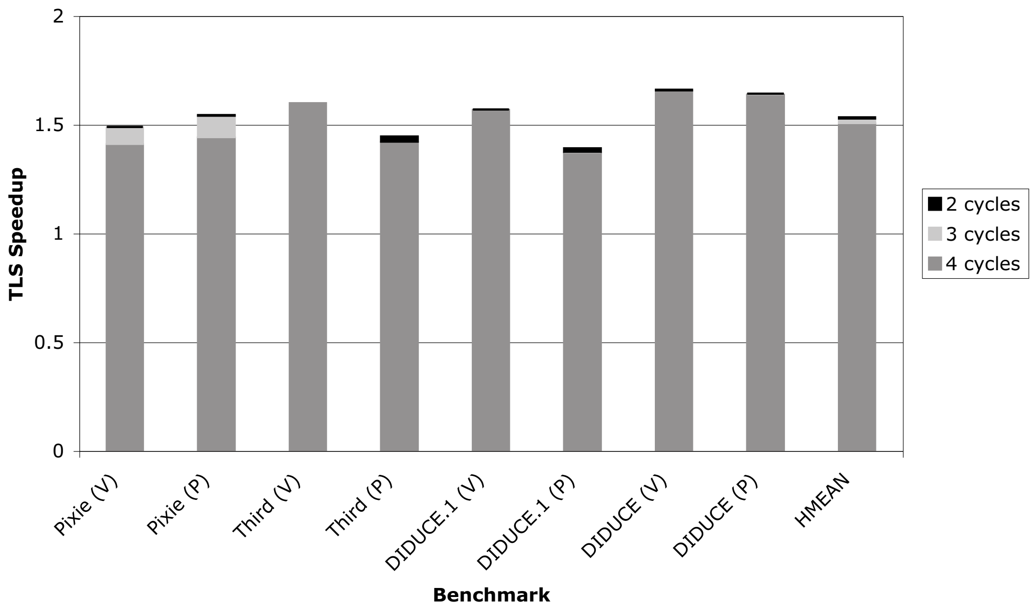


Figure 5.13: Effect of Interthread LSQ Hit Latency

variable, largely because the additional queue entries help the baseline single-threaded performance that the TLS performance is measured against. The raw IPC performance of SMT4/t8, shown in Figure 5.12, increases from 5.46 to 6.67 when moving from 32 to 64 entries per queue, and again increases slightly to 6.72 with 128 entries. Thus a queue size of 64 entries seems to provide the most of the achievable performance.

5.2.7 Effect of Interthread Load-store Queue Latency

Using the load-store queues to buffer the speculative state as well as detect data dependence violations does significantly complicate their design. Our base design assumes a one-cycle hit time if a matching store is contained in a thread's own load-store queue, but a two-cycle hit time if instead the only match (or matches) are found in the queues of of other, higher priority threads. This two-cycle latency includes the time needed to do a priority selection of the multiple hits that may be found in other threads' queues as described in Section 4.3. Figure 5.13 shows how performance is

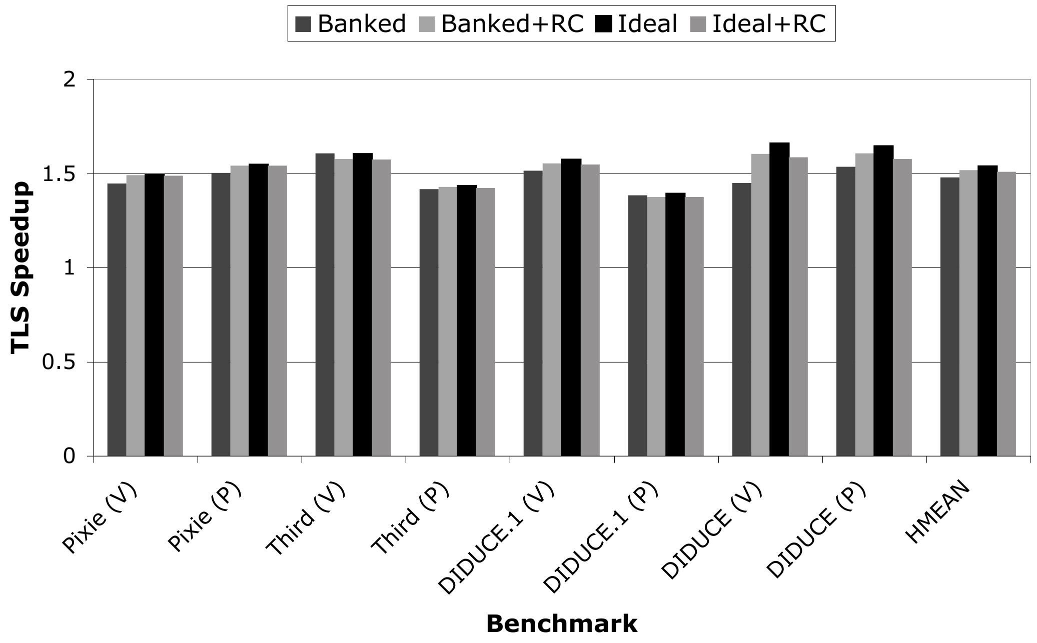


Figure 5.14: Effect of Memory Port Design

affected if the interthread latency requirements are relaxed and we instead assume a 3-cycle or 4-cycle latency for interthread queue hits. We see that Pixie is the instrumentation that is most sensitive to this latency; the other benchmarks are relatively insensitive to modest increases in the interthread hit latency.

5.2.8 Effect of Memory Port Design

Figure 5.14 shows how TLS performance changes based on how the memory interface and ports are designed. First off, all the configurations we examine utilize write-combining, meaning that stores attempting to write to the same cache line in the same cycle are able to share a single cache port. This is particularly important to expedite the bursts of writes that occur when a thread commits.

Our base SMT4 design, whose performance is shown with the black bars,‘ uses ideal memory ports; that is, each port can be used to access any location. When we move to a banked cache organization, where the first-level data cache is four-way line

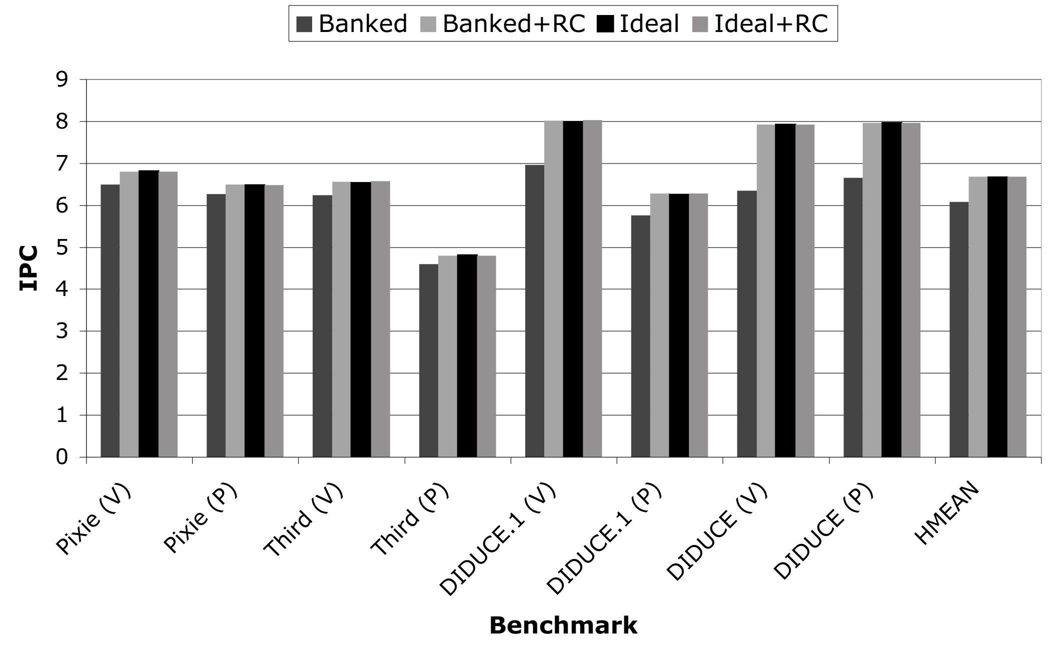


Figure 5.15: Effect of Memory Port Design on IPC

interleaved for the four memory ports, the performance impact is relatively modest.

For both the banked and ideal implementations, we examined an additional read-combining optimization, where multiple loads can be satisfied by a single port access if they are accessing the same cache line, much like the "load all wide" optimization described by Wilson[59]. This optimization significantly helps with the banked configuration, but with ideal ports the TLS performance gain is reduced when read-combining is added. This is because we also add read-combining to the baseline single-threaded performance (SMT4/t1) before calculating TLS speedup, and read-combining actually provides more benefit to a single thread than overall to multiple threads. For load instructions to be combined into a single cache access, each load must still separately access an LSQ port to ensure that no matching stores are present. In the single-threaded case, lower contention for the LSQ-only ports allows more extensive use of read-combining. If we look at the overall IPC numbers for SMT4/t8 in Figure 5.15, we see that the IPC achieved for the configurations is pretty similar, apart from the reduced performance of banked memory without read-combining.

Overall read-combining appears to be a useful optimization when memory ports are limited (banked or interleaved) but does not appear very beneficial when the memory port organization is already fairly aggressive.

5.2.9 Summary

Our experiments show that thread-level speculation is a useful mechanism for reducing the runtime overheads associated with monitoring code and instrumentation. A wide, simultaneous multithreading-based machine is able to exploit quite a bit of instruction-level parallelism in the code, but adding thread-level speculation support provides a substantial boost to overall performance. While the overheads are by no means eliminated, we believe that instrumented programs represent another class of programs for which thread-level speculation is quite appropriate.

5.3 Fine-grained Transactions

We now turn to the second part of the monitor-and-recover proposal–the use of fine-grained transactions for recovery. In Section 5.3.1, we describe some sample programs that could benefit from fine-grained transactional recovery, as well as the error detection mechanism that are used. Next, we evaluate whether the buffering capabilities of our machine are sufficient for these examples in Section 5.3.2.

5.3.1 Example Programs

To demonstrate the ability of our proposed machine to monitor for buffer overruns and recover from them using transactions, we examined some sample vulnerabilities in common UNIX programs. We use detection methods based on two tools designed to catch these types of errors: Libsafe[53] and StackGuard[9]. Libsafe is a library of replacement versions for unsafe C string library functions, while StackGuard instruments the run-time stack to check if the return addresses have been tampered with.

Our first example is an ftp daemon implementation `bftpd`, version 1.0.22. The program has a vulnerability when processing an ftp `chown` command, due to a call to `sscanf()` from the `cmd_chown()` routine. We wrap the `cmd_chown()` routine into a transaction and return a failure code if the transaction aborts. The monitoring code we use to detect this error is a modified version of `sscanf()` from Libsafe. Before executing the potentially dangerous code, it does a backwards traversal of the stack and copies frame pointer and return address values into its own private storage. It then executes the `sscanf()` and traverses the stack again to see if any frame pointers or return address were overwritten. If so, the transaction is aborted.

The second example is `imapd`, the IMAP daemon from the Pine 4.00 distribution. It contains a call to `strcpy()` from the `mail_auth()` function that can potentially cause an overflow. We wrap the `mail_auth()` function to create our transaction, and in case of an abort we return NULL. The detection code we use is again from Libsafe, but in this case the stack traversal only needs to be done once. Before executing the real `strcpy()`, the monitoring code locates which particular stack frame contains the destination buffer. If the length of the source string is greater than the size of that frame, an error is signaled and we abort the transaction.

Our final example is `ntpd`, the network time protocol daemon, version 3.1.1. In the `ctl_getitem()` routine, which processes user input strings, there is a bug in the string manipulation code that can result in an overrun of a static buffer. We detect this error using a process much like StackGuard instrumentation. We manually add guard locations around the static variables of the procedure, and initialize them to known values at the beginning of the call. After the body of the procedure has executed, but before it returns, we check these guard locations to see if the values have been overwritten. If so, we abort and return NULL. If the values are correct, we commit the transaction before returning.

We should note that the stack traversal code necessary for our Libsafe implementation (used in the `imapd` and `bftpd` examples) is complicated by the standard Alpha calling conventions. In these conventions, the frame pointer is not stored at a constant position in the frame. Because of this, we implemented a static analysis which examines the program binary and outputs, for each call site, the offset at which the frame pointer is stored in the callee's frame. This information is saved to a file, then read into a hash table at the beginning of program execution. During runtime, when we wish to follow a frame pointer back to its parent frame, we dereference the frame pointer to obtain the return address, which is then used to index into the hash table to yield the frame pointer offset. This offset is added to the current frame pointer and the result is dereferenced to yield the location of the parent frame.

Note that if a callee could not be determined statically (due to an indirect call through a function pointer, for example), a heavyweight system library call can be used to determine the parent frame pointer, though this did not occur in our examples.

5.3.2 Fine-grained Transaction Experimental Results

We have validated our implementation on our simulator, and measured the amount of state that was buffered in each transaction, as shown in Figure 5.16. First we show the natural amount of data generated by the original code that is wrapped by executing it with native C library functions and no instrumentation. The next two experiments, suffixed with `inst`, show the amount of state generated in the monitor-and-recover version of the benchmarks for both valid inputs that allow the transaction to commit, as well as an exploit input that causes a buffer overrun and aborts the transaction.

The sample transactions we looked at all fit into our default 4-way set associative 32K L1 data cache. Of course, larger transactions or smaller, less associative caches would present a problem. A 2-way 16K L1, or a 1-way 32K L1, for example, cannot buffer our largest transaction: `bftpd-inst` running with exploit inputs. In fact, even with valid data inputs, `bftpd-inst`'s transaction is too large for a 16K direct-mapped cache or a 4K 2-way associative cache. For cases such as these, we examined how often the operating system handler described in Section 4.6 would need to be invoked.

We distinguish between two possible modes of operation. First, the handler could only be invoked on store operations. If we have a load miss and the target cache set is full of speculative data, could just fetch the missing line, provide the needed data to the CPU, and then immediately discard the line instead of saving it in the L1 cache. This policy minimizes the number of handler invocations but reduces the effectiveness of the cache in minimizing load latency–effectively speculative stores would have priority over loads in occupying the cache. In the second mode of operation, the handler executes for any load or store that wishes to allocate an entry in a cache set that is already full with speculative data. The choice of operation should be made based upon the relative overheads of executing the handler more often vs. the increased memory access time if loads are not able to bring data into the L1 for repeated access. Given the expected overheads, invoking the handler only for stores seems it would typically be the best choice.

Figure 5.17 shows how often the operating system handler must be invoked when the `bftpd-inst` example is executed with valid inputs on our machine with smaller, less associative L1 data caches. Any L1 caches larger or more associative than the configurations listed have sufficient buffering to not require the handler. The worst case in this example for the store handler requires roughly 3.5% of stores in the transaction to invoke the handler (23 invocations out of 658 stores).

We determine the overall L1 miss rate of the transaction by executing it repeatedly in a tight loop until the miss rate stabilizes. The "native" L1 data cache miss rate for the instrumented transaction is shown in the third column; this is the natural miss rate obtained for the transaction if no transactional buffering (and therefore no "pinning" of data) was done in the L1. We see that the store-only handler does indeed require significantly fewer invocations, though it does cause a 14% increase in cache misses with a 4K direct-mapped cache. Keep in mind that most of these transaction-induced level-one cache misses will at least wind up hitting in the second-level cache, however.

The larger `bftpd-inst` execution with exploit inputs does require more handler executions–up to 176 when executing with a direct-mapped 4K L1. Still, the handler is only executing for 1.7% of the 10,371 stores performed in the transaction. This seems like an acceptable response to such an aberrational data input.

Application	Input	Transactional State		
		bytes	cache lines	# of stores
ntpd	valid	92	5	15
ntpd-inst	valid	132	8	26
ntpd-inst	exploit	312	13	227
bftpd	valid	796	34	353
bftpd-inst	valid	1,180	49	658
bftpd-inst	exploit	2,072	74	10,371
imapd	valid	40	2	10
imapd-inst	valid	248	10	82
imapd-inst	exploit	368	15	110

Figure 5.16: Buffer Overrun Transactions

			Store Handler only		Loads and Stores	
Cache Size	Assoc	Miss Rate	Handler Invocations	Miss Rate	Handler Invocations	Miss Rate
8K	2	1.25%	1	1.55%	2	1.29%
4K	2	3.28%	1	3.60%	2	3.35%
16K	1	1.58%	6	1.93%	8	1.58%
8K	1	4.32%	13	4.96%	22	4.32%
4K	1	6.56%	14	7.50%	23	6.56%

Figure 5.17: Handling L1 Transaction Overflows for bftpd-inst, valid input

5.3.3 Summary

Our buffer-overflow examples show how fine-grained transactions can be a useful tool in helping programmers write more reliable code. Certainly the fine-grained transaction support we propose is not suited for all programming situations, but we feel that the more tools the programmer has available, the better. We also evaluate how a fall-back handler-based solution can be used to allow transactions to complete, albeit slowly, when limited hardware buffering is exhausted.

5.4 Related Work

Speculative thread-level parallelism has been proposed by many different researchers, typically for the purposes of speeding up sequential integer programs. We discuss a number of these proposals at length in Section 2.8.

While many of the proposed machines could support the programming models we have described, our base architecture most closely resembles the DMT[1] and IMT[37] processors, which are both extensions of the Simultaneous Multithreading machine proposal[55]. The DMT and IMT are described in general in Section 2.8, so here we will only outline the main differences between the machines and our proposed machine.

The DMT does not have a mechanism for predicting return values, so we add a small value predictor for monitoring instrumentations that may communicate occasionally with the main computation, or complicated instrumentations that are decomposed into multiple functions. The DMT speculates on procedure continuations and loop continuations (the code after a loop), while we discuss only procedures here. Loop speculation is not relevant to the instrumentation code and transactions that we wish to support. The most significant difference between the machines is that the DMT does trace buffering of the speculative thread execution; this allows for selective recovery of only those instructions which must be re-executed due to data dependences. While this is obviously preferable, performance wise, to our scheme where all instructions in a thread must be re-executed, the implementation costs of this trace re-execution ability appear to be quite high.

The IMT is more like our machine in that it does not have trace buffers for selective thread recovery. Threads, however, are chosen using compiler heuristics, with preference given to loop iterations. As general TLS performance on the base IMT was initially slower than single-thread performance, a number of optimizations were added, such as thread-context multiplexing, resulting in an overall integer speedup of 1.2.

Transactional memory implementations have been proposed using a variety of implementations: hardware[20, 47], software[44], and combinations thereof[28]. The hardware implementation proposed by Herlihy is closest to ours[20]. The intention is to provide lock-free execution when shared data is used by multiple processors. Software can issue a direct instruction to abort a transaction, but the commit instruction is only a request, as the transactional may have already been invalidated by another processor.

Herlihy's proposal utilizes a separate, fully associative transaction cache to lessen overflows caused by set conflicts. The transaction cache proposed is smaller than the primary cache by a factor of thirty-two, however. This suggests that the transactional (shared) data is expected to be small, and that non-shared variables will not be transactional–even though they may be used and updated concurrently with transactional variables. Thus it is the programmer's responsibility to determine which memory accesses need to be transactional and which do not. In our scheme, all memory accesses are transactional, as we focus on providing single-threaded "undo" functionality as opposed to lock-free synchronization.

Rollback and recovery in large-scale distributed systems have been studied extensively[12]. The Recovery-Oriented Computing proposal includes a conceptual extension of "undo" functionality (common in desktop applications) to more complicated system administration tasks, such as configuring a mail server[2]. They have implemented a mail server that uses logging and checkpointing to, for example, "roll back" a virus infected or misconfigured server, allow for the correction of the error, and then "replay" logged external interactions so that those valid transactions are not lost. Additional research looks at reducing the time it takes to reboot or restart a service in order to increase availability.

Our goal is to support fine-grained transactions with little or no overhead so they can become prevalent in everyday code. Speculative thread-level parallelism is being considered by many microprocessor designers, and chips utilizing it have already been released[52]. If chips implementing TLS become widely deployed, we feel it would be an excellent opportunity to leverage that functionality to deliver additional usability and reliability without much additional cost.

Executing "subordinate" code (such as prefetching code or exception handlers) in separate threads has been suggested by a number of researchers to improve performance, but those threads never create any committed side effects[61].

Patil and Fischer have proposed Shadow Processing to hide the overheads of run-time checking by executing the checks in a separate program on another processor[38]. This shadow program is generated by a source-to-source tool which slices the original program into a reduced form containing only the instructions needed for checking. Any non-reproducible computations, such as user inputs or system calls, are forwarded from the original to the shadow. They find that the resulting overheads in the original program are typically below 10%, but the shadow process can take up to ten times longer to complete. Of course, the continued execution of the main program is not recoverable. Thus, unlike our scheme, it is not generally possible for the checker to interact with or even stop the main process before serious errors or data corruption have occurred. For non-interactive "passive" instrumentation, however, there is a clear performance benefit from the user's perspective.

There have been proposals to use speculation to provide efficient support for locking. Martinez and Torellas propose buffering the states of critical sections in the processor cache to allow that code to execute while waiting in the background for the lock to be made available[32]. This can bring usability benefits to the program writer in that they can use coarser-granularity locks without suffering the usual performance degradation that results. Rajwar and Goodman suggest speculative lock elision, where lock operations are speculatively eliminated[41]. The on-chip write buffers are made part of the coherent state in order to allow for effectively atomic updates.

Chapter 6

Conclusions

Thread-level speculation has been the subject of intense research interest, but as of yet has not been found to provide substantial performance improvements to general-purpose uniprocessor code without rather costly hardware assumptions. A stronger case for TLS has been made in various programming subdomains: Java programs, multimedia programs, irregular scientific codes, and programs modified via manual transformations, for example. We believe TLS can also be used to decrease the overheads associated with runtime instrumentation, as well as provide new software capabilities to the program writer.

In summary, we advocate the use of a monitor-and-recover programming paradigm to create more reliable software, and propose an architectural design that allows the software and hardware to cooperate in making this paradigm more efficient and easier to use.

Programmers simply write monitoring functions assuming the normal sequential execution semantics. For recovery, routines that may not complete properly are wrapped as "transactions" whose side effects can be discarded or committed as a whole. Our proposed TRY...ABORT...CATCH syntax for transactions is similar to that of an exception-handling construct, but it is much easier for programmers to deal with because all side effects, including register and memory updates, are discarded after an abort operation.

To support this monitor-and-recover paradigm efficiently, our proposed architecture gives software the control over the basic hardware mechanisms originally designed for thread-level speculation. We let the software specify that procedural speculation should be applied to specific monitoring functions. Because of the relative independence between the original code and the instrumentation, speculative parallelism is likely to be effective, thus hiding much of the costs of monitoring. The machine still guarantees sequential programming semantics, so exceptions or corrective actions can be handled accurately and safely should the monitoring function fail. We also show how our transaction construct can be translated directly into machine instructions that control when speculation begins and when to abort or commit the speculative state.

Our experiments with four different examples of execution monitoring suggest that TLS hardware can significantly reduce the overhead of monitoring–by a factor of 1.5 over a very wide superscalar with similar execution resources. Our average IPC of 6.6 is significantly greater than typical results obtained when thread-level speculation is applied to arbitrary sequential programs. As such, monitored execution represents an additional class of programs where TLS execution can be quite effective. We also show how the concept of fine-grain transactional programming is potentially useful in catching and recovering from buffer overrun exploit examples.

There is obviously more work to be done to justify the expense of adding thread-level speculation support to general-purpose processors. Achieving good general-purpose TLS performance has proven very challenging and costly from a hardware perspective. Recent trends in computer architecture have been more reflective of computing environments where superb single-threaded program performance may not be the solitary goal, so mainstream CPUs are increasingly being designed with multiple cores and multiple threads available. Still, TLS could represent an attractive way to harness these multiple threads for specific single-threaded programs that are suitable, and as we've shown can provide qualitative benefits to the user as well. Apart from hardware trends, the reliability of software (or lack thereof) looks as if it is becoming a larger problem every day. With the abundance of computation resources available and the rather sorry state of software reliability in general, we feel confident that some forms of hardware support for more reliable and trustworthy code are certain to be deployed in the future.

Bibliography

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

[2] A. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[3] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, Computer Sciences Department, University of Wisconsin-Madison, 1996.

[4] M. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

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

[6] L. Codrescu and D. S. Wills. On dynamic speculative thread partitioning and the MEM-slicing algorithm. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 40–46, October 1999.

[7] L. Codrescu, D. S. Wills, and J. D. Meindl. Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE Transactions on Computers*, 50(1):67–82, 2001.

[8] DAT Collaborative. uDAPL: User direct access programming library, 2003. Available at `http://www.datcollaborative.org/`.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, Q. Zhang P. Wagle, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, January 1998.

[10] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–844, August 1986.

[11] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, April 2001.

[12] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, October 1996.

[13] R. J. Figueiredo and J. Fortes. Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, September 2001.

[14] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grained parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, May 19–21, 1992.

[15] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 195–205, January 31–February 4, 1998.

[16] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

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

[18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.

[19] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, December 1992.

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

[21] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[22] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[23] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, 1986.

[24] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Conference Proceedings of the 1998 International Conference on Supercomputing*, pages 85–92, July 13–17, 1998.

[25] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 19–21, 1992.

[26] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[27] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1–5, 1996.

[28] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[29] P. Marcuello and A. González. A quantitative assessment of thread-level speculation techniques. In *14th International Parallel and Distributed Processing Symposium*, pages 595–604, May 2000.

[30] P. Marcuello and A. González. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 55–64, February 2002.

[31] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *Conference Proceedings of the 1998 International Conference on Supercomputing*, pages 77–84, July 13–17, 1998.

[32] J. F. Martínez and J. Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Proceedings of the Workshop on Memory Performance Issues at the 28th Annual International Symposium on Computer Architecture*, June 2001.

[33] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual*

International Symposium on Computer Architecture, pages 181–193, June 2–4, 1997.

[34] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the Hydra CMP. In *Proceedings of the 1999 Conference on Supercomputing*, pages 21–30, June 1999.

[35] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Computer Systems Laboratory, Stanford University, 1997.

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

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

[38] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, 1995.

[39] K. Poulsen. Software bug contributed to blackout. *Security Focus*, February 2004. Available as `http://www.securityfocus.com/news/8016`.

[40] M. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ACM/SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming*, 2003.

[41] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[42] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, November 1997.

[43] Y. Saito and B. Bershad. A transactional memory service in an extensible operating system. In *Proceedings of the 7th USENIX Security Conference*, pages 53–64, January 1998.

[44] N. Shavit and D. Touitou. Software transactional memory. In *Proceeedings of the Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[45] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, Department of Computer Science and Engineering, University of California, San Diego, August 1999.

[46] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 415–425, June 1995.

[47] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. Dept. of Computer Sciences Technical Report CS-TR-2000-1420, University of Wisconsin-Madison, October 2000.

[48] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[49] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–24, June 2000.

[50] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, February 2002.

[51] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Conference on High-Performance Computer Architecture*, pages 2–13, January 1998.

[52] Sun. MAJC architecture tutorial. Technical report, Sun Microsystems Inc., 1999.

[53] T. Tsai and N. Singh. Libsafe 2.0: Detection of format string vulnerability exploits. White paper, Avaya Labs, February 2001.

[54] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Conference on High-Performance Computer Architecture*, pages 14–23, January 1998.

[55] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[56] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[57] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 81–92, November 1998.

[58] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 8–11, 1991.

[59] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing cache port efficiency for dynamic superscalar microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 147–157, May 1996.

[60] A. Zhai, C. B. Colohan, J. G., and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems*, pages 171–183, October 2002.

[61] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 219–229, November 1999.