Hardware Support for Thread-Level Speculation

J. Gregory Steffan

CMU-CS-03-122

April 2003

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Todd C. Mowry, Chair
Seth C. Goldstein
David O'Hallaron
Joel Emer, Intel

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy*

This research was sponsored by the National Aeronautics and Space Administration (NASA) under grant nos. NAG2-1230 and NAG2-6054, and by a generous donation from the Intel Corporation. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any sponsoring party or the U.S. Government.

Keywords: thread-level speculation, chip-multiprocessing, automatic parallelization, distributed computing, cache coherence, value prediction, dynamic synchronization, instruction prioritization.

Abstract

Novel architectures that support multithreading, for example *chip multiprocessors*, have become increasingly commonplace over the past decade: examples include the Sun MAJC, IBM Power4, Alpha 21464, and Intel Xeon, HP PA-8800. However, only workloads composed of independent threads can take advantage of these processors—to improve the performance of a single application, that application must be transformed into a *parallel* version. Unfortunately the process of parallelization is extremely difficult: the compiler must prove that potential threads are independent, which is not possible for many *general-purpose programs* (e.g., spreadsheets, web software, graphics codes, etc.) due to their abundant use of pointers, complex control flow, and complex data structures. This dissertation investigates hardware support for *Thread-Level Speculation* (TLS), a technique which empowers the compiler to optimistically create parallel threads despite uncertainty as to whether those threads are actually independent.

The basic idea behind the approach to thread-level speculation investigated in this dissertation is as follows. First, the compiler uses its global knowledge of control flow to decide how to break a program into speculative threads as well as transform and optimize the code for speculative execution; new architected instructions serve as the interface between software and hardware to manage this new form of parallel processing. Hardware support performs the run-time tasks of tracking data dependences between speculative threads, buffering speculative state from the regular memory system, and recovering from failed speculation. The hardware support for TLS presented in this dissertation is unique because it scales seamlessly both within and beyond chip boundaries—allowing this single unified design to apply to a wide variety of multithreaded processors and larger systems that use those processors as building blocks. Overall, this cooperative and unified approach has many advantages over previous approaches that focus on a specific scale of underlying architecture, or use either software or hardware in isolation.

This dissertation: (i) defines the roles of compiler and hardware support for TLS, as well as the interface between them; (ii) presents the design and evaluation of a unified mechanism for supporting thread-level speculation which can handle arbitrary memory access patterns and which is appropriate for any scale of architecture with parallel threads; (iii) provides a comprehensive evaluation of techniques for enhancing value communication between speculative threads, and quantifies the impact of compiler optimization on these techniques. All proposed mechanisms and techniques are evaluated in detail using a fully-automatic, feedback-directed compilation infrastructure and a realistic simulation platform. For the regions of code that are speculatively parallelized by the compiler and executed on the baseline hardware support, the performance of two of 15 general-purpose applications studied improves by more than twofold and nine others by more than 25%, and the performance of four of the six numeric applications studied improves by more than twofold, and the other two by more than 60%—confirming TLS as a promising way to exploit the naturally-multithreaded processing resources of future computer systems.

Acknowledgements

As I reflect over the past five years, I realize how numerous are the people that I must thank for contributing to this work in many ways. I am indebted to my advisor, Todd Mowry, for the quality of my graduate career. His wisdom and optimism guided me through two graduate schools and degrees, while teaching me how to do good research and share it with the world. I am also grateful to Todd for arranging two enlightening internships, for his commitment to my professional development, and for his friendship. I also thank the members of my thesis committee, Joel Emer, David O'Hallaron, and Seth Goldstein, for their contributions to this dissertation and for steering it to successful completion.

This work would not have been possible without the efforts of others. I'd like to express my sincere gratitude to Chris Colohan and Antonia Zhai for building and tuning the compiler infrastructure used in this dissertation, and for years of enjoyable collaboration, advice, and companionship. I also thank the rest of the STAMPede'rs for listening to practice talks, being an effective sounding board for new ideas, and for their support—technical and otherwise. For making my days at CMU memorable (whether working or not) I thank Angela Demke-Brown, Chris Palmer, Jason Flinn, Ted Wong, and Kip Walker.

I am quite fortunate to have worked with two phenomenal groups of people in industry. Earl Killian and the architecture groups at MIPS (a division of SGI at the time) provided me with an introduction to computer architecture in the real world, and Joel Emer and the VSSAD architecture group (DEC→Compaq→Intel) helped develop initial support for TLS in a shared cache, and also demonstrated how to mix research and fun; all have my sincere gratitude.

There are also many people from whom I have benefited beyond academics. I thank Joe, Carrie, Addie, John, and Pete for much needed relief from computers during my stay in Pittsburgh. Since leaving the University of Toronto, Derek, Dan, Rob, Alex, Steve, Chris F., Masis, Stan, Sergio, and Aryn showed continued friendship and thoughtfulness. For their constant support, endless creativity, and mostly for keeping life interesting, I deeply thank my friends

Alana, Angela S., Champ, Chris B., Chris E., Eastman, Fiona, Jake, Jason, Kathy, Katie, Kev C., Kev H., Laurel, Liisa, Lucky, Mark, Maya, Mayte, Natalie, Rick, Rose, Ryan, Scooter, Scott, Stacey, Tim, and Tyler.

My time in graduate school would not have been possible without the support of family. I thank my parents-in-law Hans and Lea Schwarzbauer as well as my sister-in-law Dawn for many years of warmth and understanding. I also thank my parents Sac and Susan Steffan and my brother Ryan for providing constant encouragement, and helping me to remember what is truly important. Finally, I thank my wife Nancy who through infinite patience and unconditional love has given me the confidence to rise to many challenges.

Contents

List of Figures

List of Tables

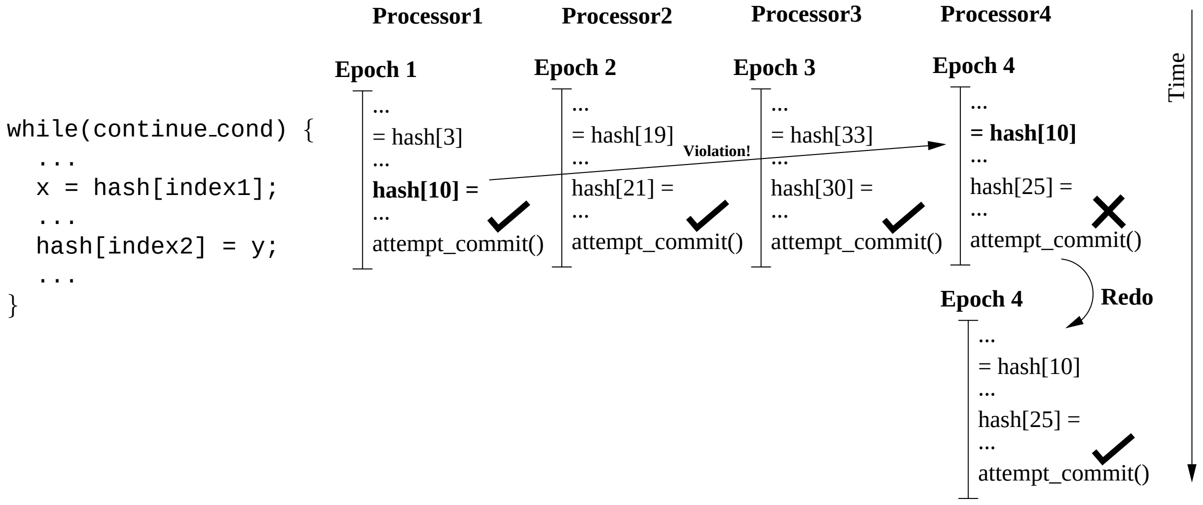
Chapter 1

Introduction

Due to rapidly increasing transistor budgets, today's microprocessor architect is faced with the pleasant challenge of deciding how to translate these extra resources into improved performance. In the last decade, microprocessor performance has improved steadily through the exploitation of *instruction-level parallelism* (ILP), resulting in superscalar processors that are increasingly wider-issue, out-of-order, and speculative. However, this highly-interconnected and complex approach to microarchitecture is running out of steam. Cross-chip wire latency (when measured in processor cycles) is increasing rapidly, making large and highly interconnected designs infeasible [1, 58]. Both development costs and the size of design teams are growing quickly and reaching their limits. Increasing the amount of on-chip cache eventually shows diminishing returns [22]. Instead, an attractive option is to exploit *thread-level parallelism* (TLP).

The transition to new designs that support multithreading has already begun: the Sun MAJC [68], IBM Power4 [38], and the Sibyte SB-1250 [8] are all *chip-multiprocessors* (CMPs), in that they incorporate multiple processors on a single die. Alternatively, the Alpha 21464 [20] supports *simultaneous multithreading* (SMT) [19, 70, 71], where instructions from multiple independent threads are simultaneously issued to a single processor pipeline. These new architectures still benefit from ILP, since ILP and TLP are complementary.

While it is relatively well-understood how to design cost-effective CMP and SMT architectures, the real issue is how to use thread-level parallelism to improve the performance of the software that we care about. Multiprogramming workloads (running several independent programs at the same time) and multithreaded programs (using separate



```
while(continue_cond) {
  ...
  x = hash[index1];
  ...
  hash[index2] = y;
  ...
}
```

(a) Pseudo-code.

(b) TLS execution.

Figure 1.1. Example of thread-level speculation (TLS).

threads for programming convenience, such as in a web server) both naturally take advantage of the available concurrent threads. However, we often are concerned with the performance of a single application. To use a multithreaded processor to improve the performance of a single application we need that application to be *parallel*.

Writing a parallel program is not an easy task, requiring careful management of communication and synchronization, while at the same time avoiding load-imbalance. We would instead like the compiler to translate any program into a parallel program automatically. While there has been much research in this area of automatic parallelization for *numeric* programs (array-based codes with regular access patterns), compiler technology has made little progress towards the automatic parallelization of *non-numeric* applications: progress here is impeded by ambiguous memory references and pointers, as well as complex control and data structures—all of which force the compiler to be conservative. Rather than requiring the compiler to prove independence of potentially-parallel threads, we would like the compiler to be able to parallelize if it is *likely* that the potential threads are independent. This new form of parallel execution is called *thread-level speculation* (TLS).

1.1 Example

TLS allows the compiler to automatically parallelize portions of code in the presence of statically ambiguous data and control dependences, thus extracting parallelism between whatever dynamic dependences actually exist at runtime. To illustrate how TLS works, consider the simple `while` loop in Figure 1.1(a) which accesses elements in a hash table. This loop cannot be statically parallelized due to possible data dependences through the array `hash`. While it is possible that a given iteration will depend on data produced by an immediately preceding iteration, these dependences may in fact be infrequent if the hashing function is effective. Hence a mechanism that could speculatively execute the loop iterations in parallel—while squashing and re-executing any iterations which do suffer dependence violations—could potentially speed up this loop significantly, as illustrated in Figure 1.1(b). In this example, the program is running on a shared-memory multiprocessor, and some number of processors (four, in this case) have been allocated to the program by the operating system. Each of these processors is assigned a unit of work, or *epoch*, which in this case is a single loop iteration. When complete, each epoch attempts to commit its speculative work. In this case a *read-after-write* (RAW) data dependence violation is detected between *epoch 1* and *epoch 4*; hence *epoch 4* is squashed and restarted to produce the correct result, while *epochs* 1, 2, and 3 commit. This example demonstrates the basic principles of TLS.

1.2 Related Work

This section provides an overview of the field of related work, which can be divided into three categories: loosely-related work, related work done prior to this dissertation, and related work done concurrently. An in-depth discussion of how this dissertation is differentiated from related work is given in each successive chapter.

1.2.1 Loosely-Related Work

Concepts similar to thread-level speculation have been explored in concurrency for databases, aggressive simulation, and functional languages. For databases, the concept of speculative synchronization [43, 45, 46] as an alternative to mutual exclusion was a matter of great debate in the database community; the results of this debate are summarized by Agrawal *et al.* [2]. The time warp operating system [36] is an optimistically-parallel discrete event simulation engine, where a simulation is parallelized with optimistic synchronization. Events are simulated early—when the input parameters may possibly still change. In the case of misspeculation, the results of the simulated event are rolled back and the event is re-simulated to ensure correctness. Finally, Knight [39] and Halstead [61] each proposed forms of thread-level speculation within the context of functional languages.

1.2.2 Prior Related Work

Herlihy et al. [35, 53] proposed using transactional memory to support TLS for numeric codes; however, this work was quite preliminary, and not explored in detail. The following two works are more relevant: the LRPD test [60] which is a software-only version of TLS that applies only to array-based codes, and the Multiscalar architecture [27, 65] which was the first complete evaluation of an architecture designed specifically for supporting TLS. More detail about each of these schemes follows.

The LRPD Test

Padua *et al.* [60] devised a method for parallelizing loops for numeric codes in the presence of ambiguous data dependences. Their entirely software-based approach, called the *LRPD* test, allows the compiler to parallelize loops without fully disambiguating all memory references. For a given loop, the LRPD test is performed on each shared variable with ambiguous references by creating corresponding *shadow arrays* to track read and write accesses. These shadow arrays are examined at the end of the parallel execution of the loop: if any cross-iteration data dependences were violated, the loop is re-executed sequentially; otherwise the parallel execution of the loop was correct.

Although a purely software-based approach is attractive, there are two shortcomings to the LRPD test. First, the LRPD test requires the creation of shadow storage for all shared data, and is therefore not applicable to most non-numeric codes due to their complex data structures and extensive use of heap-allocated objects. Second, the LRPD test does not extract any parallelism in the presence of a single cross-iteration *read-after-write* dependence: the loop is re-executed sequentially in such cases. In summary, only a narrow class of loops can be parallelized effectively using the LRPD test.

The Multiscalar Architecture

The most relevant prior work is the Wisconsin multiscalar architecture [26, 27, 65]. This architecture performs aggressive control and data speculation through the use of large hardware structures devoted to the task. The following describes the multiscalar system architecture, beginning with the compilation process.

During compilation for multiscalar execution, a program is broken up into small *tasks*. A task may consist of a only few instructions or perhaps several basic blocks. The compiler inserts a bit-vector called the *create mask* into every task denoting which registers are live at the end of the task. Hardware then uses this information to forward register values between tasks.

The multiscalar architecture speculatively executes tasks in parallel by executing the tasks before the corresponding branches and memory dependences have been resolved. Processors in the multiscalar architecture are arranged in a ring, and each processor is tightly-coupled with its two neighboring processors. At any given time during execution, one of the processors is considered the *head* of the ring: this processor is executing the oldest task, and its work is not considered speculative since there are no longer any preceding tasks. There are two main benefits to the ring architecture. First, the ordering between tasks is implied by the order of the processors in the ring, thus making it trivial to terminate the appropriate tasks when a dependence is violated. Second, the tight-coupling of adjacent processors in the ring simplifies register forwarding, since register values are forwarded between consecutive tasks.

As described earlier, the create mask informs hardware which registers are live at the end of each task, and hardware automatically forwards these register values between tasks. Synchronization of register values that are communicated between tasks is provided by a *busy bit* that is associated with each register which is set when the register value arrives at the receiving processor.

To support data speculation, the multiscalar architecture includes the *address resolution buffer* (ARB) [27] which performs dynamic memory disambiguation. The ARB sits between the processors and the first-level cache, and all memory accesses are filtered through it. For each store to memory, the store address and the value are saved in one of the ARB's associative entries for the corresponding processor. On each memory access, the ARB is searched for previous accesses to the same address: if the ARB notices that a store and load to the same address have occurred out-of-order then all speculative tasks including and beyond the violating task are terminated.

Although the multiscalar architecture facilitates efficient control and data speculation, it does so at the cost of an architecture which is highly specialized to that mode of execution. The ring layout of processors is beneficial for forwarding data between consecutive tasks, but is not efficient when executing a conventional parallel program. The ARB is a large and complex structure; since an associative search is performed for most memory accesses, latency through the ARB is longer than that of an ordinary first-level cache. Therefore the multiscalar architecture will not be efficient at executing multiprogramming workloads or even conventional parallel programs. However, experimental results for the multiscalar architecture show that speculative execution is a promising way to improve the performance of non-numeric applications using multiple processors.

Table 1.1. Summary of related hardware schemes for TLS.

Approach	System Composition	Underlying Architecture	Applications Supported	Speculative Buffering
DMT [4]	HW-only	SMT	general-purpose	buffer between processor and L1
CSMP [50]	HW-only	SMT	general-purpose	L1 cache
Trace Processor [62]	HW-only	CMP	general-purpose	(not specified)
Krishnan99 [42]	SW/HW	CMP	general-purpose	L1 caches
Hydra [34]	SW/HW	CMP	general-purpose	buffers between L1s and L2
SVC [31]	SW/HW	CMP	general-purpose	L1 caches
SUDS [25]	SW/HW	CMP (RAW)	general-purpose	undo-log/buffer at each processor
Zhang99 [78]	SW/HW	MP	numeric	undo-log/buffer at each node
Cintra00 [12]	SW/HW	MP/CMP	numeric	L1 and L2 caches
Our approach	SW/HW	MP/CMP/SMT	general-purpose	L1 caches

1.2.3 Concurrent Related Work

There are currently many approaches to support for TLS. Of these, one class of schemes are implemented entirely in software [32, 60, 63] but require explicit code and storage to track data dependences and buffer speculative modifications (or provide an undo-log); these schemes are thus only effective for array-based, numeric applications where the portions of memory for which cross-thread dependences need to be tracked are well defined. A software-only approach to TLS support for arbitrary memory accesses (pointers) is infeasible.

Approaches to TLS that involve hardware support may be divided into two classes: those that are implemented entirely in hardware [4, 50, 62], and those that use both hardware and software [12, 25, 31, 34, 41, 42, 59, 78], as summarized in Table 1.1. Hardware-only approaches have the advantage of operating on unmodified binaries, but are limited since hardware must decide how to break programs into speculative threads without knowledge of high-level program structure. Hardware-only approaches are generally more complex than those that take advantage of software support since any transformation and optimization of TLS execution must be implemented entirely in hardware, in addition to the mechanisms for selecting speculative threads.

These schemes may also be classified based on the underlying architecture: those that focus on chip-multiprocessor (CMP) architectures [25, 31, 34, 41, 56, 78][1], those that focus on simultaneously-multithreaded (SMT) or other shared-cache architectures [4, 50], and those that scale beyond a single chip to multiprocessor (MP) architectures [12, 32, 59, 60, 63, 77, 79]. With the exception of the scheme by Cintra *et al.* [12], no related approach is scalable both within a chip and also beyond a chip to multiprocessor systems; i.e., no related approach is applicable to multiprocessor

[1] Note that the SUDS scheme [25] is implemented using the MIT RAW reconfigurable architecture, as opposed to extending a conventional CMP.

systems, chip-multiprocessors, *and* shared-cache architectures (such as SMT).

Related approaches can also be differentiated by the class of applications that are supported. Two approaches [12, 78] focus solely on numeric applications, while the rest focus on general-purpose applications. Any scheme which supports general-purpose applications will also function correctly for numeric applications. However, no related work contains a thorough evaluation of both general-purpose and numeric applications. It is also important to note that related schemes that do scale beyond a chip (MP) focus solely on numeric applications.

Finally, related approaches demonstrate a wide variety of hardware implementations, of which one of the most important features is the mechanism for buffering speculative state and tracking data dependences between speculative threads. For this purpose, dynamic multithreading (DMT) [4], SUDS [25], and Zhang *et al.* [78] introduce a new buffer near the processor; the latter two approaches speculatively modify memory and use the buffers to maintain an undo log, while the former uses its buffers to separate speculative modifications from memory. The Hydra [34] introduces speculative buffers between the write-through first-level caches and the unified second-level cache. These speculative buffers must be sufficiently large to be effective, but adding large amounts of speculation-specific buffering to the memory hierarchy is undesirable. The remaining approaches [12, 31, 42, 50][2] use the existing caches as speculative buffers. A comprehensive summary and quantitative comparison of several schemes for TLS support is provided by Garazan *et al.* [29].

The next section describes the goals that differentiate this dissertation from related work.

1.3 Research Goals

While there are many important issues regarding the role of the compiler in TLS, this thesis focuses on the design of the underlying hardware support. Thus, my thesis is that: **hardware support for thread-level speculation that is simple and efficient, and that can scale to a wide range of multithreaded architectures can empower the compiler to improve the performance of sequential applications.** Furthermore, a deep understanding of the key aspects of this novel hardware support—tracking data dependences and buffering speculative state, detecting and recovering from failed speculation, management of speculative threads, communication of data values between speculative threads, and the partitioning of these mechanisms between hardware and software—can be obtained.

[2] The Trace Processor [62] approach does not specify means for buffering speculative modifications and tracking data dependences, deferring instead to related work.

The hardware support for TLS presented in this dissertation achieves the following four goals:

1. to handle arbitrary memory accesses—not just array references;

2. to preserve the performance of non-speculative workloads;

3. to provide a framework for scaling seamlessly both within and beyond chip boundaries;

4. to fully exploit the compiler and minimize the amount and complexity of the underlying hardware support.

While some previous efforts achieve a subset of the above goals, none achieves all four—hence these goals differentiate this work.

1.4 Overview

The remainder of this dissertation is organized as follows. Chapter 2 provides the background necessary for the remainder of the dissertation. We define TLS terms and its execution model, and describe a cooperative approach to TLS hardware support—defining the roles of compiler and hardware and the interface between them. Also presented are the compiler infrastructure, benchmark applications, and simulation framework used in the evaluations in this dissertation. The chapter concludes with an estimate of the potential performance benefits of TLS for the benchmark applications.

In Chapter 3, we present the design for a unified mechanism for supporting thread-level speculation which can handle arbitrary memory access patterns and which is appropriate for any scale of architecture with parallel threads. The evaluation in this chapter focuses on support within a chip-multiprocessor, while in Chapter 4 we evaluate further support for speculation in a shared cache as well as the scalability of our approach beyond chip boundaries.

Chapter 5 provides a comprehensive evaluation of techniques for enhancing value communication between speculative threads, including several schemes for prediction and synchronization. The impact of compiler optimization on these techniques is also quantified.

Finally, we conclude in Chapter 6 by enumerating the contributions of this dissertation and discussing potential future directions. Note that the Appendix defines the speculative cache coherence scheme in full detail

Chapter 2

Thread-Level Speculation

2.1 Introduction

This chapter provides the background necessary for the rest of the dissertation. We begin with a description of the high-level execution model for TLS targeted by software as well as the software interface to architected TLS support, including the definitions of new instructions. The compilation infrastructure including thread selection, code transformation, optimization, and code generation are all described, although an in-depth evaluation of compilation issues for TLS [76] is beyond the scope of this thesis[1]. This chapter also presents the experimental framework used in the evaluations throughout this dissertation including the benchmark applications and simulation infrastructure. Finally, through a preliminary simulation study we demonstrate the potential for TLS to improve performance.

2.2 Execution Model

This section describes the execution model for TLS that is targeted by the compiler and implemented in hardware. The following description also serves as a high-level overview of the the remainder of this section.

First, we divide a program into speculatively-parallel units of work called *epochs*. Each epoch is associated with its own underlying speculative thread, and creates the next epoch through a lightweight fork called a `spawn`, as shown in Figure 2.2(a). The spawn mechanism forwards initial parameters and a program counter (PC) to the appropriate processor. An alternative approach would be to have a fixed pool of speculative threads that grab epochs from a

[1] See http://www.cs.cmu.edu/ stampede for an in-depth treatment of compilation issues for TLS

Epoch: The unit of execution within a program which is executed speculatively.

Epoch Number: A number which identifies the relative ordering of epochs within an OS-level thread. Epoch numbers also indicate that certain parallel threads are unordered.

Homefree Token: A token which indicates that all previous epochs have made their speculative modifications visible to memory and hence speculation for the current epoch is successful.

Logically Earlier/Later: With respect to epochs, *logically-earlier* refers to an epoch that preceded the current epoch in the original execution while *logically-later* refers to an epoch that followed the current epoch.

OS-level Thread: A thread of execution as viewed by the operating system—multiple speculative threads may exist within an OS-level thread.

Speculative Context: The state information associated with the execution of an epoch.

Sequential Portion: The portion of a program where TLS is not exploited.

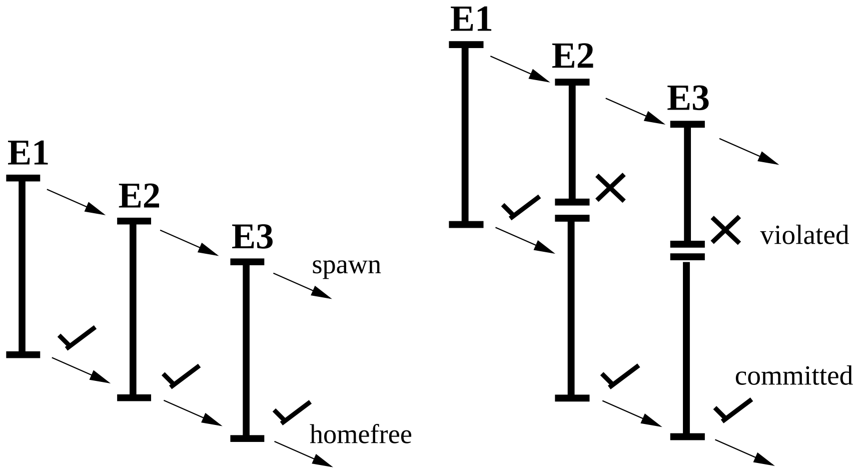
Spawn: A light-weight fork operation that creates and initializes a new speculative thread.

Speculative Region: A single portion of a program where TLS (speculative parallelism) is exploited.

Speculative Thread: A light-weight thread that is used to exploit speculative parallelism within an OS-level thread.

Violation: A thread has suffered a true data dependence violation if it has read a memory location that was later modified by a logically-earlier epoch—other types of violations are described later.

Figure 2.1. Glossary of terms.



(a) Steady-state execution.

(b) Recovering from a violation.

Figure 2.2. TLS execution model.

centralized work queue.

A key component of any architecture for TLS is a mechanism for tracking the relative ordering of the epochs. In our approach, we timestamp each epoch with an *epoch number* to indicate its ordering within the original sequential execution of the program. We say that *epoch X* is *"logically-earlier"* than *epoch Y* if their epoch numbers indicate that *epoch X* should have preceded *epoch Y* in the original sequential execution. Epochs commit speculative results in the original sequential order by passing a *homefree token* which indicates that all previous speculative threads have made all of their speculative modifications visible to the memory system and hence it is safe to commit. When an epoch is guaranteed not to have violated any data dependences with logically-earlier epochs and can therefore commit all of its speculative modifications, we say that the epoch is *homefree*.

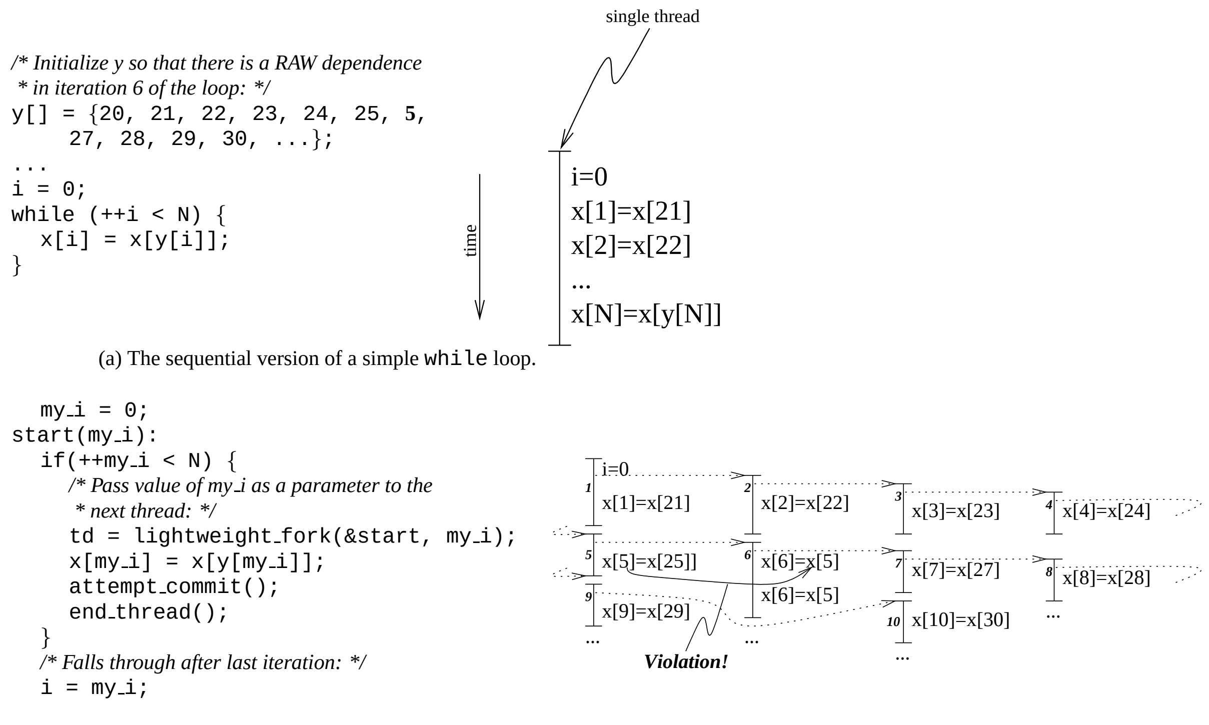
In the case when speculation fails for a given epoch, all logically-later epochs that are currently running are also violated and squashed. In Figure 2.2(b), speculation fails for epoch 2 which in turn causes epoch 3 to be squashed. Although more aggressive strategies are possible, this conservative approach ensures that an epoch does not continue to execute when it may have received incorrect data.

2.3 Software Interface

We have architected our TLS system to involve both the compiler and hardware, hence we require an interface between them. There are a number of issues to consider for such an interface: some issues are analogous to those for purely parallel applications, such as creating threads and managing the stacks; others are unique to TLS, such as passing the *homefree* token and recovering from failed speculation. In this section, we begin with a description of the important components of the software interface to TLS hardware, and then we present the new instructions that implement this interface.

2.3.1 Required Interface Mechanisms

For thread-level speculation, there are many possible implementations for the interface between hardware and software. In this section we briefly explore the design space of interfaces and also present our approach.



```
/* Initialize y so that there is a RAW dependence
 * in iteration 6 of the loop: */
y[] = {20, 21, 22, 23, 24, 25, 5,
    27, 28, 29, 30, ...};
...
i = 0;
while (++i < N) {
  x[i] = x[y[i]];
}
```

(a) The sequential version of a simple `while` loop.

```
  my_i = 0;
start(my_i):
  if(++my_i < N) {
    /* Pass value of my_i as a parameter to the
     * next thread: */
    td = lightweight_fork(&start, my_i);
    x[my_i] = x[y[my_i]];
    attempt_commit();
    end_thread();
  }
  /* Falls through after last iteration: */
  i = my_i;
```

(b) The speculative version.

Figure 2.3. Speculative parallelization of a simple loop.

Threads

Before we discuss issues which are unique to speculation, we first consider one of the requirements for any kind of parallel execution: the creation of parallel threads. Figure 2.3(a) shows an example of a simple `while` loop where the elements of the array `x` are updated using array `y` as an index into `x`. The array `y` has been initialized such that the sixth iteration of the loop (i.e. when $\mathtt{i} = 6$) will depend on the output of the preceding iteration (i.e. when $\mathtt{i} = 5$); otherwise, the iterations are independent.

There are two methods for distributing work to threads: a static method, and a dynamic method. With static distribution, all threads are initialized prior to speculative execution—for example, by specifying a round-robin assignment of epochs for the next speculative region. With dynamic creation, each speculative thread initializes the next; this approach is more tolerant of load imbalance between epochs than the static approach. The dynamic approach is also advantageous when the number of available processors is constantly changing, (e.g. in a multiprogramming environment): the dynamic approach can more easily adapt to this situation by growing or shrinking the number of threads to match the available resources.

Figure 2.3(b) illustrates the dynamic model through a simple loop. Each epoch creates the next through a lightweight

```
i = 0;
do {
  x[i] = x[y[i]];
  i++;
} while (i < x[i]);
```

(a) A simple `do-while` loop with unknown bounds.

```
  my_i = 0;
start(my_i):
  td = lightweight_fork(&start, my_i + 1);
  x[my_i] = x[y[my_i]];
  if(cancelled()) {
    /* Cascade the cancel to subsequent threads: */
    cancel_thread(td);
    end_thread();
  }
  attempt_commit();
  my_i++;
  if(my_i < x[my_i]) {
    end_thread();
  }
  /* Cancel extra loop iterations and continue: */
  cancel_thread(td);
  i = my_i;
```

(b) Speculative version with cancelling.

Figure 2.4. Interface for cancelling superfluous epochs.

fork operation, performs its speculative work, and then attempts to commit its speculative modifications to memory. The lightweight fork returns a thread descriptor (`td`) which serves as a handle on the next thread.

Stacks

A key design issue is the management of references to the stack. A naive implementation would maintain a single stack pointer (shared among all epochs) and a stack in memory that is kept consistent by the underlying data dependence tracking hardware. The problem with this approach is that speculation would fail frequently and unnecessarily: for example, whenever multiple epochs store values to the same location on the stack. These dependence violations would effectively serialize execution. In addition, whenever the stack pointer is modified, the new value must be forwarded to all successive epochs. An alternative approach is to create a separate *stacklet* [30] for each epoch to hold local variables, spilled register values, return addresses, and other procedure linkage information. These stacklets are created at the beginning of the program, assigned to each of the participating processors, and re-used by the dynamic threads. The stacklet approach allows each epoch to perform stack operations independently, allowing speculation to proceed unhindered.

Epoch Cancellation

To speculatively parallelize certain code structures we require support for *control speculation*. For example, a `while` loop with an unknown termination condition can be speculatively parallelized, but superfluous epochs beyond the correct termination of the loop may be created. For correct execution we require the ability to cancel any such

```
i = 0;
while (++i < N) {
  x[i] = x[y[i]];
}
```

(a) Original code.

```
  my_i = 0;
start(my_i):
  if(++my_i < N) {
    td = lightweight_fork(&start, my_i);
    x[my_i] = x[y[my_i]];
    wait_for_homefree_token();
    commit_speculative_writes();
    pass_homefree_token(td);
    end_thread();
  }
  /* Falls through after last iteration: */
  i = my_i;
```

(b) Threaded code.

Figure 2.5. Interface for passing the homefree token.

superfluous epochs. Figure 2.4(a) shows a `while` loop with an unknown ending condition, and Figure 2.4(b) shows the speculatively-parallelized version of the loop with support for cancelling superfluous epochs. Each speculative context provides a flag that indicates whether the current epoch has been cancelled. Any epoch that is cancelled also cancels its child epoch if one exists.[2] In contrast to an epoch that suffers a violation, a cancelled epoch is not re-executed by the run-time system.

Homefree Token

We cannot determine whether speculation has succeeded for the current epoch until all previous epochs have made their speculative modifications visible to memory—hence the act of committing speculative modifications to memory must be serialized. Two unattractive options would be 1) for a central entity to maintain the ordering of the active epochs, or 2) to broadcast to all processors any changes in the allocation of epoch numbers. A more efficient approach is to directly pass an explicit token—which we call the *homefree token*—from the logically-earliest epoch to its successor when it commits and makes all of its speculative modifications visible to memory. Receipt of the homefree token indicates that the current epoch has no predecessors, and hence is no longer speculative.

Figure 2.5 illustrates how the `attempt_commit` operation can be implemented by waiting for the homefree token, committing speculative modifications to memory, and then passing the homefree token directly to the next epoch—thereby avoiding the need for central coordination. This homefree token mechanism is simply a form of producer/consumer synchronization and hence can be implemented using normal synchronization primitives. One option is to perform homefree synchronization at the user level through regular shared memory, without differentiating the homefree token from other synchronization objects. Another option is to make the homefree token visible to

[2] Another option is to interrupt and terminate the epoch, rather than poll a flag.

`thread_descriptor spawn(start_address):` Creates a new thread which begins execution at the start address given. A thread descriptor is returned which can be used to refer to the thread in future calls—if no processor is able to enqueue this request for a new thread then a thread descriptor of zero is returned. A speculative context including a *stacklet* is also allocated for the new thread, and the contents of the forwarding frame are copied to the new thread's forwarding frame.

`void end_thread():` Terminates execution of the current thread, frees its resources, frees its stack (unless it was saved), and invalidates any uncommitted speculative modifications.

`error_code cancel_thread(thread_descriptor):` Locates the specified thread and delivers a cancel signal to it. If the thread has registered a cancel handler, the thread asynchronously jumps to it. If not, then the thread is terminated. If the specified thread does not exist then this instruction is treated as a no-op.

`error_code violate_thread(thread_descriptor):` Notifies the specified thread of a violation—if the thread descriptor is null or invalid then this is a no-op.

`sp save_sp():` Returns the current stack pointer and marks it so that the stacklet is not freed when the thread ends.

`void restore_sp(sp):` Frees the current stacklet and sets the stack pointer to the value provided.

Figure 2.6. TLS instructions for thread and stack management.

hardware so that hardware can react immediately to its arrival. For now, our software interface defers this decision and assumes that the underlying run-time system provides the necessary synchronized behavior. We will evaluate these alternatives later in Chapter 3.7.5.

Value Forwarding

There are often variables that are, at compile time, provably dependent between epochs: for example, a local scalar that is both used and defined every epoch. There are two options when such cases arise. First, the compiler could allocate the variable in memory which would then cause dependence violations between all consecutive epochs. Second, the compiler could synchronize and forward that variable between consecutive epochs, avoiding dependence violations. In our approach, the compiler allocates forwarded variables on a special portion of the stack called the *forwarding frame*; the forwarding frame supports the communication of values between epochs, and synchronizes the accesses to these variables. The forwarding frame is defined by a base-address within the stack frame and an offset from that base address; this way, any regular load or store to an address within the predefined forwarding frame address range can be treated appropriately. The address range of the forwarding frame is defined through the software interface at the beginning of every speculative region. Accesses to the forwarding frame are exempt from the data dependence tracking mechanisms of the underlying hardware.

`void set_sequence_number(sequence_number):` Sets the sequence number of the current thread to create a partial ordering in relation to other speculative threads.

`void become_speculative():` Informs the processor that subsequent memory references should be treated as speculative; if the homefree token has already arrived then this is a no-op. If a violation occurs during speculation then the processor discards all speculative modifications, returns to the address of this `become_speculative()` instruction, and restarts execution.

`void wait_for_homefree_token():` Blocks a thread until it receives the homefree token. If the homefree token is already held then this is treated as a no-op.

`void commit_speculative_writes():` This blocking instruction makes buffered speculative modifications visible to all other speculative threads before returning.

`void pass_homefree_token(thread_descriptor):` This instruction passes the homefree token to the specified thread.

Figure 2.7. TLS instructions for speculation management.

`void set_forwarding_frame(forward_struct_address):` Sets the base address of the forwarding frame (within the stack frame) When used in combination with set_forwarding_size, this specifies a portion of the stack to be copied to the child epoch upon a `spawn`, as well as individual locations to be forwarded mid-epoch.

`void set_forwarding_size(size):` Specifies the size of the forwarding frame (as measured from the base address provided by the set_forwarding_frame instruction).

`void wait_for_value(offset):` Causes the current thread to block until it receives a value at the specified offset in the forwarding frame.

`void send_value(thread_descriptor,offset):` Send the scalar value from the specified location within the forwarding frame to the specified thread. The receiving thread will wake up if it is waiting for that value, and will not block should it subsequently attempt to wait for that value.

Figure 2.8. TLS instructions for value forwarding.

2.3.2 TLS Instructions

We now describe the software interface to TLS hardware support. The goal of this design is to provide the necessary interface to hardware while permitting the exploration of implementation alternatives. For this reason we decompose all TLS events to their component parts and assign a new TLS instruction to each part.

Instructions for Managing Threads and Stacks

A TLS program requires the ability to create speculative threads. In our approach, we are primarily concerned with providing concurrency at a very low cost—hence we implement a lightweight fork instruction called a `spawn`. A `spawn` instruction creates a new thread which begins execution at the start address (PC) given, and is initialized through a copy of the current thread's forwarding frame; a thread descriptor for the child thread is returned as a handle. When its speculative work is complete, a thread is terminated by the `end_thread` instruction. Rather than require software to be aware of the number of available speculative contexts (eg., processors), the semantics of the `spawn` primitive are such that it may fail: failure is indicated by a returned thread-descriptor value of zero. Whenever a `spawn` fails, the speculative thread that attempted the `spawn` simply executes the next epoch itself. This method allows speculative threads to grow to consume all of the available processing resources without creating an unmanageable explosion of threads.

The `cancel_thread` and `violate_thread` primitives allow an epoch to cancel another epoch or trigger a violation and recovery for another epoch, facilitating speculation on more than just data values, such as control speculation. There are also instructions for saving and restoring the stack pointer, and returning to the regular stack after using stacklets during the speculative region.

Instructions for Managing Speculation

Within a speculative thread, software must first initialize speculative execution. The `set_sequence_number` instruction allows software to specify an epoch number to hardware. After an epoch is created, it may perform non-speculative memory accesses to initialize itself. Once this phase of execution is complete, the `become_speculative` instruction indicates that future memory references are speculative.

Several instructions manage the homefree token. It is created and passed by the `wait_for_homefree_token` and `pass_homefree_token` primitives, while the `commit_speculative_writes` primitive instructs hardware

```
for(i = 0; i < N; i++) {
  $S_1$;
  x[i] = y[z[i]];
  $S_2$;
}
```

(a) A `for` loop with a possible dependence.

```
  struct forwarding_frame {int i;} ff;
  int my_i = 0;
  set_forwarding_frame(&ff);
  set_forwarding_size(sizeof(struct forwarding_frame));
  ff.i = 0;
start:
  my_i = ff.i;
  if(my_i < N) {
    ff.i = my_i + 1;
    td = spawn(&start);
    set_sequence_number(my_i);
    become_speculative();
    $S_1$;
    x[my_i] = y[z[my_i]];
    $S_2$;
    wait_for_homefree_token();
    commit_speculative_writes();
    pass_homefree_token(td);
    end_thread();
  }
```

(b) Transformed TLS code.

Figure 2.9. Threads in a basic for loop.

to make all speculative modifications visible to the memory system before returning. All three of these instructions could potentially be combined into one instruction, but for now we keep them separate to maximize flexibility.

Instructions for Forwarding Values

There are four primitives that support the forwarding of values between epochs. First are two instructions that are executed at the beginning of a speculative region to define the forwarding frame by giving its base address and size. The other two instructions name a value to wait for or send by specifying an offset into the forwarding frame. The `send_value` primitive specifies the thread descriptor of the target epoch, and the location of the actual value to be sent. These primitives implement *fine-grained* synchronization, since we synchronize on each individual value (rather than waiting before the first use of any forwarded value and sending after the last definition of any forwarded value). This granularity also allows the processor to issue instructions out-of-order with respect to a blocked `wait_for_value` instruction. In particular, the `wait_for_value` instruction is also associated with a destination register so that the corresponding load instruction (which actually loads the value from the forwarding frame) can be blocked while other instructions are issued around it.

2.3.3 Examples

The instructions presented so far are sufficient to generate a broad class of TLS programs. Figure 2.9 shows an example loop that has been speculatively parallelized using the new TLS instructions, and the illustration shows its

```
for(i = 0; i < f(i); i++) {
  $S_1$;
  x[i] = x[y[i]];
  $S_2$;
}
$S_3$;
```

(a) A `for` loop with an unknown ending condition.

```
  struct forwarding_frame {int i;} ff;
  int my_i = 0;
  if(my_i < f(my_i)) {
    set_forwarding_frame(&ff);
    set_forwarding_size(sizeof(struct forwarding_frame));
    ff.i = 0;
start:
    my_i = ff.i;
    ff.i = my_i + 1;
    set_cancel_handler(&cancelled);
    td = spawn(&start);
    set_sequence_number(my_i);
    become_speculative();
    $S_1$;
    x[my_i] = x[y[my_i]];
    $S_2$;
    my_i++;
    loop_test = my_i < f(my_i);
    wait_for_homefree_token();
    commit_speculative_writes();
    if (loop_test) {
      pass_homefree_token(td);
      end_thread();
    } else {
      /* This was the last iteration. */
      if(td != 0) {
        /* Cancel subsequent threads: */
        cancel_epoch(td);
      }
      goto continue;
    }
cancelled:
    /* Cancel any more epochs if they exist
    cancel_thread(td);
    end_thread();
  }
continue:
  $S_3$;
```

(b) Transformed for TLS.

Figure 2.10. Threads in a `for` loop with an unknown ending condition.

execution on a four processor multiprocessor. The variable `i` is updated and copied to the next epoch through the forwarding frame at each `spawn`, and also serves as the epoch number which is set by the `set_sequence_number` primitive. The code is constructed such that the `spawn` instruction may fail and the current speculative thread will continue and execute the next epoch itself. The final speculative thread will branch around the outermost `if` construct, wait to be homefree, and then continue to execute the code following the loop.

As described in Section 2.3.1, the bounds of a loop may not be known at compile time, and hence a program may erroneously continue to speculate beyond the end of the loop. In this case the extra epochs are cancelled using the `cancel_thread` instruction once the end of the loop is known. Figure 2.10 shows how such a loop with unknown bounds can be speculatively parallelized. In the example, the next iteration of the loop is always spawned, and extra iterations are cancelled when the end of the loop is found.

```
for(i = 0; i < N; i++) {
  $S_1$;
  x[i] = y[z[i]];
  $S_2$;
}
```

(a) A `for` loop with a possible dependence.

```
  struct forwarding_frame {int i;} ff;
  int my_i = 0;
  if(my_i < f(my_i)) {
    set_forwarding_frame(&ff);
    set_forwarding_size(sizeof(struct forwarding_frame));
    ff.i = 0;
start:
  my_i = ff.i;
  if (my_i < N) {
    ff.i = my_i + 1;
    td = spawn(&start);
    set_sequence_number(my_i);
    become_speculative();
    $S_1$;
    x[my_i] = y[z[my_i]];
    $S_2$;
    wait_for_homefree_token();
    commit_speculative_writes();
    if(td == 0) {
      /* Spawn failed, re-use the thread: */
      goto start;
    } else {
      pass_homefree_token(td);
      end_thread();
    }
  }
  wait_for_homefree_token();
```

(b) Transformed TLS code.

Figure 2.11. A loop that re-uses threads.

Figure 2.11 shows how we can cleverly code the speculative version of a loop so that an existing thread is re-used whenever a `spawn` fails. If the `spawn` returns zero, then the `spawn` failed and the current thread goes on to execute the next epoch itself.

Figure 2.12(a) illustrates two cross-epoch dependences: one that is ambiguous (through `x`) and one that is definite (through `v`). Our TLS interface allows us to speculate on the ambiguous dependence while directly satisfying the definite dependence through value forwarding. As shown in Figure 2.12(b), the array `x` is modified speculatively while `v` is synchronized using the TLS instructions for forwarding values. Since both `i` and `v` are stored on the forwarding frame, they have offsets of zero and one respectively. The entire forwarding frame is copied when each epoch is spawned, initializing each epoch with the proper value of `i`.

After speculatively updating `x`, each epoch must synchronize and update `v`. The `wait_for_value` primitive stalls the execution of all loads from the forwarding frame at the offset specified, and the pipeline logic also stalls any further indirectly-dependent instructions. Once the value of `v` is produced by the previous thread, the `wait_for_value` instruction unblocks and the value of `v` is loaded from the forwarding frame. The variable `v` is then updated, stored

```
for(i = 0; i < N; i++) {
  $S_1$;
  x[i] = y[z[i]];
  $S_2$;
  v += z[i];
  $S_3$;
}
```

(a) A `for` loop with both definite and ambiguous dependences.

```
  struct forwarding_frame {
    int i; /* offset 0 */
    int v; /* offset sizeof(int) */
  } ff;
  int my_i = 0;
  int my_v = 0;
  set_forwarding_frame(&ff);
  set_forwarding_size(sizeof(struct forwarding_frame));
  ff.i = 0;
  ff.v = 0;
start:
  my_i = ff.i;
  if (my_i < N) {
    ff.i = my_i + 1;
    td = spawn(&start);
    set_sequence_number(my_i);
    become_speculative();
    $S_1$;
    x[my_i] = y[z[my_i]];
    $S_2$;
    wait_for_value(sizeof(int));
    my_v = ff.v;
    my_v += z[i];
    ff.v = my_v;
    send_value(td,sizeof(int));
    $S_3$;
    wait_for_homefree_token();
    commit_speculative_writes();
    pass_homefree_token(td);
    end_thread();
  }
  wait_for_homefree_token();
```

(b) Transformed TLS code.

Figure 2.12. A loop requiring forwarding.

back to the forwarding frame, and then the next epoch is sent the updated value. The send_value instruction must not be issued out-of-order with respect to the updated definition of the variable v, hence it references v as a parameter to create a dependence.

2.4 Compiler Support

In contrast with hardware-only approaches to TLS, we rely on the compiler to define where and how to speculate. Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [67] which operates on C code. For Fortran applications, the source files are first converted to C using sf2c, and then converted to SUIF format. Our infrastructure performs the following phases when compiling an application to exploit TLS.

Deciding Where to Speculate

One of the most important tasks in a thread-speculative system is deciding which portions of code to speculatively parallelize [14]. For the evaluations in this dissertation, the compiler uses profile information to decide which loops in a program to speculatively parallelize—a thorough treatment of this issue is beyond the scope of this dissertation. We limit our focus to loops for two reasons: first, loops comprise a significant portion of execution time (coverage) and hence can impact overall program performance; second, loops are fairly regular and predictable, hence it is straightforward to transform loop iterations into epochs. Investigation of the impact of parallelizing regions other than loops is also beyond the scope of this dissertation.

The following gives a basic description of the loop selection process used to compile benchmark applications. The first step is to measure every loop in every benchmark application by instrumenting the start and end of each potential speculative region (loop) and epoch (iteration). Second, we filter the loops to only consider those that meet the following criteria:

- the coverage (fraction of dynamic execution) is more than 0.1% of execution time;
- there is more than one iteration per invocation (on average);
- the number of instructions per iteration is less than 16000 (on average);
- the total number of instructions per loop invocation is greater than 30 (on average);

- it does not contain a call to `alloca()`, which would interfere with stack management.

The purpose of this initial filtering is to remove from consideration those loops that are unlikely to contribute to improved performance.

In the third step, we unroll each loop by factors of 1 (no unrolling), 2, 4, and 8, generating several versions of each benchmark to measure. Next we measure the expected performance of each loop and unrolling when run speculatively in parallel using detailed simulation (see Section 2.5.2) on our baseline hardware support for TLS (see Chapter 3), and select loops in two different ways for the purposes of evaluating our hardware support. The first way is to maximize performance, where we select the loops that contribute the greatest performance gain—we will refer to this as the *select* version of the benchmarks. The second way is to maximize coverage, where we greedily select loops that represent the largest fraction of execution time, regardless of performance—we will refer to this as the *max-coverage* version of the benchmarks. In both cases the best performing unrolling factor is used for each loop, and chosen independently for the sequential and speculative versions of each application.

Transforming to Exploit TLS

Once speculative regions are chosen, the compiler inserts the TLS instructions (described in Section 2.3.2) that interact with hardware to create and manage the speculative threads and forward values.

Optimization

Without optimization, execution can be unnecessarily serialized by synchronization (through `wait` and `signal` operations). A pathological case is a “`for`” loop in the `C` language where the loop counter is used at the beginning of the loop and then incremented at the end of the loop—if the loop counter is synchronized and forwarded then the loop will be serialized. However, scheduling can be used to move the `wait` and `signal` closer to each other, thereby reducing this critical path. Our compiler schedules these critical paths by first identifying the computation chain leading to each `signal`, and then using a dataflow analysis which extends the algorithm developed by Knoop [40] to schedule that code in the earliest safe location. We can do even better for any loop induction variable that is a linear function of the loop index; the scheduler hoists the associated code to the top of the epoch and computes that value locally from the loop index, avoiding any extra synchronization altogether. These optimizations have a large impact on performance [76], as we show later in Chapter 5.

Table 2.1. Benchmark descriptions and inputs used.

Benchmark		Description	Input
SPECInt2000	BZIP2	compression	reduced input: `./bzip2.c 1`
	CRAFTY	chess board solver	test input
	GAP	group theory interpreter	test input
	GCC	compiler	test input
	GZIP	compression	test input
	MCF	combinatorial optimization	test input
	PARSER	natural language parsing	test input
	PERLBMK	perl interpreter	subset of test input (avoiding `fork()`)
	TWOLF	place and route for standard cells	test input
	VORTEX	OO database	test input
	VPR	place and route for FPGAs	place portion of test input
SPECfp2000	AMMP	models molecular dynamics	test input
	ART	thermal image recognition with a neural network	test input
	BUK ∗	bucket sort	4MB
	EQUAKE	simulates an earthquake using an unstructured mesh	test input
	MESA	an OpenGL work-alike library	test input
	MGRID	computational fluid dynamics multigrid solver	test input
SPECint95	COMPRESS	compression/decompression	test input
	GCC	compiler	test input
	GO	game playing, AI, plays against itself	test input
	IJPEG	image processing	train input
	LI	lisp interpreter	test input
	M88KSIM	microprocessor simulator	train input
	PERL	perl interpreter	primes.pl from test input
	VORTEX	OO database	test input

Code Generation

Our compiler outputs `C` source code which encodes our new TLS instructions as in-line MIPS assembly code using `gcc`'s "`asm`" statements. This source code is then compiled with `gcc` v2.95.2 using the "`-O3`" flag to produce optimized, fully-functional MIPS binaries containing new TLS instructions.

2.5 Experimental Framework

In this section we describe and analyze the benchmark applications (compiled as described in Section 2.4) and simulation infrastructure that are used throughout this dissertation to evaluate our scheme for TLS.

2.5.1 Benchmarks

We evaluate our support for TLS using all of the SPECint95, SPECint2000, and SPECfp2000 benchmarks [16] with the following exceptions: EON, which is written in C++ and not supported by SUIF; GALGEL, FACEREC, LUCAS, and FMA3D which are written in Fortran90 and not supported by SUIF; and WUPWISE, APPLU, SIXTRACK, SWIM, and APSI for which SUIF compilation fails. Note that we also evaluate the BUK application from the NAS-Parallel

Table 2.2. Truncation of benchmark execution.

Benchmark		Starting Point	Description of Skipped Code	Ended Early?	Instructions Simulated
SPECInt2000	BZIP2	spec.c:306	init	y	334.3M
	CRAFTY	main.c:2022	arg parsing, config parsing, init	y	351.6M
	GAP	gap.c:188	init	y	258.8M
	GCC	toplev.c:3917	argument parsing, setting up register sets	y	1040.9M
	GZIP	gzip.c:293	arg parsing, init	y	446.9M
	MCF	none	(reading input coupled with intense computation)	n	300.7M
	PARSER	main.c:1839	arg parsing, random init	y	599.8M
	PERLBMK	unix_perlmain.c:49	init	n	41.6M
	TWOLF	main.c:89	input file reading	n	276.7M
	VORTEX	bmt0.c:512	command parsing and initialization	y	1007.6M
	VPR	main.c:178	arg parsing, init	y	536.2M
SPECfp2000	AMMP	ammp.c:141	variable init	y	256.7M
	ART	scanner.c:1160	argument parsing, reading input	y	385.6M
	BUK ∗	none	-	n	51.3M
	EQUAKE	quake.c:244	memory init, reading input	n	742.6M
	MESA	mesa4.c:486	argument parsing, GL init, reading mesh	y	1015.7M
	MGRID	mgrid.f:68	setup routine	y	696.5M
SPECInt95	COMPRESS	harness.c:232	generation of input data set	n	3.6M
	GCC	toplev.c:3483	argument parsing, setting up register sets	y	1018.3M
	GO	g2.c:414	init up to file reading	y	1028.5M
	IJPEG	none	(command execution mixed in with argument parsing)	y	637.0M
	LI	xlisp.c:41	init stuff, init file reading and processing	y	747.1M
	M88KSIM	main.c:234	init up to file reading	n	172.5M
	PERL	none	(input script parsing is really part of perl—nothing to skip)	n	12.4M
	VORTEX	bmt0.c:512	command parsing and initialization	y	1008.9M

benchmark suite [6]. A brief description of each benchmark and the input data set used is given in Table 2.1. To maintain reasonable simulation times, we use the *test* input set for most benchmarks; for PERLBMK we have modified the *test* input to avoid calls to `fork()`, and we have reduced the BUK application to its kernel, removing the data set generation and verification code.

To maintain reasonable simulation time, we truncate the execution of each benchmark as described in Table 2.2. For all appropriate benchmarks we also skip the initialization portion of execution and begin simulation with a "warmed-up" memory system, loaded from a pre-saved snapshot. For each benchmark we simulate up to the first billion instructions. Since the sequential and TLS versions of each benchmark are compiled differently, the compiler instruments them to ensure that they terminate at the same point in their executions relative to the source code so that the executions are comparable.

Table 2.3 shows an analysis of the *select* benchmark versions. In two cases, GZIP and PERL, the region selection algorithm has opted to select no regions at all. For the remaining integer benchmarks, coverage (the portion of dynamic

Table 2.3. Statistics for the *select* benchmark versions.

Application Name		Portion of Dynamic Execution Parallelized (Coverage)	Number of Unique Parallelized Regions	Average Epoch Size (dynamic insts)	Average Number of Epochs Per Dynamic Region Instance
SPECInt2000	BZIP2	45.0%	4	1089.2	45876.4
	GCC	18.0%	44	316.1	62.9
	CRAFTY	9.7%	6	5188.3	6.0
	GAP	10.0%	1	374.1	2.6
	GZIP	0.0%	0	0.0	0.0
	MCF	41.0%	6	602.8	552.2
	PARSER	18.0%	11	392.0	456.4
	PERLBMK	17.8%	4	90.2	298.8
	TWOLF	20.6%	6	167.5	10.6
	VORTEX	4.2%	4	2718.1	14.7
	VPR	70.2%	2	182.0	6.4
SPECfp2000	AMMP	68.3%	1	489.9	11.2
	ART	58.4%	7	176.4	2403.8
	BUK ∗	49.7%	2	94.0	8192.0
	EQUAKE	32.3%	4	723.5	2087.6
	MESA	35.4%	3	291.4	41043.0
	MGRID	83.4%	5	3422.3	18.2
SPECint95	GCC	15.6%	35	314.8	61.0
	COMPRESS	39.3%	1	42.0	539.0
	GO	21.2%	21	428.9	41.5
	IJPEG	93.3%	16	1546.5	15.6
	LI	0.5%	1	94.7	50.8
	M88KSIM	61.5%	6	813.8	66.2
	PERL	0.0%	0	0.0	0.0
	VORTEX	4.3%	4	8557.9	14.6

execution of the sequential version that is speculatively parallelized) ranges from 0.5% to 93.3%, and averages 28.8%. Many of these applications spend significant time in regions of code other than loops (for example in recursion); consideration of speculative regions other than loops is beyond the scope of this thesis. For the floating-point benchmarks, which tend to be dominated by loops, coverage ranges from 37.8% to 99.4%. For some benchmarks, good coverage is achieved by selecting several significant loops, such as for EQUAKE which has a coverage of 86.1% through 8 different selected loops. In contrast, VPR has a coverage of 70.2% through only 2 unique loops. The SPECint2000 version of GCC has 44 loops selected, but a coverage of merely 18.0%.

Epoch size is another important characteristic. If epochs are too small, the overheads of speculative parallelization will be overwhelming. If epochs are too large, then they will likely be dependent, or we may not have enough buffer space to hold all of the speculative state. Table 2.3 indicates a wide range of average epoch sizes for the selected regions, from just 42.0 dynamic instructions in COMPRESS to over 8 thousand dynamic instructions in the SPECint95 version of VORTEX. Over all benchmarks, the average number of dynamic instructions per epoch is 768.5, which is quite large. However, the average number of epochs per dynamic region instance can be quite small for some benchmarks: for example, 2.6 for GAP. On average, only 2 or 3 processors will be busy when speculating for GAP

TLS will have a minimal performance impact on applications for which coverage is low. For this reason, we also use the *max-coverage* benchmark versions described in Section 2.4 which contain regions selected to maximize coverage rather than performance. For the integer *max-coverage* benchmarks, coverage ranges from 13.9% to 99.5% and averages 69.9%—a significant increase over the *select* benchmarks. Similarly, coverage for the floating point benchmarks increases to an average of 93.5%. Average epoch size also increases to 16473.5 across all integer benchmarks and 1176.1 across all floating point benchmarks. As we explore more advanced techniques for improving TLS performance, it will be important to know whether techniques can improve the performance of the *max-coverage* benchmarks and hence increase the overall impact of TLS.

Table 2.5 presents a comparison of TLS and sequential versions of the *select* benchmarks. Usually, the size of the executable for the TLS version of an application is larger than that of the sequential version due to the addition of TLS instructions and forwarding frame references—over all of the benchmarks the average increase in size is 6.5%, which is reasonable. In some cases, the TLS version is actually smaller: this is caused by TLS instructions which hinder certain optimizations that can increase code size. Dynamic loads and stores increase by 16.7% and 80.1% respectively,

Table 2.4. Statistics for the *max-coverage* benchmark versions.

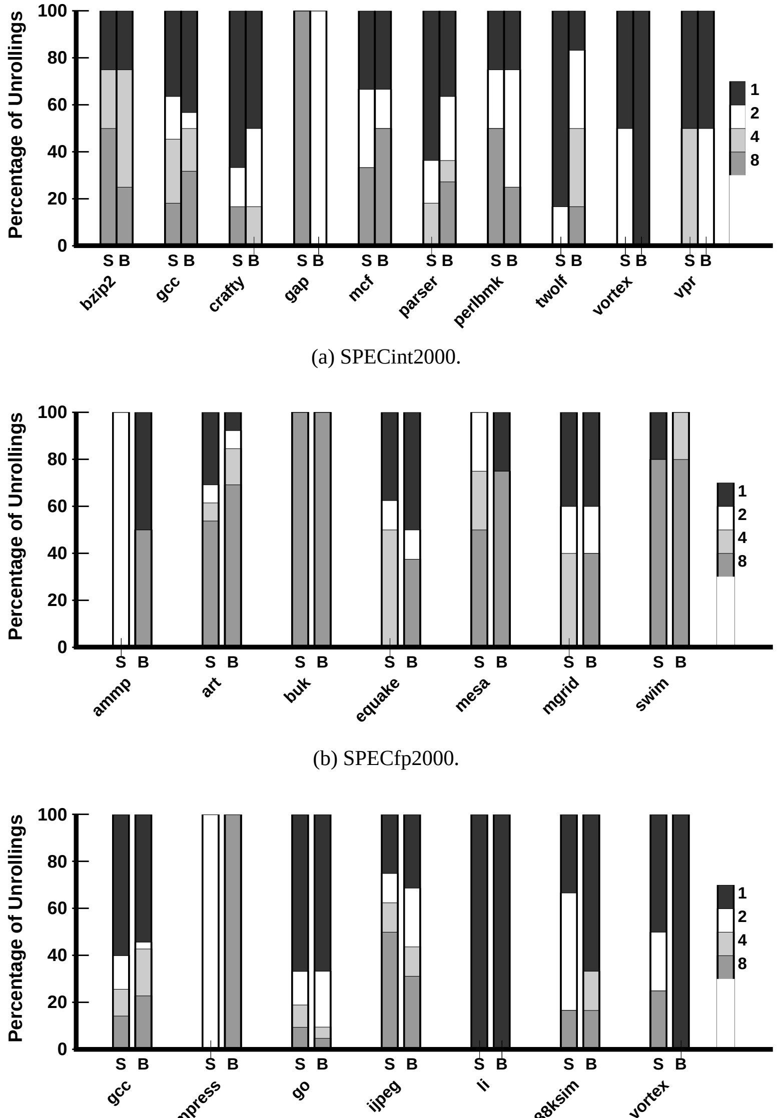
Application Name		Portion of Dynamic Execution Parallelized (Coverage)	Number of Unique Parallelized Regions	Average Epoch Size (dynamic insts)	Average Number of Epochs Per Dynamic Region Instance
SPECInt2000	BZIP2	99.0%	9	761.1	49110.6
	GCC	81.6%	107	1218.8	82.6
	CRAFTY	35.2%	30	389.8	70.6
	GAP	94.6%	5	282981.3	3.3
	GZIP	39.6%	12	193.5	1754.0
	MCF	61.3%	16	1413.9	864.5
	PARSER	91.1%	57	2350.2	173.6
	PERLBMK	32.2%	19	214.8	64.6
	TWOLF	98.1%	24	2422.8	63.3
	VORTEX	13.9%	6	3062.3	11.9
	VPR	99.5%	1	3298.2	5785.5
SPECfp2000	AMMP	79.7%	6	222.3	81.0
	ART	99.9%	18	371.9	2569.4
	BUK *	49.7%	2	94.0	8192.0
	EQUAKE	85.3%	20	494.3	1382.5
	MESA	96.9%	6	468.1	27369.5
	MGRID	99.6%	8	4731.2	30.5
SPECint95	GCC	80.6%	98	1391.7	58.8
	COMPRESS	97.9%	4	673.0	153.6
	GO	93.8%	58	1686.4	16.8
	IJPEG	95.9%	21	2942.1	13.4
	LI	58.6%	1	2587.4	1.0
	M88KSIM	96.0%	3	1851.6	6171.4
	PERL	44.7%	5	498.1	2.6
	VORTEX	14.0%	6	3060.2	11.9

Table 2.5. Comparing TLS and sequential versions of the *select* benchmarks.

Benchmark		Increase in Dynamic Executable Size	Increase in Dynamic Loads	Increase in Dynamic Stores	Fraction Dynamic Forwarding Frame References	Fraction Dynamic TLS Instructions
SPECInt2000	BZIP2	1.4%	4.2%	2.0%	8.8%	0.8%
	GCC	-0.1%	9.3%	22.6%	35.4%	4.2%
	CRAFTY	2.7%	24.0%	194.3%	40.7%	5.6%
	GAP	-0.9%	8.7%	32.4%	36.1%	4.0%
	MCF	31.7%	22.6%	65.3%	36.0%	8.6%
	PARSER	2.2%	36.1%	757.9%	35.3%	2.4%
	PERLBMK	-0.3%	24.0%	314.1%	55.7%	9.4%
	TWOLF	4.2%	24.1%	94.2%	51.2%	5.4%
	VORTEX	-0.8%	2.5%	7.0%	22.3%	1.9%
	VPR	2.5%	13.6%	38.5%	21.7%	4.3%
SPECfp2000	AMMP	3.5%	4.0%	3.6%	19.1%	0.8%
	ART	42.2%	51.1%	194.0%	38.9%	5.5%
	BUK ∗	0.6%	-11.0%	18.5%	25.8%	3.2%
	EQUAKE	35.6%	6.7%	39.4%	8.9%	0.7%
	MESA	3.2%	15.4%	-5.1%	32.7%	9.3%
	MGRID	14.2%	-0.5%	-29.7%	0	0.1%
SPECint95	GCC	2.9%	11.5%	26.2%	34.3%	4.1%
	COMPRESS	10.6%	75.1%	-2.9%	78.5%	5.9%
	GO	0.9%	1.5%	14.4%	20.3%	2.7%
	IJPEG	-7.8%	1.0%	-5.5%	33.7%	2.7%
	LI	-0.2%	25.0%	33.9%	21.0%	5.5%
	M88KSIM	1.6%	20.8%	34.1%	53.2%	3.1%
	VORTEX	-3.0%	2.5%	7.0%	22.3%	1.8%

on average: this increase is significant, and is due partly to the hindrance of optimizations, and partly due to turning register-allocated variables into references to the forwarding frame. For some benchmarks the increase in stores can be quite large: as high as 757.9% for PARSER. A back-end which is designed specifically for TLS would be able to produce code that is significantly more efficient. Within speculative regions across all benchmarks, forwarding frame references account for 30.7% of all memory references, and TLS instructions account for 3.9% of all instructions.

As described in Section 2.4, the region selection algorithm also selects an unrolling for each selected loop. Figure 2.13 visualizes the unrollings selected for the *select* benchmark versions. The floating-point benchmarks favor larger unrollings since they tend to have smaller, more regular loops. Comparing the sequential and TLS versions we see that the unrollings chosen are quite different, and not necessarily larger for either version. For the sequential versions, an average of 40.9% of loops are not unrolled (an unrolling of one), while for the TLS versions an average of 34.5% of loops are not unrolled, indicating that TLS favors unrolling. Figure 2.14 shows whether the unrollings chosen differ between the sequential and TLS versions of each benchmark. Across all of the benchmarks, an average of 57.5% of selected loops have differing unrollings.

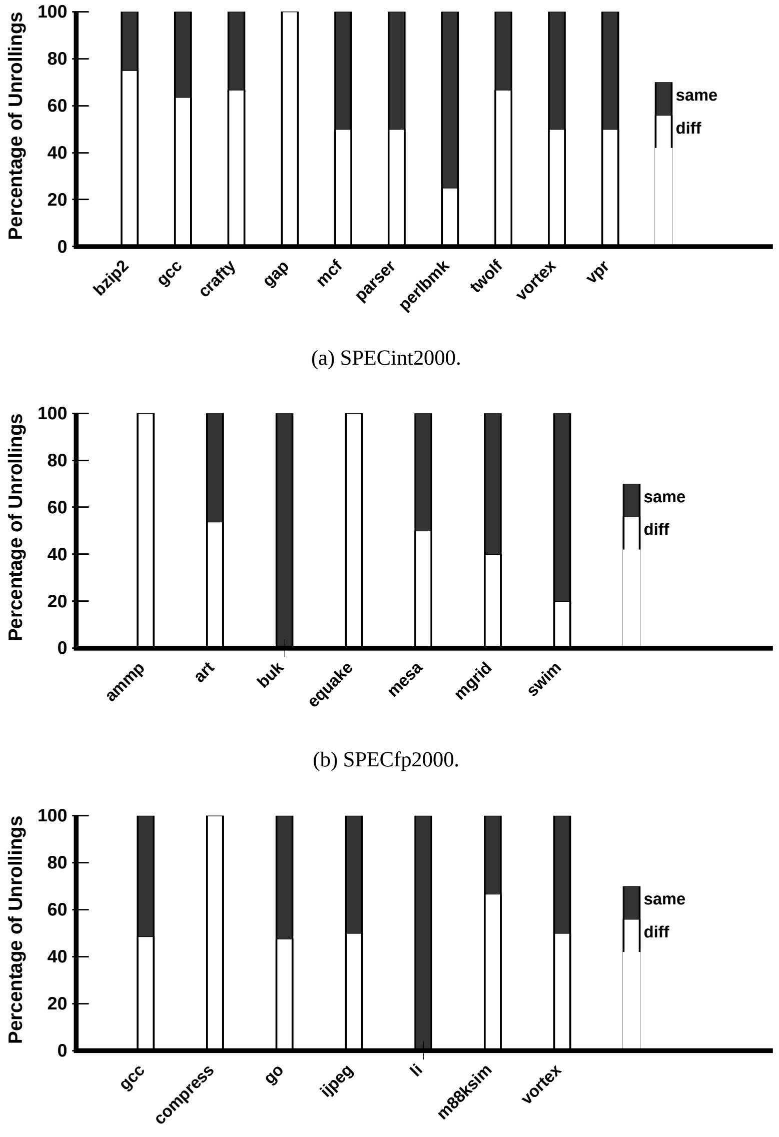


(a) SPECint2000.

(b) SPECfp2000.

(c) SPECint95.

Figure 2.13. Breakdown of unrollings used in the *select* benchmarks. *S* is the sequential version and *B* is the baseline TLS version

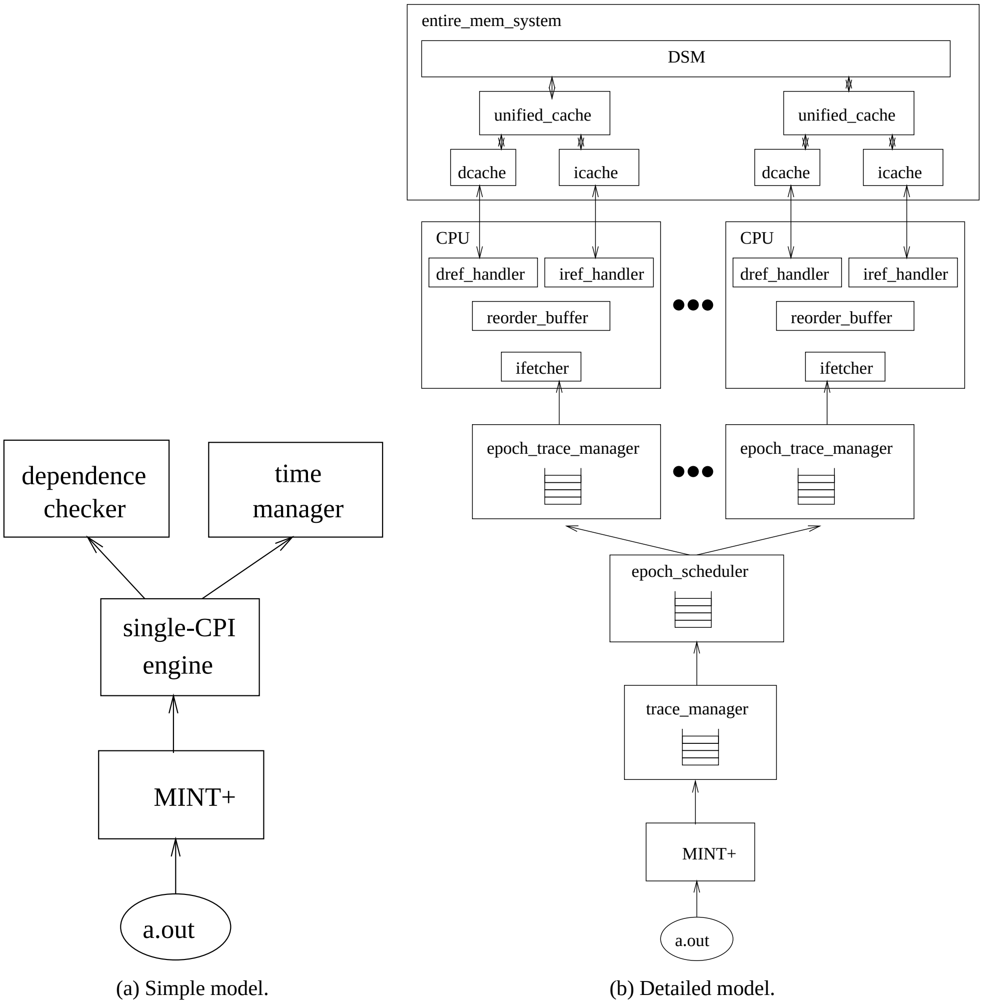


(a) SPECint2000.

(b) SPECfp2000.

(c) SPECint95.

Figure 2.14. Comparing the sequential and TLS versions of the *select* benchmarks, this graph shows the fraction of the unrollings used that are the same or different.



(a) Simple model.

(b) Detailed model.

Figure 2.15. Simulation Models.

For the remainder of the dissertation, we will only analyze a subset of the SPEC benchmarks. We do not study SPECint2000:GZIP or SPECint95:PERL in the *select* version of the benchmarks since there were no regions selected for those benchmarks. Finally, both GCC and VORTEX appear in both SPECint95 and SPECint2000, so we only use the more recent version.

2.5.2 Simulation Models

We evaluate the potential of our approach to TLS using a simple machine model that is built on top of the `MINT+` [72] MIPS emulator, as shown in Figure 2.15(a). It models a four-processor TLS machine where each instruction takes a single cycle to execute (a.k.a. single-CPI), and there is no communication latency between processors. This simple model provides an idealized dependence checker and a basic timing model that is easy to understand and

Table 2.6. Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	128
Integer Multiply	12 cycles
Integer Divide	76 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction	GShare (16KB, 8 history bits)

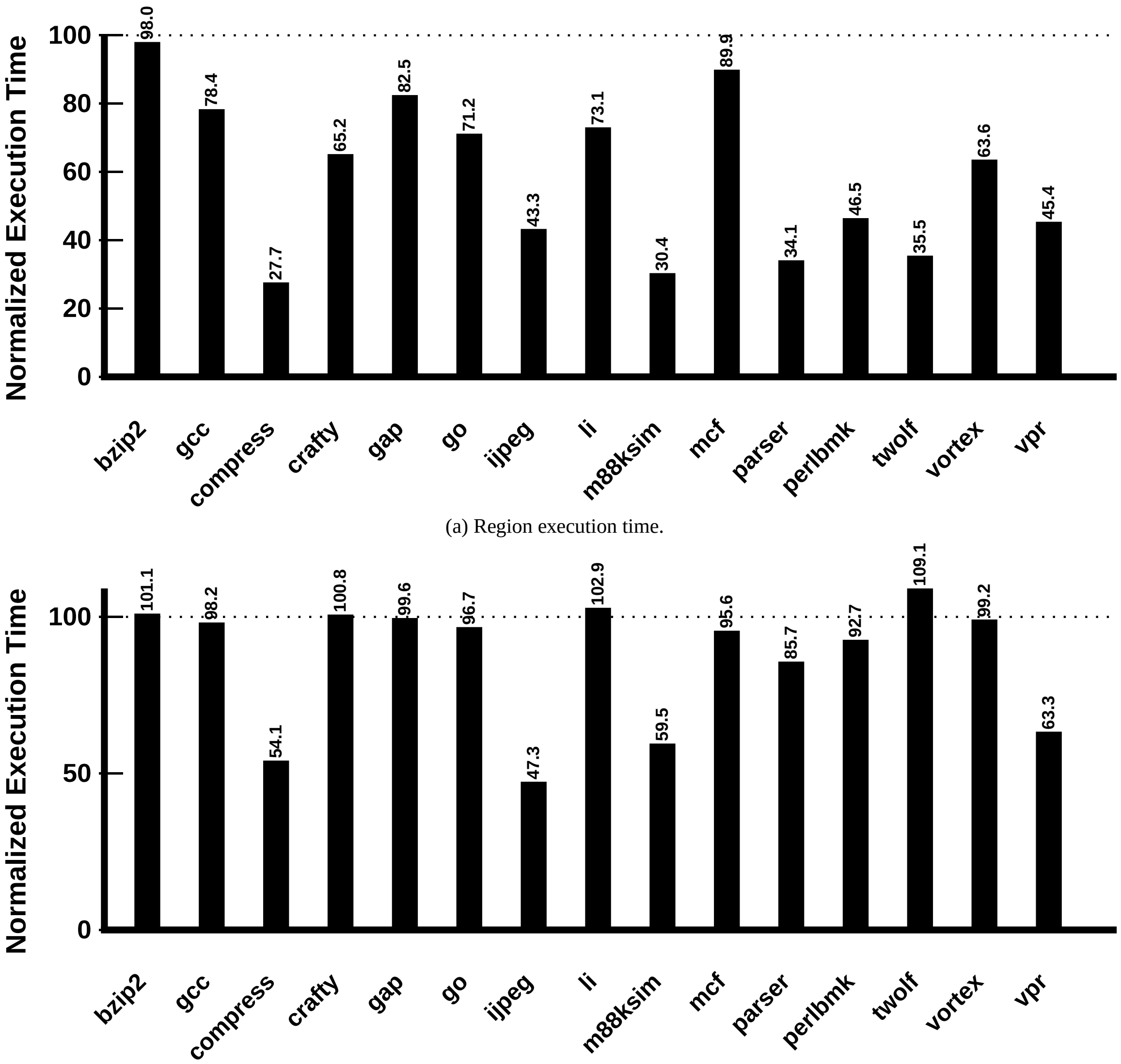
Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 2-way set-assoc, 2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Miss Handlers	16 for data, 2 for insts
Crossbar Interconnect	8B per cycle per bank
Minimum Miss Latency to Secondary Cache	10 cycles
Minimum Miss Latency to Local Memory	75 cycles
Main Memory Bandwidth	1 access per 20 cycles

provides a clear perspective on key issues.

We also evaluate our approach using a detailed model—built upon the `MINT+` [72] MIPS emulator as well—that simulates 4-way issue, out-of-order, superscalar processors similar to the MIPS R10000 [75] but with more modern structure sizes (i.e. a 128 entry reorder buffer, larger caches etc.). As illustrated in Figure 2.15(b), `MINT+` is used to generate a trace of instructions which are then broken into parallel traces and buffered. In this multithreaded simulator, each CPU has an instruction fetcher and pools of memory reference handlers, as well as its own physically private data and instruction caches, connected to a unified second level cache by a crossbar switch. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table 2.6. In the underlying memory system, to avoid the implementation and simulation overhead of time-outs and re-tries, cache coherence is modeled by performing each action globally and atomically, but then modeling the latency and contention of the coherence messages for the action so that it does not complete until after an appropriate delay. The performance of this coherence model is somewhat optimistic in that it avoids certain conflicts between two processors which may lead to occasional failure and re-try of requests in a real multiprocessor, which would incur a greater latency than we model; however, this impact should be small since such events are relatively rare. For some experiments, a distributed shared memory (DSM) network is modeled where each cache line is assigned a home memory module in a round-robin fashion across all nodes in the simulated system.

There are two ways in which this detailed model is inaccurate, since the performance model does not directly control the fetch and execute operation of the underlying `MINT+` emulator.[3] First, we do not have the ability to simulate the

[3] An example of a more recent simulation infrastructure where the performance model directly controls fetch and execute is the ASIM [21] framework and the underlying AINT feeder.



(a) Region execution time.

(b) Program execution time.

Figure 2.16. Improvement in (a) region and (b) program execution time of the TLS version of the *select* benchmarks, according to the simple simulator.

exact mispredicted path—instead, we execute the correct path and flush the pipeline when the mispredicted branch is resolved, then re-execute the correct path again. Second, we do not model "overshoot" for while loops: this is the case when extra epochs beyond the end of a while loop are spawned and later cancelled when they are deemed unnecessary. It is not clear whether simulated TLS performance benefits or suffers from these inaccuracies.

2.6 Potential Performance

Before we delve into the design and implementation of hardware support for TLS in the next chapter, we first want to understand the potential performance benefits of the applications we have speculatively parallelized. Using the simple model described in Section 2.5.2, we model four single-issue processors and exact dependence tracking between speculative threads. In this model, any *read-after-write* dependence through memory between epochs causes the offending epoch and all logically-later epochs to be violated and restarted immediately—other dependences, such as *write-after-write*, do not cause violations. Values that are explicitly synchronized through the forwarding frame mechanism are communicated instantly.

Figure 2.16(a) shows the execution time of the speculative regions of code, each normalized to that of the corresponding sequential version. We observe that excellent region speedups are attainable for most applications. However, considering the program coverage of these regions (which is small for some applications) reported in Table 2.3, the resulting program speedups shown in Figure 2.16(b) are more modest. COMPRESS, IJPEG, M88KSIM, and VPR still obtain significant speedup; MCF, PARSER, and PERL obtain modest speedup; and the remaining benchmarks are unaffected due to low coverage. For LI and TWOLF performance is actually degraded slightly: speculative parallelization has hindered compiler optimization in these cases. The challenge for a real hardware implementation of TLS support will be to approach these theoretical results, and perhaps to surpass them by further improving the efficiency of speculative execution.

2.7 Summary

The purpose of this chapter has been to provide a background on thread-level speculation. We introduced our software interface to TLS (through new instructions) and demonstrated its use. We also discussed the TLS compiler and the benchmark applications to be used throughout this dissertation. Finally, we observed that TLS shows good potential for improving the performance by automatically parallelizing sequential programs.

Chapter 3

Support for Thread-Level Speculation in a Chip-Multiprocessor

The goal of this chapter is to design a unified mechanism for supporting thread-level speculation which can handle arbitrary memory access patterns (i.e. not just array references) and which is appropriate for any scale of architecture with parallel threads, including: simultaneous-multithreaded processors [70], chip-multiprocessors [55, 68], and more traditional shared-memory multiprocessors of any size [47]. We evaluate chip-multiprocessor support in this chapter—other scales are investigated later in Chapter 4.

3.1 Introduction

To support thread-level speculation, we must perform the difficult task of detecting data dependence violations at run-time, which involves comparing load and store addresses that may have occurred out-of-order with respect to the sequential execution. These comparisons are relatively straightforward for *instruction-level* data speculation (i.e. within a single thread), since there are few load and store addresses to compare. For *thread-level* data speculation, however, the task is more complicated since there are many more addresses to compare, and since the relative interleaving of loads and stores from different threads is unknown until run time.

There are three possible ways to track data dependences at run time; for each option, a different entity is responsible for detecting dependence violations. First, a third-party mechanism could observe all memory operations and ensure that they are properly ordered—similar to the approach taken in the Wisconsin Multiscalar's *address resolution buffer* (ARB) [27, 65]. This centralized approach has the drawback of load hit latencies that are greater than one cycle [7] which can hinder the performance of non-speculative workloads.[1]. Second, the producer could detect dependence violations and notify the consumer. This approach requires the producer to be notified of all addresses consumed by logically-later epochs, and for the producer to save all of this information until it completes. On every store, the producer checks if a given address has been consumed by a logically-later epoch and if so notifies that epoch of the dependence violation. This scheme has the drawback that the logically-earliest epoch must perform the detection—but we want the logically-earliest epoch to proceed unhindered!

A third approach is to make consumers responsible for detecting data dependence violations—the producer reports the locations that it has produced to the consumers, and the consumers track which locations have been speculatively consumed. Our key insight is that this behavior is similar to that of an invalidation-based cache coherence scheme: whenever a cache line is modified that has recently been read by another processor, an invalidation message is sent to the cache that has a copy of the line. To extend this behavior to detect data dependence violations, we simply need to track which locations have been *speculatively* loaded, and whenever a logically-earlier epoch modifies the same location (as indicated by an arriving invalidation message), we know that a violation has occurred. Since invalidation-based coherence already works at a variety of scales (chip-multiprocessors, simultaneously-multithreaded processors, and larger-scale machines which use these multithreaded processors as building blocks), an approach that builds on coherence should also work at those scales.

3.1.1 An Example

To illustrate the basic idea behind our scheme, consider an example of how it detects a read-after-write (RAW) dependence violation. Recall that a given speculative load violates a RAW dependence if its memory location is subsequently modified by another epoch such that the store should have preceded the load in the original sequential program. As shown in Figure 3.1, the state of each cache line is augmented to indicate whether the cache line has been speculatively loaded (SL) and/or speculatively modified (SM). Each cache maintains a logical timestamp (*epoch number*) which indicates the sequential ordering of that epoch with respect to all other epochs, and a flag indicating

[1] Subsequent designs of the ARB were more distributed [7]

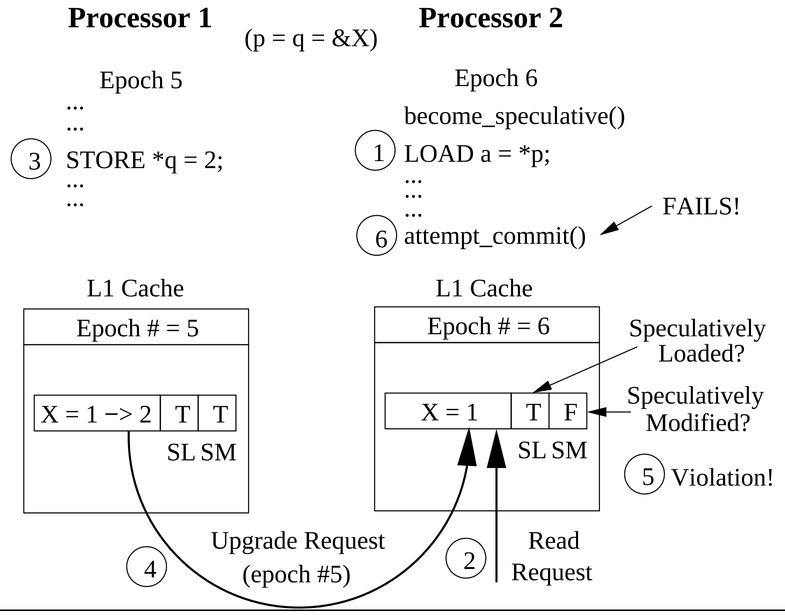


Figure 3.1. Using cache coherence to detect a RAW dependence violation.

whether a data dependence violation has occurred.

In the example, *epoch 6* performs a speculative load, so the corresponding cache line is marked as speculatively loaded. *Epoch 5* then stores to that same cache line, generating an invalidation containing its epoch number. When the invalidation is received, three things must be true for this to be a RAW dependence violation. First, the target cache line of the invalidation must be present in the cache. Second, it must be marked as having been speculatively loaded. Third, the epoch number associated with the invalidation must be from a *logically-earlier* epoch. Since all three conditions are true in the example, a RAW dependence has been violated; and *epoch 6* is notified. The full coherence scheme presented in this chapter handles many other cases, but the overall concept is analogous to this example.

3.1.2 Potential Performance

Before we delve into the details of our design for hardware support for TLS, we first quantify the potential for success of our scheme using the simple model described in Section 2.5.2. Table 3.1 presents some memory access statistics for the *select* benchmarks, to determine whether the first-level data cache can feasibly be used as a speculative buffer. We report the average and maximum number of unique words and unique 32 byte cache lines accessed by each epoch. On average across all benchmarks, each epoch accesses 37.2 unique data words. However, five benchmarks access more than 50 unique words per epoch on average. Hence the performance of these applications would likely be hindered if we were to use only the load and store queues to buffer speculative modifications. Each epoch accesses only

Table 3.1. Memory Access Statistics.

	Unique Accesses per Epoch			
	Word Granularity		Line Granularity	
Benchmark	Average	Maximum	Average	Maximum
BZIP2	20.8	126870.5	8.1	15867.7
GCC	15.7	68.1	8.6	34.5
COMPRESS	127.8	128.0	7.0	17.0
CRAFTY	51.6	99.8	3.4	63.4
GAP	11.9	157.0	8.0	76.0
GO	28.3	110.5	8.4	72.8
IJPEG	40.2	88.9	4.0	29.0
LI	9.2	124.0	7.3	54.0
M88KSIM	15.2	26.8	9.0	17.1
MCF	60.5	64.3	1.8	25.3
PARSER	30.5	119.6	1.5	70.0
PERLBMK	6.3	26.7	4.0	16.7
TWOLF	51.8	94.1	6.0	45.0
VORTEX	56.1	1826.9	9.1	496.4
VPR	29.4	70.4	6.2	35.9

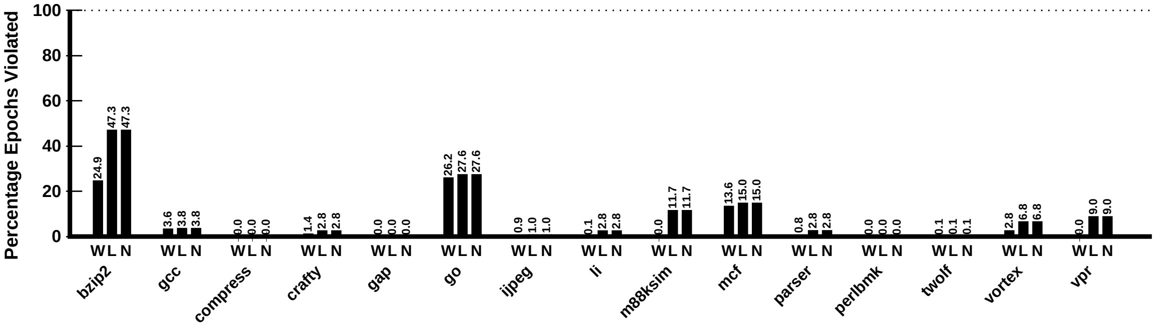
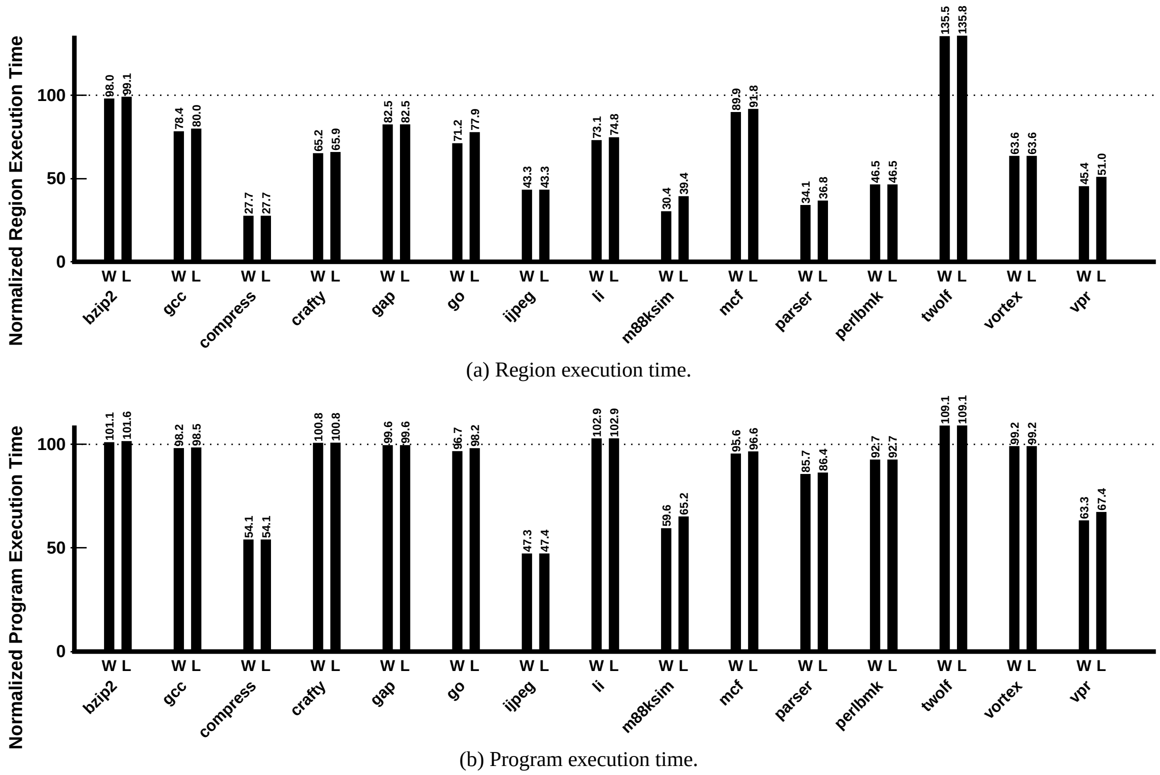


Figure 3.2. Percentage of epochs that are violated. W **tracks true dependences at a word granularity,** L **tracks true dependences at a cache line granularity, and** N **builds on** L **by dissallowing implicit forwarding.**

6.2 unique cache lines on average across all benchmarks, indicating that the data accessed by the average epoch should easily fit in a reasonable-sized first level data cache. For BZIP2 and VORTEX, the maximum number of unique cache lines accessed is quite large, and hence the data accessed by some epochs may not fit entirely in the data cache—our scheme must be able to handle this case.

To estimate the impact of tracking true data dependences more conservatively, Figure 3.2 shows the percentage of epochs that are violated for three increasingly-conservative approaches to dependence tracking: at a word granularity; at a cache line granularity; and at a cache line granularity when dissallowing *implicit forwarding*. Implicit forwarding allows the result value of a speculative store to automatically propagate between properly-ordered epochs, and is generally quite complex to implement for distributed dependence tracking mechanisms: speculative modifications would have to be broadcast to all logically-later epochs, or an epoch would have to poll the caches of all logically-



(a) Region execution time.

(b) Program execution time.

Figure 3.3. Improvement in (a) region and (b) program execution time of the TLS version of the *select* benchmarks, according to the simple simulator—*W* tracks dependences at a word granularity, *L* tracks dependences at a cache line granularity.

earlier epochs for the most up-to-date value on every load. We evaluate the benefits of implicit forwarding for a shared cache implementation later in Section 4.2.1). Many of the regions chosen for the *select* benchmarks suffer little from failed speculation. For those that do, only two applications (BZIP2 and VORTEX) show a significant increase in failed speculation when tracking true data dependences at a cache line granularity. None of the applications show a significant increase in failed speculation when dissallowing implicit forwarding, indicating that such support may not be worthwhile.

We also use the simple simulator described in Section 2.5.2 to estimate the potential improvement in execution time. Figure 3.3(a) shows the potential performance impact of tracking true data dependences at both word and cache line granularities. In most cases, we can greatly improve the performance of the regions of code that are speculatively parallelized, and performance is nearly as good when tracking true data dependences at a cache line granularity. Only BZIP2 and TWOLF perform poorly for this model. Figure 3.3(b) shows the impact on program performance, which is significant for five of the applications and modest for two others. The remaining applications do not speed up under this model, and TWOLF performs significantly worse. While these preliminary results are encouraging, the simple simulator used so far does not model many important aspects of the underlying system—hence the the remainder of this dissertation will only study detailed simulation

3.1.3 Related Work

While there are many approaches to hardware support for TLS [4, 13, 31, 33, 34, 41, 50, 56, 66, 69], there are few that leverage the compiler while attempting to extend generic hardware. The following describes two closely-related approaches.

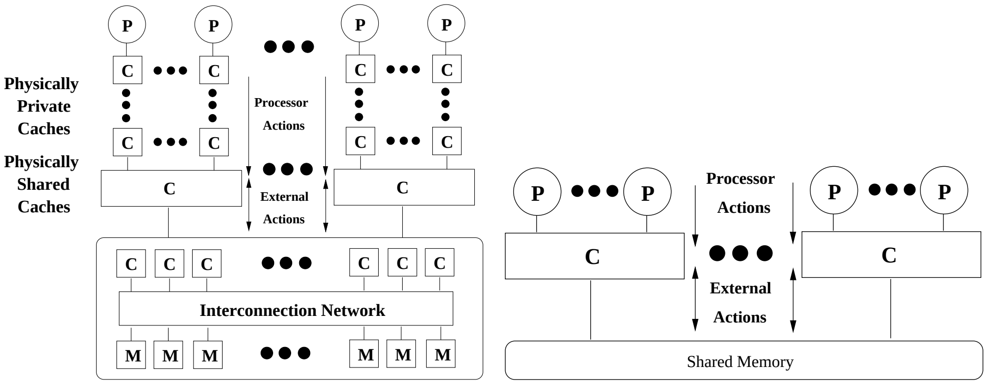
Stanford Hydra

Similar to our approach, the Stanford Hydra architecture [34] adds memory disambiguation support to a general-purpose, chip-multiprocessor (CMP), and uses software to manage threads and speculation. However, there are two important distinctions between Hydra and our approach. First, each processor in a Hydra CMP has a special write-buffer for speculative modifications, while our implementation of TLS uses the first-level caches to buffer speculative state. Second, to ensure that data dependences are preserved, Hydra employs write-through coherence and snoops these write buffers on every store, while our approach uses write-back coherence. While such an approach may be perfectly reasonable within a single chip, it was not designed to scale to larger systems.

I-ACOMA

The Illinois I-ACOMA group has produced three approaches to hardware support for TLS. First, Zhang *et al.* [78] proposed a form of TLS within large-scale NUMA multiprocessors. While this approach can potentially scale up to large machine sizes, it has only been evaluated with matrix-based programs, and its success in handling pointer-based codes has yet to be demonstrated. In addition, it does not appear to be a good choice for small-scale machines (e.g., within a single chip).

Second, Krishnan *et al.* [41] proposed support for TLS within a chip-multiprocessor (CMP). First, a binary annotation pass specifies where to speculate on inner loops, and it also analyzes register dependences. A memory disambiguation table (MDT)—a fully associative list of tags and speculative state—is used to track data dependences, and speculative modifications are stored in the write-back first-level data caches. A "synchronizing scoreboard", a distributed and shared register communication mechanism designed specifically for TLS, is used to synchronize register dependences.



(a) General architecture.

(b) Simplified architecture.

Figure 3.4. Base architecture for the TLS coherence scheme.

Finally, Cintra *et al.* [12] have proposed using a hierarchy of MDTs to support TLS across a NUMA multiprocessor comprised of speculative chip-multiprocessors. While there are many subtle differences between our respective approaches, perhaps the most striking difference is that their hardware enforces a hierarchical ordering of the threads, with one level inside each speculative multiprocessor chip and another level across chips. In contrast, our scheme separates ordering from physical location through explicit software-managed epoch numbers and integrates the tracking of dependence violations directly into cache coherence (which may or may not be implemented hierarchically); hence consecutive epochs may be assigned to non-consecutive processors (although we only evaluate round-robin assignment in this dissertation).

3.1.4 Overview

The remainder of this chapter is organized as follows. First, we describe and evaluate in detail the invalidation-based cache coherence scheme for detecting dependence violations and buffering speculative state. Then, we propose an implementation of this scheme and evaluate its performance and show that it does provide an efficient framework for scalability. We then investigate the possibilities for further tuning our scheme, we evaluate the sensitivity of our approach to various architectural parameters, and compare the performance of alternative implementations for various mechanisms.

3.2 Coherence Scheme For Scalable TLS

3.2.1 Underlying Architecture

The goal of our coherence scheme is to be both general and scalable to any size of machine. We want the coherence mechanism to be applicable to any combination of single-threaded or multithreaded processors within a shared-memory multiprocessor (i.e. not restricted simply to chip-multiprocessors, etc.).

We assume that the shared-memory architecture supports an invalidation-based cache coherence scheme where all hierarchies enforce the inclusion property. Figure 3.4(b) shows a generalization of the underlying architecture. There may be a number of processors or perhaps only a single multithreaded processor, followed by an arbitrary number of levels of physical caching. The level of interest is the first level where invalidation-based cache coherence begins, which we refer to as the *speculation level*. We generalize the levels below the speculation level (i.e. further away from the processors) as an interconnection network providing access to main memory with some arbitrary number of levels of caching.

The amount of detail shown in Figure 3.4(a) is not necessary for the purposes of describing our cache coherence scheme. Instead, Figure 3.4(b) shows a simplified model of the underlying architecture. The speculation level described above happens to be a physically shared cache and is simply referred to from now on as “the cache”. Above the caches, we have some number of processors, and below the caches we have an implementation of cache-coherent shared memory.

Although coherence can be recursive, speculation only occurs at the speculation level. Above the speculation level (i.e. closer to the processors), we maintain speculative state and buffer speculative modifications. Below the speculation level (i.e. further from the processors), we simply propagate speculative coherence actions and enforce inclusion.

3.2.2 Overview of Our Scheme

The remainder of this section provides a summary of the key features of our coherence scheme, which requires the following key elements: (i) a notion of whether a cache line has been speculatively loaded and/or speculatively modified; (ii) a guarantee that a speculative cache line will not be propagated to regular memory, and that speculation will fail if a speculative cache line is replaced; and (iii) an ordering of all speculative memory references (provided by

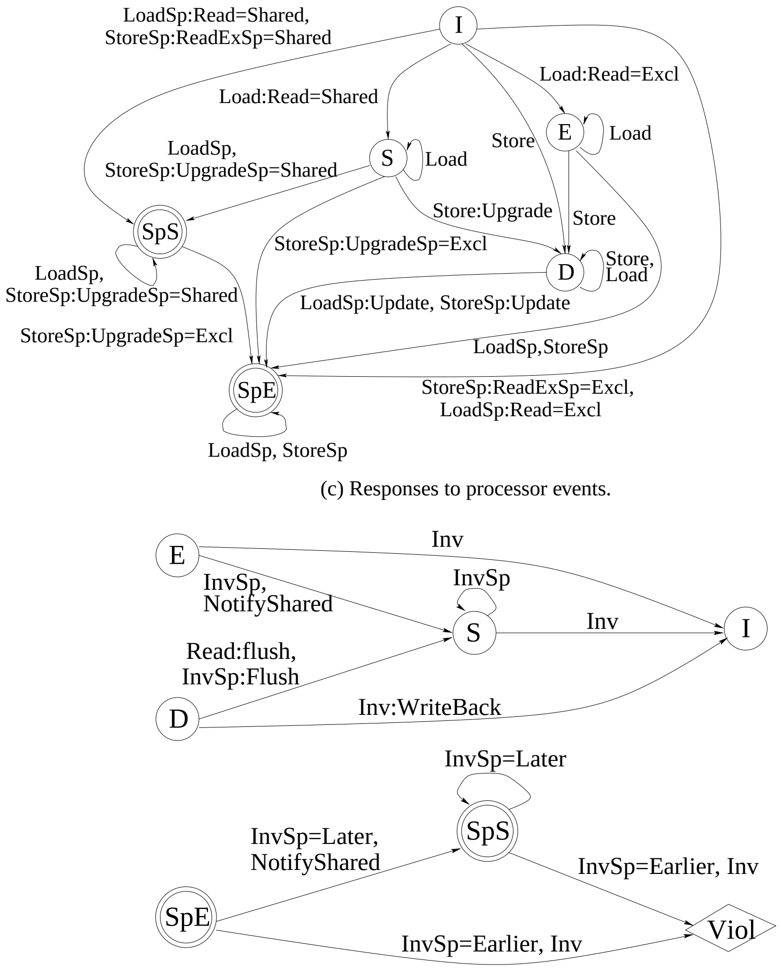
State	Description
I	Invalid
E	Exclusive
S	Shared
D	Dirty
SpE	Speculative (SM and/or SL) and exclusive
SpS	Speculative (SM and/or SL) and shared

(a) Cache line states.

Message	Description
Read	Read a cache line.
ReadEx	Read-exclusive: return a copy of the cache line with exclusive access.
Upgrade	Upgrade-request: gain exclusive access to a cache line that is already present.
Inv	Invalidation.
Writeback	Supply cache line and relinquish ownership.
Update	Supply cache line but maintain ownership.
NotifyShared	Notify that the cache line is now shared.
ReadExSp	Read-exclusive-speculative: return cache line, possibly with exclusive access.
UpgradeSp	Upgrade-request-speculative: request exclusive access to a cache line that is already present.
InvSp	Invalidation-speculative: only invalidate cache line if from a logically-earlier epoch.

Condition	Description
=Shared	The request has returned shared access.
=Excl	The request has returned exclusive access.
=Later	The request is from a logically-later epoch.
=Earlier	The request is from a logically-earlier epoch.

(b) Coherence messages.



(c) Responses to processor events.

(d) Responses to external coherence events.

Figure 3.5. Our coherence scheme for supporting thread-level speculation.

epoch numbers and the *homefree token*). The full details of the coherence scheme are available in the Appendix A.

3.2.3 Cache Line States

A cache line in a standard invalidation-based coherence scheme can be in one of the following states: *invalid* (*I*), *exclusive* (*E*), *shared* (*S*), or *dirty* (*D*). The *invalid* state indicates that the cache line is no longer valid and should not be used. The *shared* state denotes that the cache line is potentially cached in some other cache, while the *exclusive* state indicates that this is the only cached copy. The *dirty* state denotes that the cache line has been modified and must be written back to memory. When a processor attempts to write to a cache line, exclusive access must first be obtained—if the line is not already in the *exclusive* state, invalidations must be sent to all other caches which contain a copy of the line, thereby invalidating these copies.

To detect data dependences and to buffer speculative memory modifications, we extend the standard set of cache line states as shown in Figure 3.5(a). For each cache line, we need to track whether it has been *speculatively loaded* (*SL*) and/or *speculatively modified* (*SM*), in addition to exclusiveness. While our speculative coherence scheme does track exactly this information, to ease its explanation for now we generalize both *speculatively loaded* and *speculatively modified* states as simply *speculative*; i.e., rather than enumerating all possible permutations of *SL*, *SM*, and exclusiveness, we instead summarize by having two speculative states: *speculative-exclusive* (*SpE*) and *speculative-shared* (*SpS*). The full details of the coherence scheme (without a generalization of cache-line states) are available in the Appendix A.

For speculation to succeed, any cache line with a speculative state must remain in the cache until the corresponding epoch becomes *homefree*. Speculative modifications may not be propagated to the rest of the memory hierarchy, and cache lines that have been speculatively loaded must be tracked in order to detect whether data dependence violations have occurred. If a speculative cache line must be replaced, then this is treated as a violation causing speculation to fail and the epoch is re-executed—note that this will affect performance but neither correctness nor forward progress.

3.2.4 Coherence Messages

To support thread-level speculation, we also add the three new speculative coherence messages shown in Figure 3.5(b): *read-exclusive-speculative*, *invalidation-speculative*, and *upgrade-request-speculative*. These new speculative messages behave similarly to their non-speculative counterparts except for two important distinctions. First, the epoch number of the requester is piggybacked along with the messages so that the receiver can determine the logical ordering between the requester and itself. Second, the speculative messages are only hints and do not compel a cache to relinquish its copy of the line (whether or not it does is indicated by an acknowledgment message). Note that it is possible to re-design the speculative coherence scheme so that speculative messages are not used—the performance impact of this potential design is evaluated in Section 3.5.2.

3.2.5 Baseline Coherence Scheme

Our coherence scheme for supporting TLS is summarized by the two state transition diagrams shown in Figures 3.5(c) and 3.5(d). The former shows transitions in response to processor-initiated events (i.e. speculative and non-speculative loads and stores), and the latter shows transitions in response to coherence messages from the external memory system.

Let us first briefly summarize standard invalidation-based cache coherence. If a load suffers a miss, we issue a *read* to the memory system; if a store misses, we issue a *read-exclusive*. If a store hits and the cache line is in the *shared* (*S*) state, we issue an *upgrade-request* to obtain exclusive access. Note that *read-exclusive* and *upgrade-request* messages are only sent *down* into the memory hierarchy by the cache; when the underlying coherence mechanism receives such a message, it generates an *invalidation* message (which only travels *up* to the cache from the memory hierarchy) for each cache containing a copy of the line to enforce exclusiveness. Having summarized standard coherence, the following highlights key features of the extended coherence scheme to support TLS.

Key Features of the Speculative Coherence Scheme

When a speculative memory reference is issued, we transition to the *speculative-exclusive* (*SpE*) or *speculative-shared* (*SpS*) state as appropriate. For a speculative load we set the *SL* flag, and for a speculative store we set the *SM* flag. When a speculative load misses, we issue a normal read to the memory system. In contrast, when a speculative store misses, we issue a *read-exclusive-speculative* containing the current epoch number. When a speculative store hits and the cache line is in the *shared* (*S*) state, we issue an *upgrade-request-speculative* which also contains the current epoch number.

When a cache line has been speculatively loaded (i.e. it is in either the *SpE* or *SpS* state with the *SL* flag set), it is susceptible to a read-after-write (RAW) dependence violation. If a normal *invalidation* arrives for that line, then speculation fails. In contrast, if an *invalidation-speculative* arrives, then a violation only occurs if it is from a *logically-earlier* epoch (as determined by comparing the epoch numbers).

When a cache line is *dirty*, the cache owns the only up-to-date copy of the cache line and must preserve it. When a speculative store accesses a *dirty* cache line, we generate an *update* to ensure that the only up-to-date copy of the cache line is not corrupted with speculative modifications. For simplicity, we also generate an *update* when a speculative load accesses a *dirty* cach line.

Speculative Invalidation of Non-Speculative Cache Lines: A goal of the coherence scheme is to avoid slowing down non-speculative threads to the extent possible—hence a cache line in a non-speculative state is not invalidated when an *invalidation-speculative* arrives from the external memory system. For example, a line in the *shared* (*S*) state remains in that state whenever an *invalidation-speculative* is received. Alternatively, the cache line could be relinquished to give exclusiveness to the speculative thread, possibly eliminating the need for that speculative thread to obtain ownership when it becomes *homefree*. Since the superior choice is unclear without concrete data, we compare the performance of both approaches later in Section 3.5.3.

Dirty and Speculatively Loaded State: As described for the baseline scheme, when a speculative load or store accesses a *dirty* cache line we generate an *update*, ensuring that the only up-to-date copy of a cache line is not corrupted with speculative modifications. Since a speculative load cannot corrupt the cache line, it is safe to delay writing the line back until a speculative store occurs. This minor optimization is supported with the addition of the *dirty and speculatively loaded* state (*DSpL*), which indicates that a cache line is both dirty and speculatively loaded. Since it is trivial to add support for this state, we include it in the baseline scheme.

When Speculation Succeeds

Our scheme depends on ensuring that epochs commit their speculative modifications to memory in logical order. We implement this ordering by waiting for and passing the *homefree token* at the end of each epoch. When the *homefree token* arrives, we know that all *logically-earlier* epochs have completely performed all speculative memory operations, and that any pending incoming coherence messages have been processed—hence memory is consistent. At this point, the epoch is guaranteed not to suffer any further dependence violations with respect to logically-earlier epochs, and therefore can commit its speculative modifications.

Upon receiving the *homefree token*, any line which has only been speculatively loaded immediately makes one of the following state transitions: either from *speculative-exclusive* (*SpE*) to *exclusive* (*E*), or else from *speculative-shared* (*SpS*) to *shared* (*S*). We will describe in the next section how these operations can be implemented efficiently.

Whenever an epoch speculatively-modifies a location, and then a logically-later epoch modifies that same location, both corresponding cache lines will become speculatively-modified (i.e. the *SM* flag is set) and shared, and hence both be in the state *speculative-shared* (*SpS*). When speculation succeeds, for every such cache line we must issue an *upgrade-request* to acquire exclusive ownership. Once it is owned exclusively, the line may transition to the *dirty* (*D*) state—effectively committing the speculative modifications to regular memory. Maintaining the notion of exclusiveness is therefore important since a speculatively modified line that is exclusive (i.e. *SpE* with *SM* set) can commit its results immediately simply by transitioning directly to the *dirty* (*D*) state.

It would obviously take far too long to scan the entire cache for all speculatively modified and shared lines—ultimately this would delay passing the *homefree token* and hurt the performance of our scheme. Instead, we propose that the addresses of such lines be added to an *ownership required buffer* (ORB) whenever a line becomes both speculatively modified and shared. Hence whenever the *homefree token* arrives, we can simply generate an *upgrade-request* for each entry in the ORB, and pass the *homefree token* on to the next epoch once they have all completed.

When Speculation Fails

When speculation fails for a given epoch, any speculatively modified (*SM*) lines must be invalidated, and any speculatively loaded (*SL*) lines make one of the following state transitions: either from *speculative-exclusive* (*SpE*) to *exclusive* (*E*), or else from *speculative-shared* (*SpS*) to *shared* (*S*). In the next section, we will describe how these operations can also be implemented efficiently.

Forwarding Data Between Epochs

As described in Section 2.3.1, we can avoid violations due to predictable data dependences between epochs through the forwarding frame and wait–signal synchronization. As defined by the instruction interface, the forwarding frame and synchronization primitives may be implemented a number of ways, the most simple being through regular memory. Our coherence scheme can be extended to support value forwarding through regular memory by allowing an epoch to make non-speculative memory accesses while it is still speculative. Hence an epoch can perform a non-speculative store whose value will be propagated to the logically-next epoch without causing a dependence violation. We will evaluate several implementations of the forwarding frame and synchronization in Section 3.7.3.

3.3 Implementation

We now describe the implementation of our coherence scheme, beginning with a hardware implementation of epoch numbers. We then give an encoding for cache line states, and describe the organization of epoch state information. Finally, we describe how to allow multiple speculative writers and how to preserve correctness.

3.3.1 Epoch Numbers

In previous sections, we have mentioned that *epoch numbers* are used to determine the relative ordering between epochs. In the coherence scheme, an epoch number is associated with every speculatively-accessed cache line and every speculative coherence action. The implementation of epoch numbers must address several issues. First, epoch numbers must represent a *partial ordering* (rather than total ordering) since epochs from independent programs or even from independent chains of speculation within the same program are unordered with respect to each other. We implement this by having each epoch number consist of two parts: a thread identifier (TID) and a sequence number. If the TIDs from two epoch numbers do not match exactly, then the epochs are *unordered*. If the TIDs do match, then the signed difference between the sequence numbers is computed to determine a logical ordering.

Figure 3.6 defines the principles behind epoch sequence number allocation and comparison. The speculation system must guarantee that there are twice as many consecutive epoch sequence numbers in the sequence number space as there are speculative contexts in the system; given this constraint, the sign of the difference between two non-equal sequence numbers indicates which epoch sequence number is *logically earlier*. Such a comparison is trivial to implement, and should only require a small amount of dedicated hardware. This implementation of sequence numbers is quite similar to the *inums* implemented in the Alpha 21264 [15].

3.3.2 Implementation of Speculative State

We encode the speculative cache line states given in Figure 3.5(a) using five bits as shown in Figure 3.7(a). Three bits are used to encode basic coherence state: *exclusive* (*Ex*), *dirty* (*Di*), and *valid* (*Va*). Two bits—*speculatively*

The following implements a system of epoch sequence numbers such that two properly-allocated, non-equal epoch numbers can be compared even when the sequence numbers wrap around:

Let c be the current epoch sequence number;
Let i be the incoming epoch sequence number;
Let N_{max} be the largest epoch sequence number;
Let N_{sc} be the number of speculative contexts.

A system that properly allocates epoch sequence numbers guarantees that:

1) $N_{max} \geq 2 \times N_{sc} - 1$,
2) $0 \leq c \leq N_{max}$,
3) $0 \leq i \leq N_{max}$,
4) $c \neq i$,
5) $\mid c - i \mid < N_{sc}$.

We can now decide whether i is logically earlier than c by checking if $c - i > 0$, regardless of whether $c < i$ or $i < c$.

Figure 3.6. Comparing Epoch Sequence Numbers.

loaded (*SL*) and *speculatively modified* (*SM*)—differentiate speculative from non-speculative states. Figure 3.7(b) shows the state encoding which is designed to have the following two useful properties. First, when an epoch becomes *homefree*, we can transition from the appropriate speculative to non-speculative states simply by resetting the *SM* and *SL* bits. Second, when a violation occurs, we want to invalidate the cache line if it has been speculatively modified; this can be accomplished by setting its *valid* (*Va*) bit to the AND of its *Va* bit with the complement of its *SM* bit (i.e. $Va = Va \wedge \overline{SM}$).

Figure 3.7(c) illustrates how the speculative state can be arranged. Notice that only a small number of bits are associated with each cache line, and that only one copy of an epoch number is needed per speculative context. The *SL* and *SM* bit columns are implemented such that they can be flash-reset by a single control signal. The *SM* bits are also wired appropriately to their corresponding *Va* bits such that they can be simultaneously invalidated when an epoch is squashed. Also associated with the speculative state are an ownership required buffer (ORB), and the address of the cancel routine.

3.3.3 Preserving Correctness

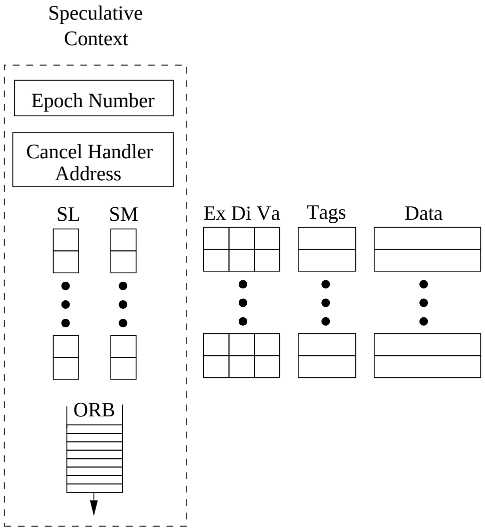
There are several issues—in addition to observing data dependences—that must be addressed to preserve correctness for TLS. First, speculation must fail whenever any speculative state is lost: if the ORB overflows or if a cache

Bit	**Description**
Va	valid
Di	dirty
Ex	exclusive
SL	speculatively loaded
SM	speculatively modified

(a) Cache line state bits.

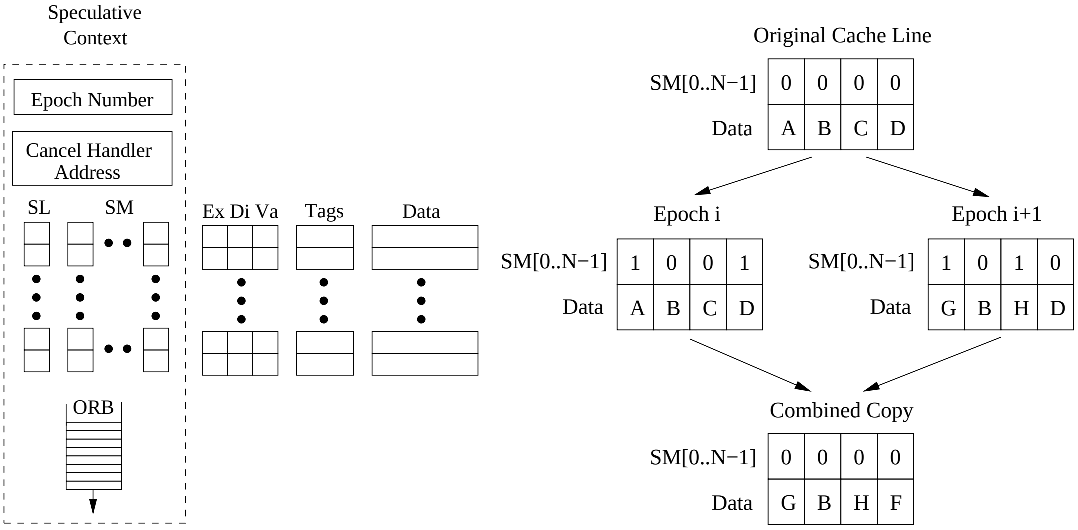
State	SL	SM	Ex	Di	Va
I	X	X	X	X	0
E	0	0	1	0	1
S	0	0	0	0	1
D	0	0	X	1	1
DSpL	1	0	X	1	1
SpE	1	0	1	0	1
	0	1	1	1	1
	1	1	1	1	1
SpS	1	0	0	0	1
	0	1	0	1	1
	1	1	0	1	1

(b) State encoding.



(c) Hardware support.

Figure 3.7. Encoding of cache line states. In (b), X means “don’t-care”.



(a) State for multiple writers.

(b) Combining cache lines.

Figure 3.8. Support for multiple writers.

line in a speculative state must be replaced, then speculation must fail for the corresponding epoch. Note that we do not need special support to choose which cache line to evict from an associative set: the existing LRU (least recently used) policy ensures that any non-speculative cache line is evicted before a speculative one. Second, as with other forms of speculation, a speculative thread should not immediately invoke an exception if it dereferences a bad pointer, divides by zero, etc.; instead, it must wait until it becomes *homefree* to confirm that the exception really should have taken place, and for the exception to be precise. Third, if an epoch relies on polling to detect failed speculation and it contains a loop, a poll must be inserted inside the loop to avoid infinite looping. Finally, system calls generally cannot be performed speculatively without special support; if a thread attempts to perform a system call, we simply stall it until it is *homefree*.

3.3.4 Allowing Multiple Writers

It is often advantageous to allow multiple epochs to speculatively modify the same cache line at the same time. Supporting a multiple writer scheme requires the ability to merge a modified cache line with a previous copy of that line; this in turn requires the ability to identify partial modifications. One possibility is to replicate the *SM* column of bits so that there are as many *SM* columns as there are words (or even bytes) in a cache line, as shown in Figure 3.8(a). We will call these *fine-grain SM* bits. When a write occurs, the appropriate *SM* bit is set. If a write occurs which is of lesser granularity than the *SM* bits can resolve (e.g. a byte-write when there is only one SM bit per 4-byte word), we must conservatively set the *SL* bit for that cache line since we can no longer merge this cache line with others—setting the *SL* bit ensures that a violation is raised if a logically-earlier epoch writes the same cache line.

Figure 3.8(b) shows an example of how we can combine speculatively modified versions of a cache line with a non-speculative one. Two epochs speculatively modify the same cache line simultaneously, setting the *fine-grain SM* bit for each location modified. A speculatively modified cache line is committed by updating the current non-speculative version with only the words for which the *fine-grain SM* bits are set. In the example, both epochs have modified the first location. Since *epoch i+1* is logically-later, its value (G) takes precedence over *epoch i's* value (E).

Because dependence violations are normally tracked at a cache line granularity, another potential performance problem is *false violations*—i.e. where disjoint portions of a line were read and written. To help reduce this problem, we observe that a line only needs to be marked as *speculatively loaded* (*SL*) when an epoch reads a location that it has not previously overwritten (i.e. the load is *exposed* [3]). The *fine-grain SM* bits allow us to distinguish exposed loads, and therefore can help avoid false violations. We evaluate the performance benefits of support for multiple writers in Section 3.5.1, and show that this support has a significant impact.

3.4 Evaluation of Baseline Hardware Support

We now present the performance of our baseline coherence scheme for TLS, and evaluate the overheads of our approach. In this chapter we focus on chip-multiprocessor architectures—later (in Chapter 4) we evaluate larger-scale machines that cross chip boundaries, as well as smaller-scale machines that share the first-level data cache.

3.4.1 Performance of the Baseline Scheme

Table 3.2 summarizes the performance of each application on our baseline architecture (as reported by the detailed simulation model described in Chapter 2.5.2), which is a four-processor chip-multiprocessor that implements our baseline coherence scheme. Throughout this dissertation, all speedups (and other statistics relative to a single processor) are with respect to the *original* executable (i.e. without any TLS instructions or overheads) running on a single processor. Hence our speedups are *absolute speedups* and not self-relative speedups. As we see in Table 3.2, we achieve speedups on the speculatively-parallelized regions of code ranging from 4% to 134%, with the exception of BZIP2 which slows down. The overall program speedups are limited by the *coverage* (i.e. the fraction of the original execution time that was parallelized) which ranges from 1% to 93%. Looking at program performance, IJPEG is more

Table 3.2. Region and Program Speedups and Coverage.

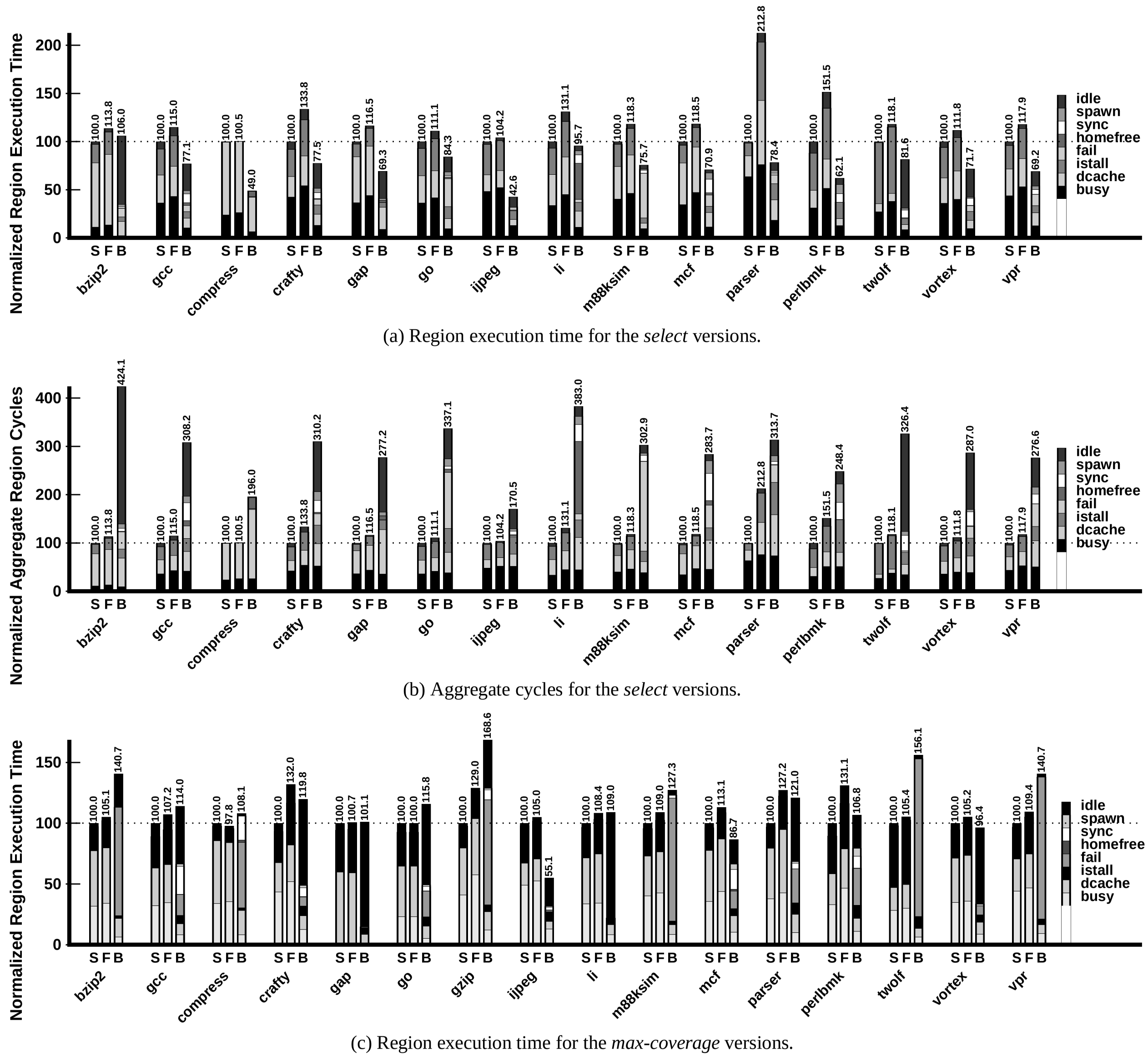
Benchmark	Program Speedup	Region Speedup	Outside-Region Speedup	Coverage
BZIP2	0.96	0.94	0.98	45%
GCC	0.97	1.29	0.91	18%
COMPRESS	1.21	2.04	0.96	39%
CRAFTY	0.95	1.28	0.92	9%
GAP	0.93	1.44	0.90	10%
GO	0.90	1.18	0.85	21%
IJPEG	2.01	2.34	0.67	93%
LI	0.95	1.04	0.95	01%
M88KSIM	1.06	1.32	0.81	61%
MCF	1.10	1.40	0.96	40%
PARSER	1.02	1.27	0.97	19%
PERLBMK	1.03	1.61	0.96	17%
TWOLF	0.86	1.22	0.79	21%
VORTEX	1.02	1.39	1.01	4%
VPR	1.13	1.44	0.74	70%

than twice as fast, and three other applications improve by at least 10%. Four other applications show more modest improvement, while the remaining applications perform slightly worse. To simplify our discussion, we will focus only on the speculatively parallelized regions of code throughout the remainder of this chapter.

Figure 3.9(a) shows execution time normalized to that of the original sequential version for the *select* benchmark versions; each of the bars are broken down into eight segments explaining what happened during all potential graduation slots.[2] The top segment, *idle*, represents slots where the pipeline has nothing to execute—this could be due to either fetch latency, or simply a lack of work. The next three segments represent slots where instructions do not graduate for the following TLS-related reasons: waiting to begin a new epoch (*spawn*); waiting for synchronization for a forwarded value (*sync*); and waiting for the homefree token to arrive (*homefree*). The *fail* segment represents all slots wasted on failed speculation, including slots where misspeculated instructions graduated. The remaining segments represent regular execution: the *busy* segment is the number of slots where instructions graduate; the *dcache* segment is the number of non-graduating slots attributed to data cache misses; and the *istall* segment is all other slots where instructions do not graduate. Figure 3.9(b) shows *aggregate cycles*, which is simply the normalized number of cycles multiplied by the number of processors; it is somewhat easier to directly compare segments in this view where an increase in the size of a segment means that a problem is becoming worse.

The first bar (*S*) shows the breakdown for the original sequential version of each benchmark. Some are dominated

[2] The number of graduation slots is the product of (i) the issue width (4 in this case), (ii) the number of cycles, and (iii) the number of processors.



(a) Region execution time for the *select* versions.

(b) Aggregate cycles for the *select* versions.

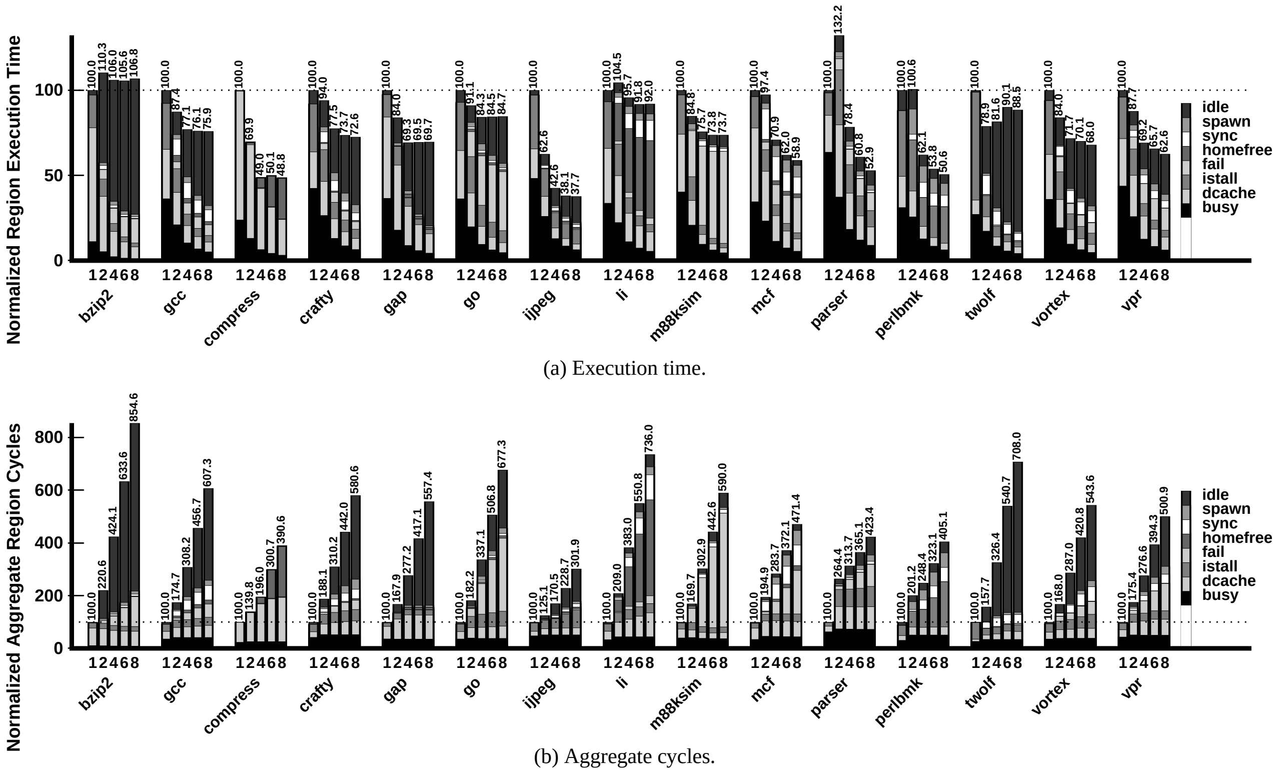
(c) Region execution time for the *max-coverage* versions.

Figure 3.9. Impact on region execution time of our baseline hardware support for TLS. S is the original sequential version, F is the speculative version run sequentially, and B is the speculative version run in parallel on four processors.

by data cache miss time (*dcache*), while others are dominated by pipeline stalls (*istall*). Very little time is lost due to an empty pipeline (*idle*) for any application. The next bar (*F*) shows the performance of the TLS version of each benchmark when executed on a single processor (in this model, all `spawn`'s simply fail, resulting in a sequential execution); this experiment shows the overheads of TLS compilation that must be overcome with parallel execution for performance to improve. The increase in busy time between the *S* and *F* bars is due to the TLS instructions added to the TLS version of each benchmark, as well as other instruction increases due to compilation differences. In most cases, data cache miss time (*dcache*) remains relatively unchanged, while pipeline stalls (*istall*) increase. Some overhead is due to inefficient compilation: inserted TLS instructions are encoded as in-line MIPS assembly code using `gcc`'s "`asm`" statements, and the unmodified version of `gcc` (v2.95.2) that we use as a back-end compiler is conservative in the presence of these statements (e.g. generating superfluous register spills).

The third bar (*B*) shows the TLS version executed speculatively in parallel on four processors. Some benchmarks show a large increase in *idle* time—for the most part this is caused by a small number of epochs per region instance, resulting in idle processors. For example, GAP has an average of only 2.6 epochs per region instance (see Table 2.3), which is not enough parallelism to completely occupy 4 processors. To some extent this is an artifact of our simulation, since a more sophisticated TLS system could increase the number of speculative threads by allowing speculation beyond the end of a loop (i.e., the loop *continuation* could be executed speculatively in parallel with the iterations of the loop). Looking at Figure 3.9(b), we see that data cache and pipeline stalls (*dcache* and *istall*) increase only modestly, while the most significant bottlenecks are failed speculation (*fail*), synchronization (*sync*), and idle time (*idle*). It is important to note that the failed speculation component (*fail*) includes data cache misses, pipeline stalls, synchronization, etc., for all cycles spent on failed speculation—hence when a bottleneck such as synchronization is reduced, the failed speculation component may be reduced as well.

At the beginning and end of each epoch we measure the time spent waiting for an epoch to be spawned (*spawn*, and the time spent waiting for the *homefree* token. Spawn time is not a significant bottleneck for any benchmark, but is most evident in MCF and PERLBMK. Passing the homefree token (*homefree*) is an insignificant portion of execution time for every benchmark except LI, indicating that this aspect of our scheme is not a performance bottleneck for most benchmarks. As we will see later in Section 3.4.2, the large *homefree* segment for LI is not due to a large number of ORB entries, but rather load imbalance: when the size of epochs varies greatly, a small epoch ends up waiting to be



(a) Execution time.

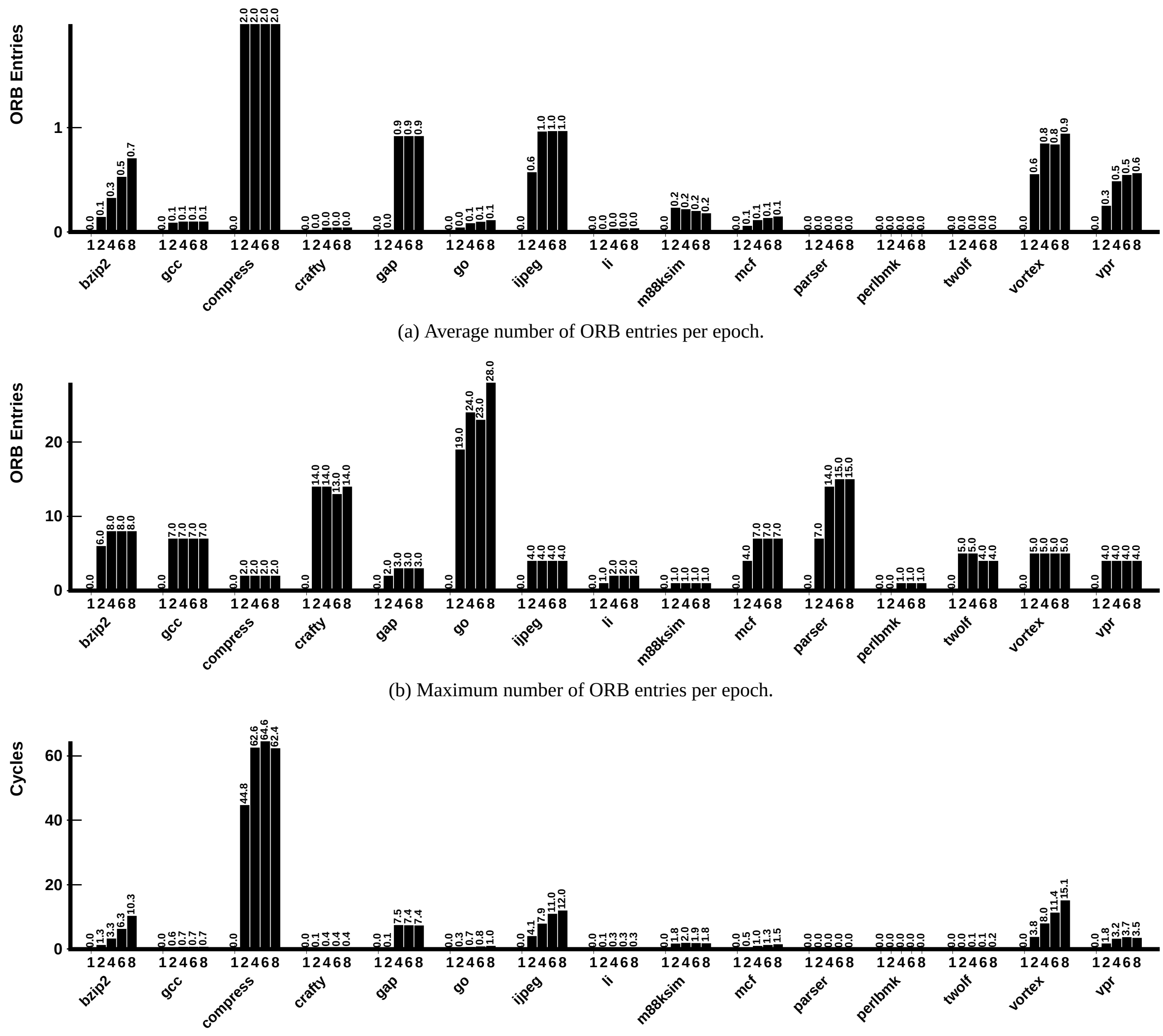
(b) Aggregate cycles.

Figure 3.10. Varying the number of processors from one through eight. The baseline architecture has four processors.

passed the homefree token from a logically-earlier large epoch.

Turning our attention to the *max-coverage* version (Figure 3.9(c)), we observe that many of the speculatively-parallel executions (*B*) do not speed up, with the exception of IJPEG, MCF, and VORTEX. For a majority of benchmarks, the bottleneck is failed speculation (*fail*); this is understandable, since regions for these versions have been chosen to maximize coverage rather than performance, which in turn leads to larger epochs and a smaller probability of speculation succeeding. Techniques that increase the independence of epochs and decrease failed speculation will have a large impact on this version of the benchmarks (as we will see later in Chapter 5).

Figure 3.10 shows how performance for the *select* benchmarks varies across a number of different processors from two different perspectives: normalized execution time, and aggregate cycles (similar to Figure 3.9(b)). The bars in Figure 3.10(b) would remain at the 100% line if we achieved linear speedup; in reality, they increase as the processors become less efficient. As we increase the number of processors, performance continues to improve through 8 processors for six cases; for seven other cases, performance levels-off prior to 8 processors, indicating a limit to the amount of available parallelism. For GAP and TWOLF, performance actually degrades as we increase the number of



(a) Average number of ORB entries per epoch.

(b) Maximum number of ORB entries per epoch.

(c) Average ORB flush latency.

Figure 3.11. Size and flush latency of the *ownership required buffer* (ORB), as we vary the number of processors.

processors due to increasing idle time, indicating that additional processors are under-utilized. The amount of time spent waiting for the *homefree* token increases for both COMPRESS and LI, not because of an increasing number of ORB entries, but because of increasing load imbalance. For several applications, the amount of failed speculation increases, indicating a limit to the independence of epochs.

3.4.2 Overheads of Thread-Level Speculation

We now investigate the overheads of our baseline scheme in greater detail. The most significant overheads are the ORB and passing the homefree token, decreased cache locality, and failed speculation. Recall that the ORB maintains a list of addresses of speculatively-modified cache lines that are in the *speculative-shared* (*SpS*) state. When the *homefree token* arrives, we must issue and complete upgrade requests to obtain exclusive ownership of these lines (thereby committing their results to memory) prior to passing the homefree token on to the next *logically-later* epoch. In addition, speculation fails if the ORB overflows. For these reasons, we desire the average number of ORB entries per epoch to be small. Figure 3.11 shows several measurements of the ORB mechanism (see Section 3.2.5) for a varying number of processors. First, in Figure 3.11(a) we see that the average number of ORB entries per epoch is less than one for all applications except COMPRESS for which it is roughly two entries. Note that the number of entries for COMPRESS and LI does not increase as the number of processors increases, indicating that the increase in time spent waiting for the *homefree* token observed in Figure 3.10 for those benchmarks is indeed due to load imbalance.

Figure 3.11(b) shows the maximum number of ORB entries per epoch which is less than ten for most applications, and between 20 and 30 entries for GO and VORTEX. Note also that for these two benchmarks the maximum number of ORB entries per epoch increases as the number of processors increases, indicating that there are shared cache lines for which the number of sharers increases with the number of processors; in support of this claim, we look back at Figure 3.10 and observe that both GO and VORTEX do suffer an increasing amount of failed speculation as the number of processors increases. From these results we can conclude that an ORB of 10 entries is sufficient for 11 of the 15 benchmarks, and an ORB of 20 entries would capture the requests for 2 additional benchmarks.

Figure 3.11(c) shows the latency in cycles of flushing the ORB. We observe that this latency is related to the average size of the ORB. For most applications, this latency is negligible; for four applications it is less than 10 cycles with 4 processors; and it is sizable for COMPRESS at 62.5 cycles with 4 processors, which also has more than twice the average number of ORB entries per epoch than other applications.

Figure 3.12 shows the percentage of execution for each benchmark lost to failed speculation for a varying number of processors. Each bar is broken down, showing the fraction of speculation that failed for each of five reasons. The first segment, *CHAIN*, represents time spent on epochs that were squashed because logically-earlier epochs were previously squashed. This *violation chaining* prevents an epoch from using potentially incorrect data that was forwarded from a logically-earlier epoch. The next two segments (*REPL* and *RINV*) represent violations caused by replacement in either the first-level data caches or the shared unified cache respectively. The last two segments (*DINV* and *DINVSP*) represent violations caused by true data dependences. A *DINV* violation occurs when an epoch commits and flushes its

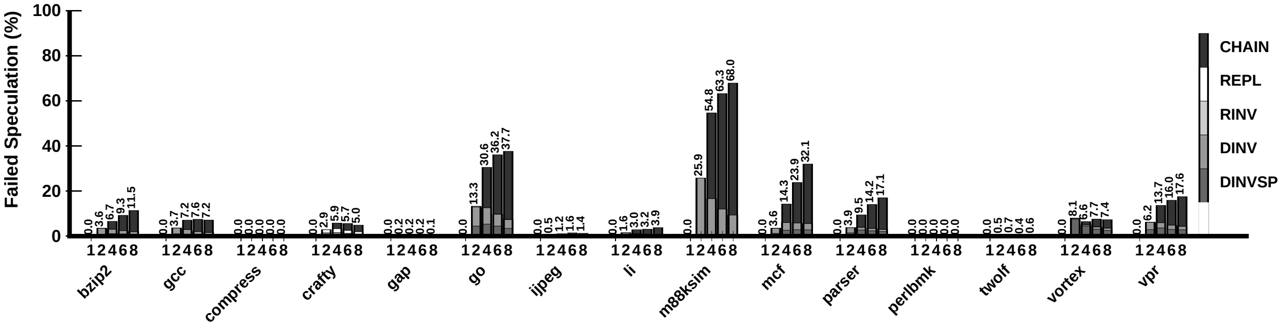


Figure 3.12. Percentage of execution time wasted on failed speculation, and the breakdown of reasons for violations as we vary the number of processors.

ORB, generates a read-exclusive request, and invalidates a speculative cache line belonging to a logically-later epoch. A *DINVSP* violation is caused by a *speculative* invalidation which is sent before an epoch commits.

COMPRESS, GAP, IJPEG, PERLBMK, and TWOLF have an insignificant amount of failed speculation, and for BZIP2, GCC, CRAFTY, and LI the amount of failed speculation is quite small. The amount of failed speculation due to *CHAIN* violations increases with the number of processors; for a given violated epoch, the number of *CHAIN* violations is equal to the number of logically-later epochs currently in-flight. The portion of failed speculation due to *CHAIN* violations is usually greater than the portion due to other reasons, except for the two-processor cases where at any given time only one epoch is speculative and can be violated (and the other is always non-speculative). Replacement in the unified cache (*RINV*) is not significant for any benchmark (since the data sets for these applications fits well within the 2MB unified secondary cache); replacement from the first-level data caches (*REPL*) is evident for CRAFTY, but is not a large component. Speculative invalidations (*DINVSP*) are preferable over ORB-generated invalidations (*DINV*) because they give earlier notification of violations and also help reduce the size of the ORB itself. For five applications, *DINVSP* violations are as frequent as *DINV* violations, while three other applications are dominated by *DINV* violations.

Finally, we estimate the impact of speculative parallelization on data cache locality. Figure 3.13 shows the fraction of cache misses where the cache line in question is currently resident in another first-level cache. A high percentage indicates that cache locality for the corresponding application has been decreased. We see that this percentage is quite high for most applications, the average being 75.0%. Looking again at Figure 3.10(b), we observe that there is an increase in the amount of time spent on data cache misses (*dcache*) for the speculative versions as compared with the

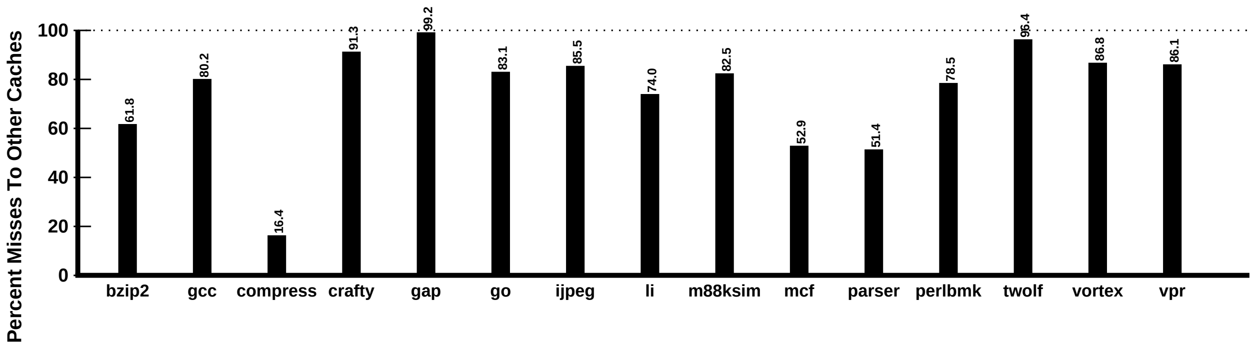


Figure 3.13. Percentage of misses where the cache line is resident in another first-level cache, which indicates the impact of TLS execution on cache locality.

sequential version. This loss of locality, while not prohibitive, is an opportunity for improvement through prefetching and other techniques for dealing with distributed data access.

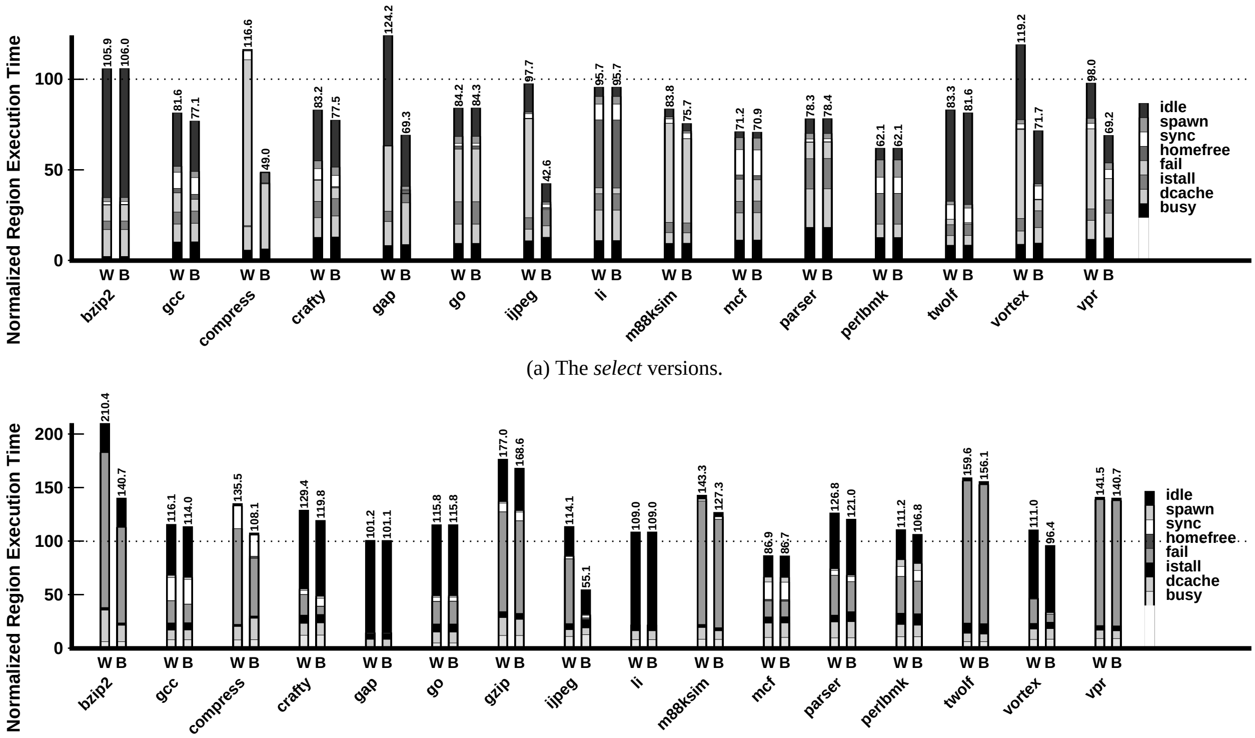
In summary, the overheads of TLS remain small enough that we still enjoy significant performance gains for the speculatively-parallelized regions of code. We now focus on other aspects of our design.

3.5 Tuning the Coherence Scheme

Our performance analysis has shown that TLS is promising, that our scheme for extending cache coherence to track data dependences and buffer speculative state is efficient and effective, and that it scales well within a single chip. In this section we evaluate the performance of several implementation alternatives for various aspects of our coherence scheme including support for multiple writers, support for speculative coherence without speculative messages, and speculative invalidation of non-speculative cache lines.

3.5.1 Support for Multiple Writers

In this section we evaluate the benefits of support for multiple writers, as described earlier in Section 3.3.4. Recall that multiple writer support allows us to avoid both violations due to write-after-write dependences as well as violations due to false dependences where speculative loads are not *exposed*. Figure 3.14 compares the performance of our baseline hardware (*B*) which does support multiple writers, with that of less complex hardware (*W*) that does not support multiple writers. For many applications, support for multiple writers drastically reduces the amount of failed speculation; five *select* applications only speed up with this support enabled. Support for multiple writers shows significant impact for the *max-coverage* benchmarks as well.



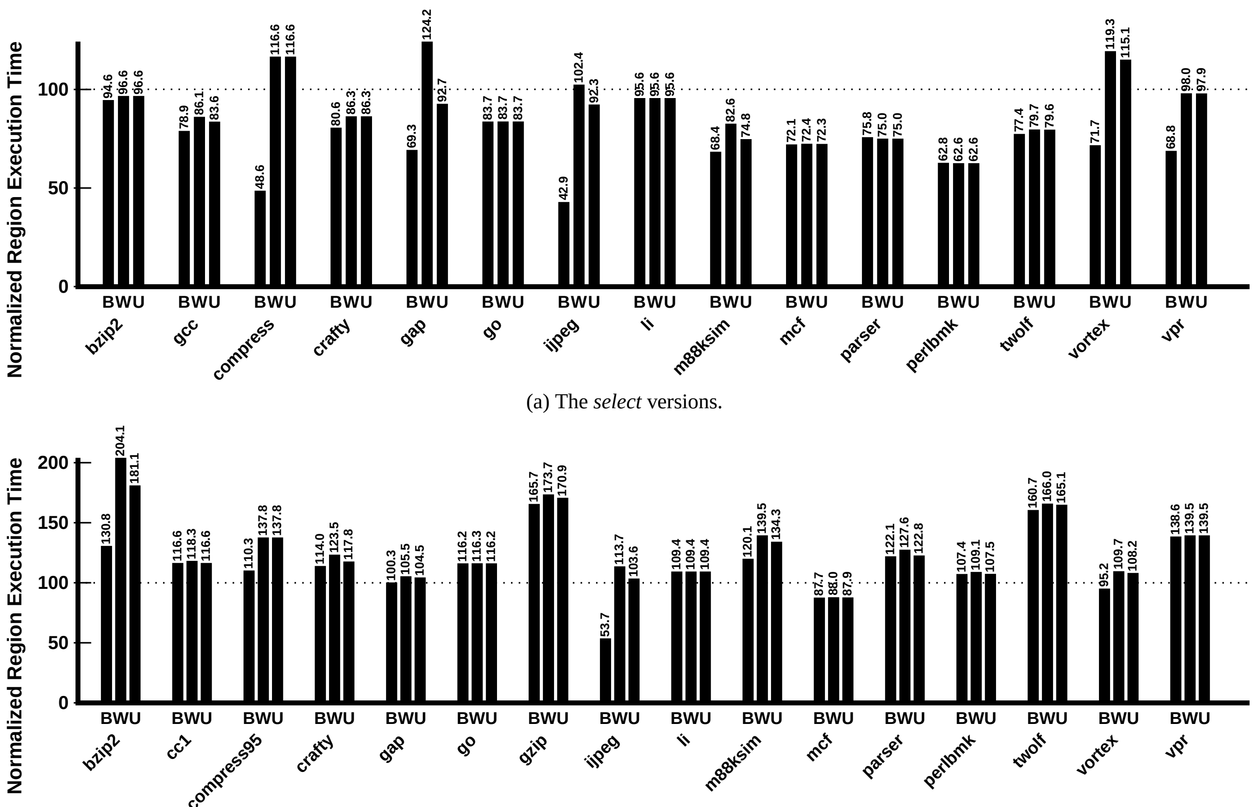
(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 3.14. Impact of support for multiple writers. *W* **does not model support for multiple writers while** *B* **(our baseline architecture) does.**

All speculatively-parallelized loops have been unrolled to maximize performance as described in Section 2.5.1. However, these unrollings were chosen assuming hardware support for multiple writers. For some applications, unrolling can reduce the need for multiple-writers support [66] by ensuring that multiple epochs do not write to the same cache line.

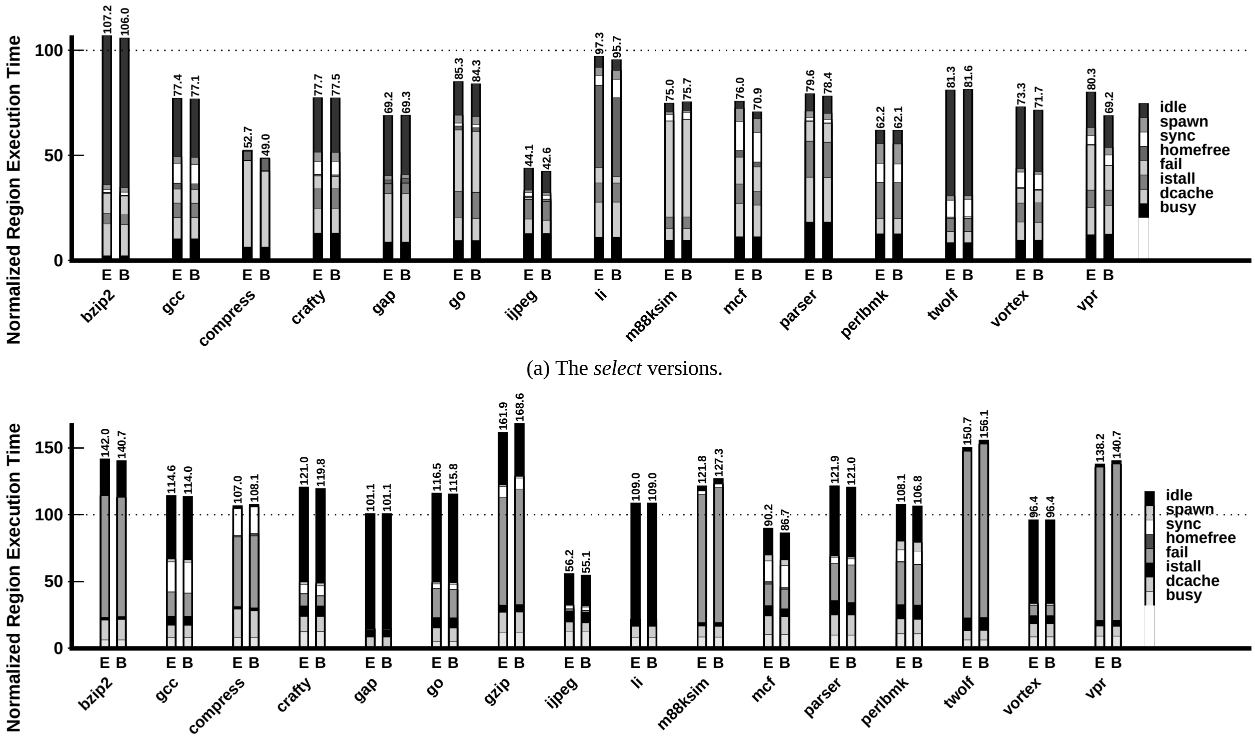
In Figure 3.15 we measure the impact of re-selecting unrollings assuming that multiple-writers are not supported. The *B* experiment is our baseline (which supports multiple writers), while the *W* and *U* experiments do not support multiple writers. In the *U* experiment, we re-select unrollings assuming that multiple writers is not supported. Note that results may differ from Figure 3.14 since they are measured from benchmark versions where all loops have been speculatively parallelized with unrollings of 2, 4, and 8 (as described in Chapter 2.4). For the *select* benchmarks we observe that re-selecting unrollings can partially compensate for multiple writers support: GCC, GAP, IJPEG, and M88KSIM improve significantly with alternate unrollings, while COMPRESS, VORTEX, and VPR do not. Most of the *max-coverage* benchmarks (12 of 15) improve when unrollings are re-selected, although performance is still significantly worse than when multiple writers is supported for *bzip2*, *compress*, and *ijpeg*. In general, the performance



(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 3.15. Impact of re-selecting unrollings when multiple-writers is not supported. B is our baseline (which does support multiple writers), W and U do not support multiple writers, and U re-selects unrollings. Note that results may differ from Figure 3.14 since they are measured from benchmark versions where all loops and all unrollings have been speculatively parallelized.



(a) The *select* versions.

(b) The *max-coverage* versions.

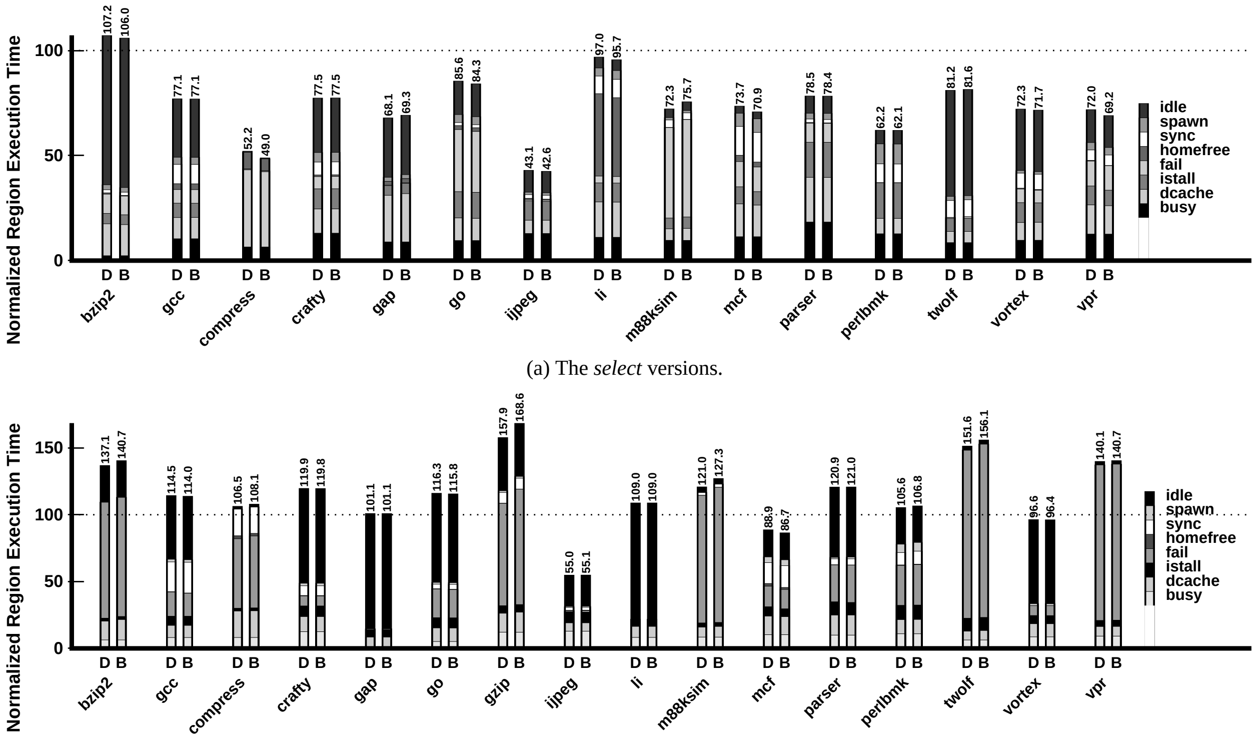
Figure 3.16. Impact of support for speculative coherence messages. E **has no speculative coherence messages while** B **(our baseline) does.**

of multiple writers support cannot be achieved through unrolling alone for these applications.

3.5.2 Speculative Coherence without Speculative Messages

It is possible to re-design the speculative coherence scheme so that speculative messages are not used—this has the advantage that the underlying coherence mechanisms are not modified in any way, and that only the cache state and cache controllers must be extended to support TLS. In the baseline version of the coherence scheme, the speculative messages *read-exclusive-speculative* and *upgrade-request-speculative* attempt to obtain exclusive access to the corresponding cache line. We can replace those requests with non-speculative read requests while maintaining coherence, but potentially degrading performance; without speculative coherence messages, exclusive access will be obtained less frequently, the ORB will have more entries on average, and hardware support that can react instantly to violations will be less agile (as will be described in Section 3.7.4).

Figure 3.16 shows performance both with (B) and without (E) support for speculative coherence messages. Performance is improved for 9 of 15 *select* benchmarks, but only 5 of 15 *max-coverage* benchmarks. Of the improved



(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 3.17. Impact of speculative invalidation of non-speculative cache lines. B **(our baseline) models speculative invalidation of non-speculative cache lines while** D **does not.**

applications, most show a decrease in the amount of failed speculation and time spent waiting for the homefree token. Since the homefree token cannot be passed until all ORB requests are satisfied, minimizing the number of ORB entries allows the homefree token to be passed more quickly. Speculative coherence messages decrease the latency of flushing the ORB significantly for the *select* benchmarks: from 9.9 cycles to 6.6 cycles on average across all applications. The average size of the ORB across all applications is also significantly reduced—the average size decreasing from 1.92 to 0.42 entries and the maximum size from 46.2 to 8.2 entries. However, it is interesting to note that good performance can be obtained without support for speculative coherence messages, so long as the size of the ORB is large enough to accommodate the increase in addresses requiring ownership.

3.5.3 Speculative Invalidation of Non-Speculative Cache Lines

As discussed earlier in Section 3.2.5, one design choice is whether a speculative invalidation should invalidate a cache line in a non-speculative state. Recall that our baseline scheme does speculatively invalidate non-speculative cache lines. As we see in Figure 3.17, this design decision leads to better performance for seven of the 15 benchmarks, most notably VPR which improves by 4%. This option reduces the average number of ORB entries, and hence the latency of flushing the ORB and passing the *homefree token*. In some cases, such as M88KSIM, performance is better without speculative invalidation of non-speculative cache lines: in such cases, the progress of the homefree epoch is hindered because a cache line that it was about to access was invalidated by a speculative epoch. Overall, allowing speculative invalidation of non-speculative cache lines (as is the case for our baseline scheme) improves performance by 0.6% on average (across all applications), indicating that this is not a crucial design decision.

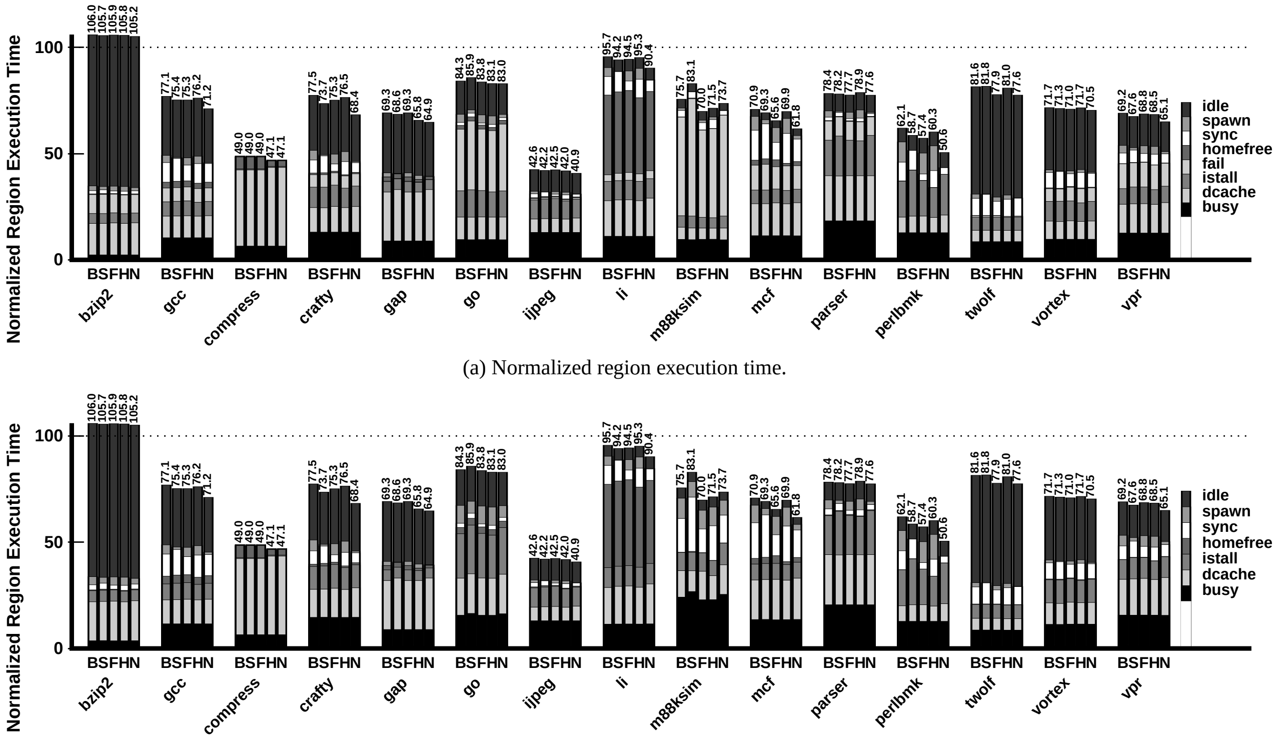
3.6 Sensitivity to Architectural Parameters

To better understand the bottlenecks of TLS execution, it is important to know the performance impact of various architectural parameters. Since many architectural mechanisms can be made larger and faster for an increased cost, we want to understand which features have a significant impact on performance. In this section we explore the sensitivity of TLS execution to the size and complexity of several architectural features including inter-processor communication mechanisms, the memory system, and the pipeline.

3.6.1 Inter-Processor Communication Latencies

Our TLS simulation model allows us to independently modify the latency between processors for several TLS communication events, allowing us to model a range of latencies from the slower speeds of the regular memory system to faster mechanisms such as a dedicated or shared register file. There are three important types of communication event in TLS execution. First, there is the latency to spawn a child thread. In our execution model, this is the latency from when a `spawn` instruction is executed on one processor until the child thread begins executing on the target processor. Since the state for the child thread (the speculative context) has been pre-allocated, it is possible for this mechanism to be quite fast. Second, there is the latency for forwarding a value between speculative threads. This latency is measured from the time a `signal` instruction executes on one processor until the corresponding `wait` instruction on the receiving processor may proceed. Finally, we can independently vary the latency of passing the *homefree* token from one epoch to the next.

In Figure 3.18 we investigate the potential impact of improving each of these latencies independently. Comparing with our baseline (*B*) for which all inter-processor latencies for TLS events are 10 cycles, we independently set the latency for each of the three types of TLS communication event to zero: the *S* experiment has no spawn latency, the *F*



(a) Normalized region execution time.

(b) Normalized region execution time with failed speculation incorporated.

Figure 3.18. Impact of various communication latencies on the *select* benchmarks. B is our baseline which models 10 cycle interprocessor communication latency, S modifies B to have a zero-cycle spawn latency, F modifies B to have a zero-cycle forwarding latency, H modifies B to pass the homefree token in zero cycles, and N has no interprocessor communication latency.

experiment has no forwarding latency, and the *H* experiment has no latency for forwarding the homefree token.

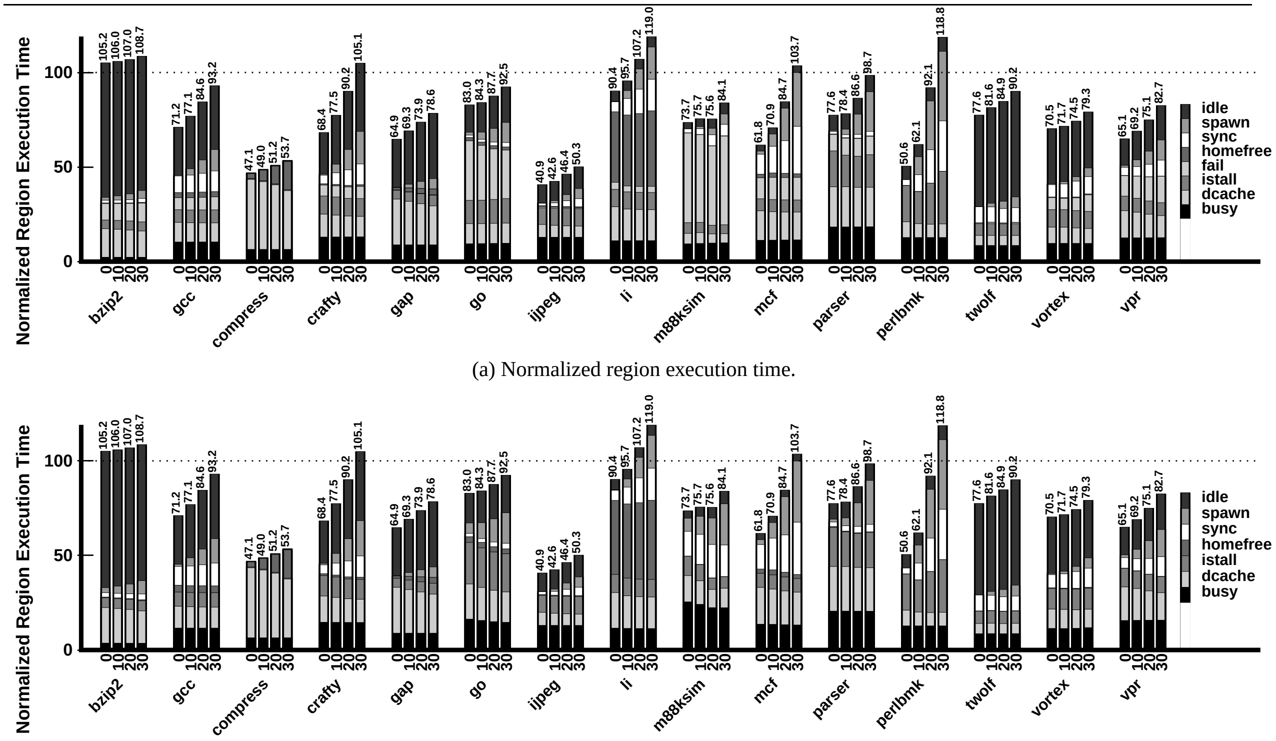
When spawn latency is zero (*S*), the remaining *spawn* segment for nearly all applications is negligible, indicating that spawn latency is not a bottleneck. For GO and M88KSIM, performance is slightly worse when we eliminate spawn latency: since there is more parallel overlap, there is also a greater incidence of violated data dependences and hence a larger failed speculation segment.

When the forwarding latency is set to zero (*F*), the *sync* portion is reduced for most applications. However, this component of execution time is not removed completely, and in some cases it is only slightly reduced—this indicates that the actual communication of signal messages (once they are ready) is not a bottleneck.

When the latency of sending the homefree token is set to zero (*H*), performance improves slightly in eight cases. However, for the application which is the most constrained by the homefree mechanism (LI), this decreased latency shrinks the homefree segment only slightly, and does not have much impact on overall performance—further supporting the claim that the *homefree* segment for this application is the result of load imbalance. These results indicate that the communication latency for passing the homefree token is not a bottleneck.

Finally, we measure the impact of zero-cycle latency for all three communication events at once (*N*). Interestingly, for all but one case (M88KSIM) this results in better performance than for any of the previous experiments, indicating that all three communication latencies must be improved in order to achieve an overall performance benefit. Intuitively, since all three communication events are potential serialization points for parallel execution, lowering the latency for just one in isolation results simply in spending more time waiting for another. For 11 of the 15 benchmarks, however, lowering all three latencies did not have a large impact on overall region performance—hence a more expensive interprocessor communication mechanism is likely not worth the cost.

An inter-processor communication latency of 10 cycles is itself quite fast. Can speedups be achieved with larger communication latencies? Figure 3.19 shows the impact of varying all three communication latencies simultaneously from zero to thirty cycles. About six of the fifteen applications are extremely sensitive to the communication latencies. *Spawn* time is the component that increases the most: since it is the first latency to occur for each epoch, it receives a majority of the blame. The increase in the *sync* segment is less pronounced, while the *homefree* segment only increases slowly for COMPRESS. One application (LI) no longer speeds up when communication latency is 20 cycles, while five applications (CRAFTY, LI, MCF, PARSER, and PERLBMK) no longer speed up when communication latency is 30



(a) Normalized region execution time.

(b) Normalized region execution time failed speculation incorporated.

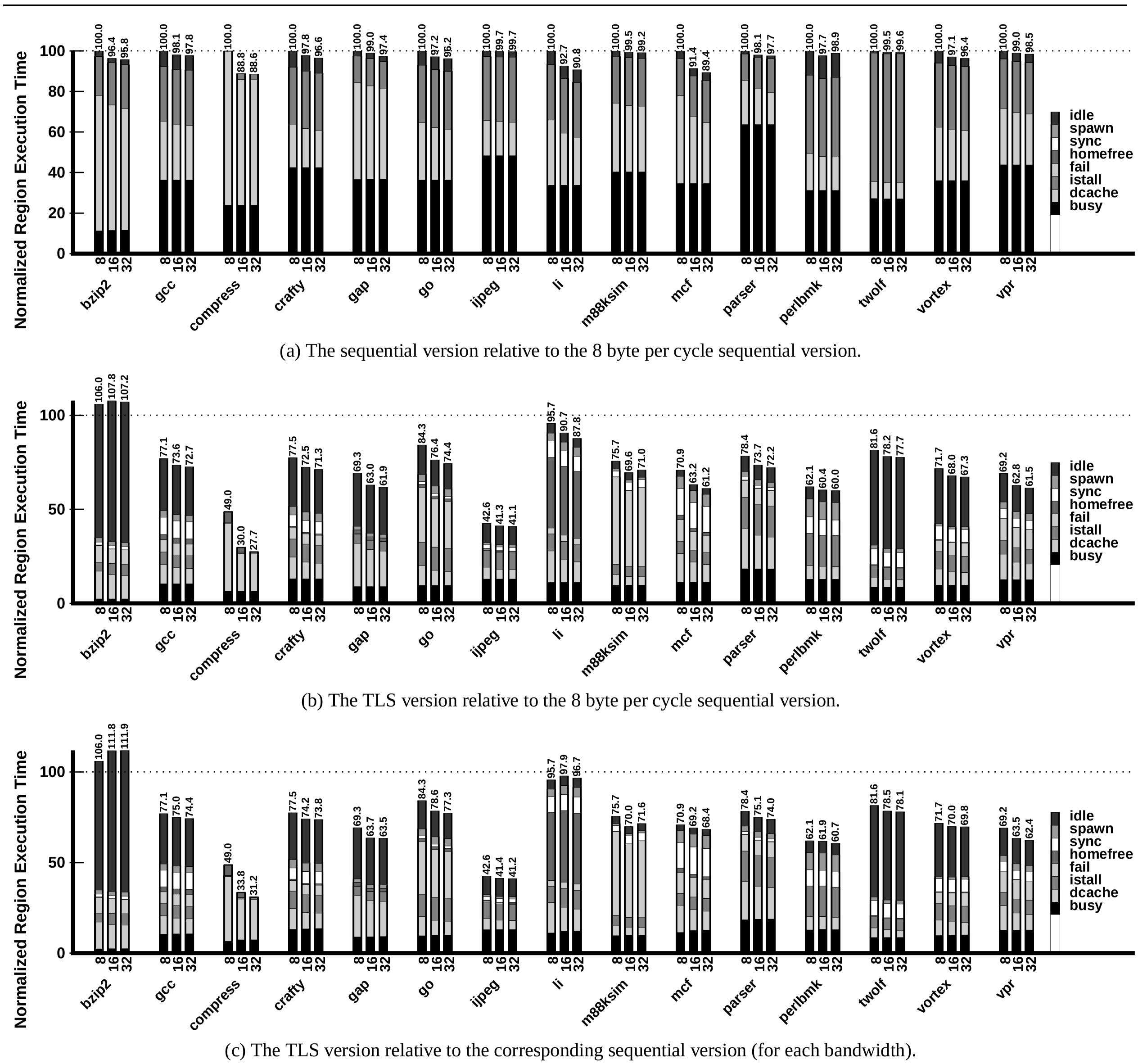
Figure 3.19. Impact of varying communication latency (by tens of cycles).

cycles. These results suggest that an inter-processor communication latency of no more than 20 cycles is necessary for good performance on general-purpose programs.

3.6.2 Memory System

Our scheme for TLS uses the caches and coherence scheme to implement data dependence tracking and buffering speculative state, and hence requires an efficient underlying memory system. In this section we investigate the sensitivity of TLS execution to three aspects of the memory system's design: the bandwidth of the interconnection network between processors, the number of data references handlers processor, and the size and associativity of the first-level data caches.

Figure 3.20 shows the impact of varying the crossbar bandwidth from 8 to 32 bytes per cycle—for our baseline architecture it is 8 bytes per cycle. Since we model a fully-connected crossbar, each processor may be connected to one bank of the unified, second-level cache at a time. Recall that we do model the extra overhead of piggybacking epoch numbers along with speculative coherence messages. According to the figure, increasing crossbar bandwidth has only a small impact on performance which is similar for both sequential and speculative executions; the exception is



(a) The sequential version relative to the 8 byte per cycle sequential version.

(b) The TLS version relative to the 8 byte per cycle sequential version.

(c) The TLS version relative to the corresponding sequential version (for each bandwidth).

Figure 3.20. Varying crossbar bandwidth from 8 to 32 bytes per cycle (for the *select* benchmarks). Note that our baseline architecture has a crossbar bandwidth of 8 bytes per cycle.

COMPRESS which improves significantly when bandwidth is increased from 8 to 16 bytes per cycle—this indicates that much of the large dcache component for COMPRESS is due to crossbar transfer time, as opposed to bank contention or other memory system bottleneck.

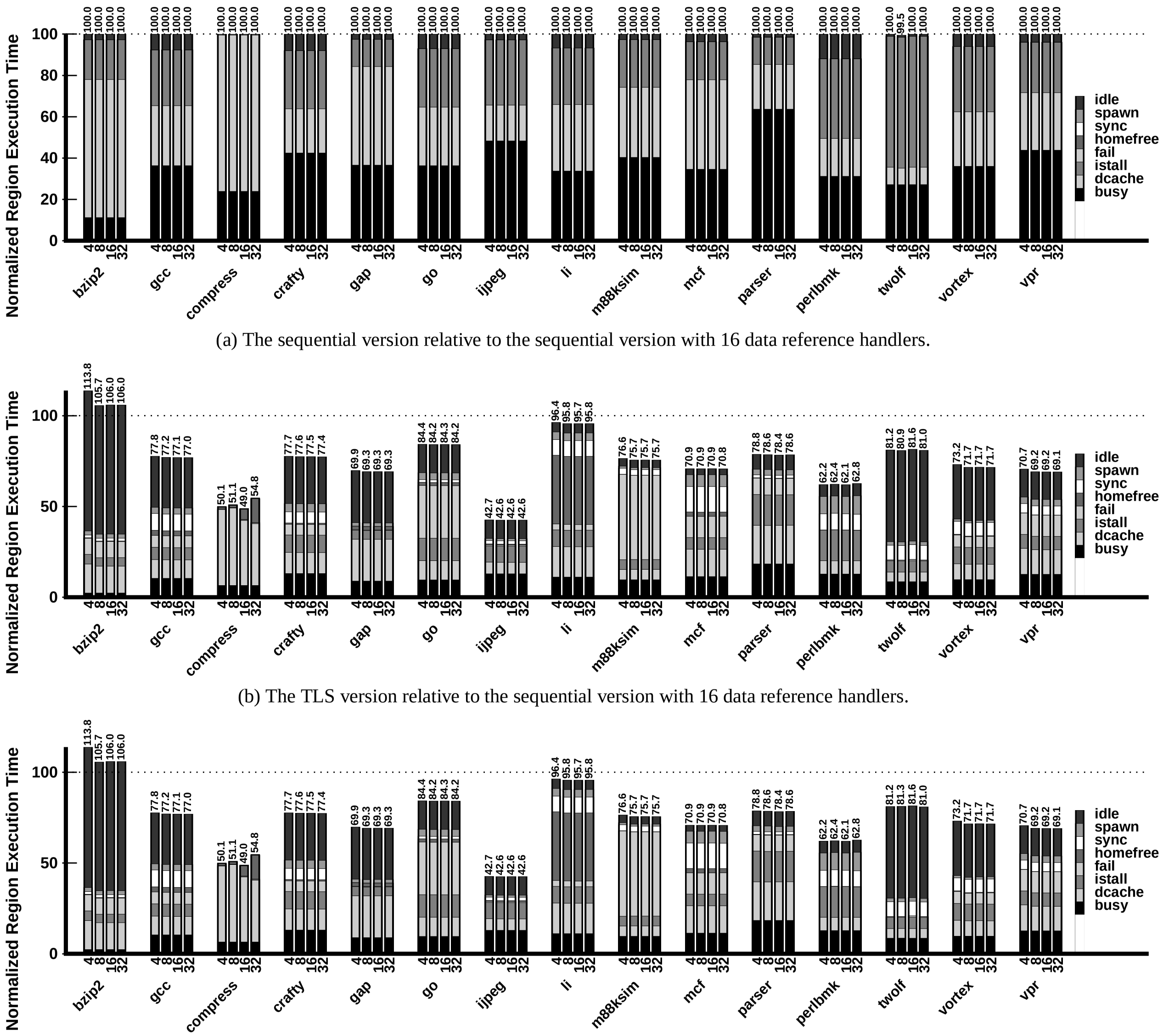
Figure 3.21 shows the impact of varying the number of data reference handlers from 4 to 32—for our baseline architecture it is 16. This experiment has almost no performance impact for the sequential versions, indicating that 4 data reference handlers is sufficient for sequential execution. For the speculative versions, other than for VORTEX, increasing the number of data reference handlers does not have a significant performance impact—4 data reference handlers per processor are sufficient for speculative execution as well.

In Figure 3.22 we vary the size of the data caches from 8KB to 64KB—our baseline architecture has a 32KB data cache per processor. This experiment has a significant impact on both the sequential and TLS versions of the applications, with larger caches performing better. As is evident in Figure 3.22(d) for the TLS versions, larger caches also reduce the amount of failed speculation due to replacement. For the 8KB caches, CRAFTY, GO, IJPEG, PARSER, and VORTEX all suffer a significant amount of failed speculation due to replacement, indicating that 8KB caches alone are insufficient. Performance improves for 16KB caches, while 64KB caches do not offer a significant improvement over 32KB caches.

Finally, in Figure 3.23 we evaluate the impact of having direct-mapped first-level data caches, as opposed to the two-way set-associative data caches used in our baseline architecture. We observe that with direct-mapped caches there is an increase in failed speculation for many benchmarks, due to an increase in violations related to the replacement of cache lines in speculative states. This effect could be mitigated by the addition of a victim cache [37]: a small, fully associative buffer which saves recently evicted cache lines. We note that decent speedup is still possible with direct-mapped caches, but that 2-way set-associative caches perform significantly better (an average of 6.6% better when measured relative to the sequential execution with the corresponding associativity, as shown in Figure 3.23(c)).

3.6.3 Reorder Buffer Size and Complexity

Another important architectural feature is the size and complexity of the pipeline. In this section we investigate the impact on TLS execution of two aspects of issue logic design: support for out-of-order execution, and the size of the instruction window.

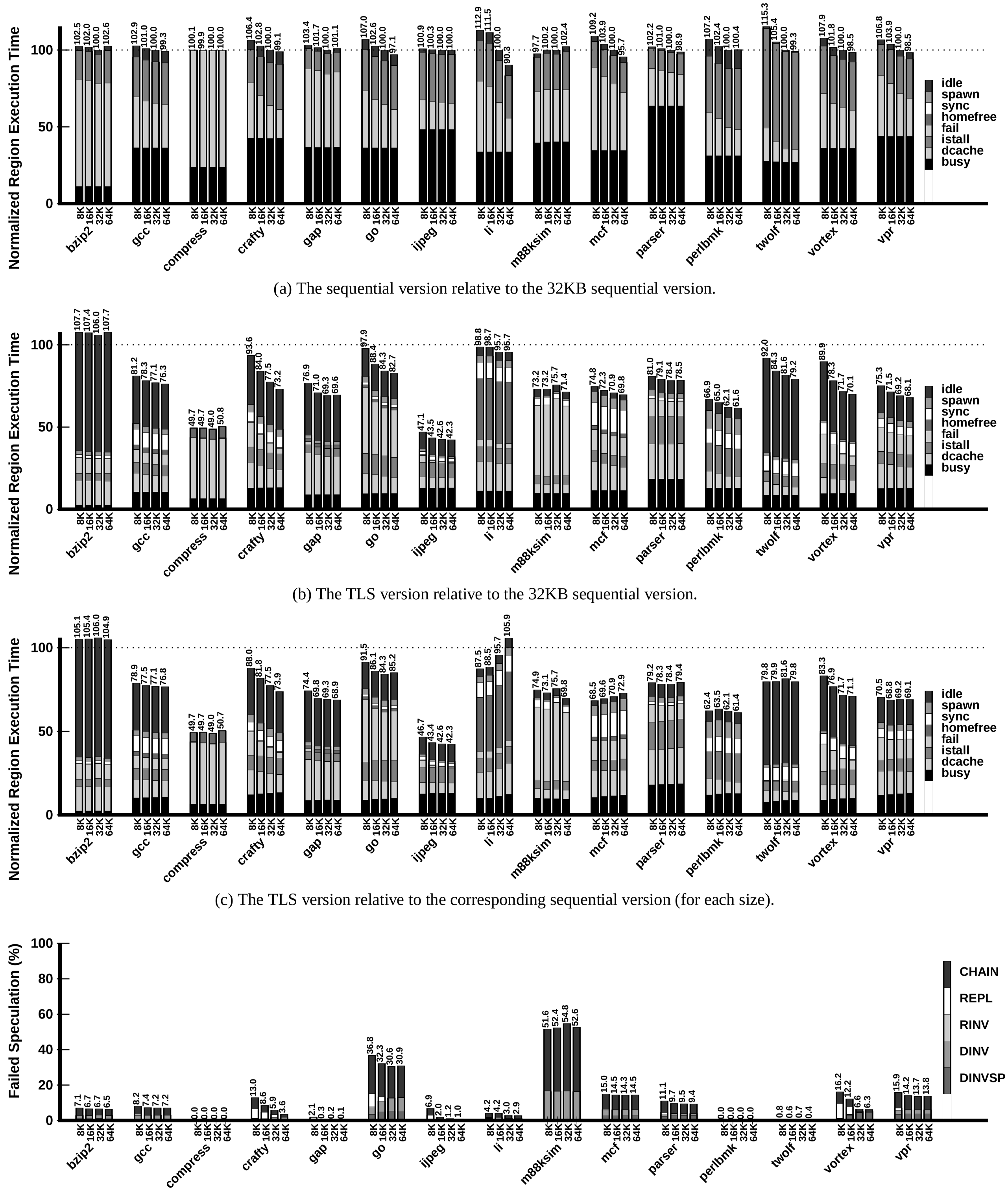


(a) The sequential version relative to the sequential version with 16 data reference handlers.

(b) The TLS version relative to the sequential version with 16 data reference handlers.

(c) The TLS version relative to the corresponding sequential version (for each number of handlers).

Figure 3.21. Varying the number of data reference handlers from 4 to 32 (for the *select* benchmarks). Note that our baseline architecture has 16 data reference handlers per processor.



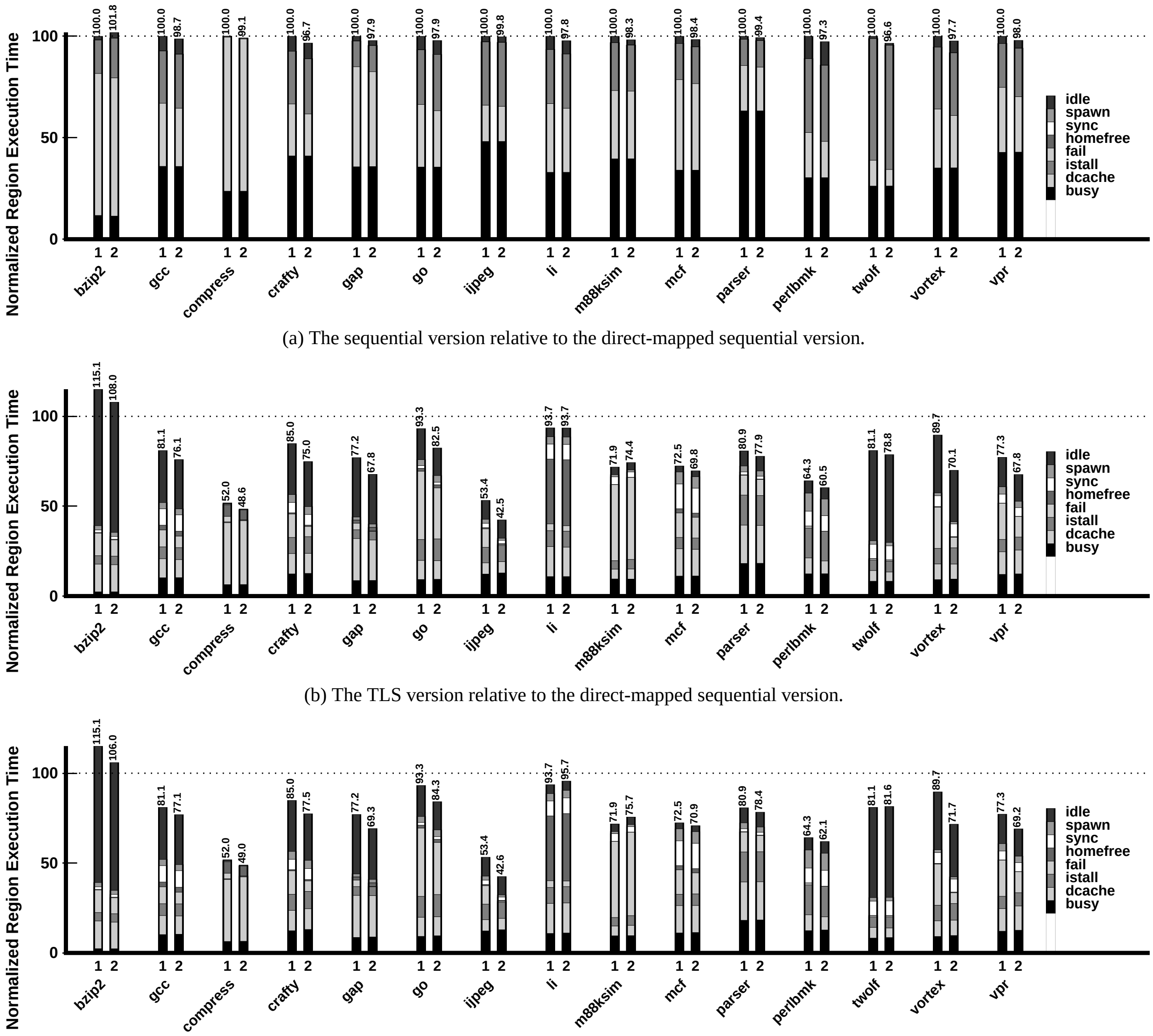
(a) The sequential version relative to the 32KB sequential version.

(b) The TLS version relative to the 32KB sequential version.

(c) The TLS version relative to the corresponding sequential version (for each size).

(d) Percent failed speculation for the TLS version.

Figure 3.22. Varying data cache size from 8KB to 64KB (for the *select* benchmarks). Note that our baseline architecture has a 32KB data cache per processor.

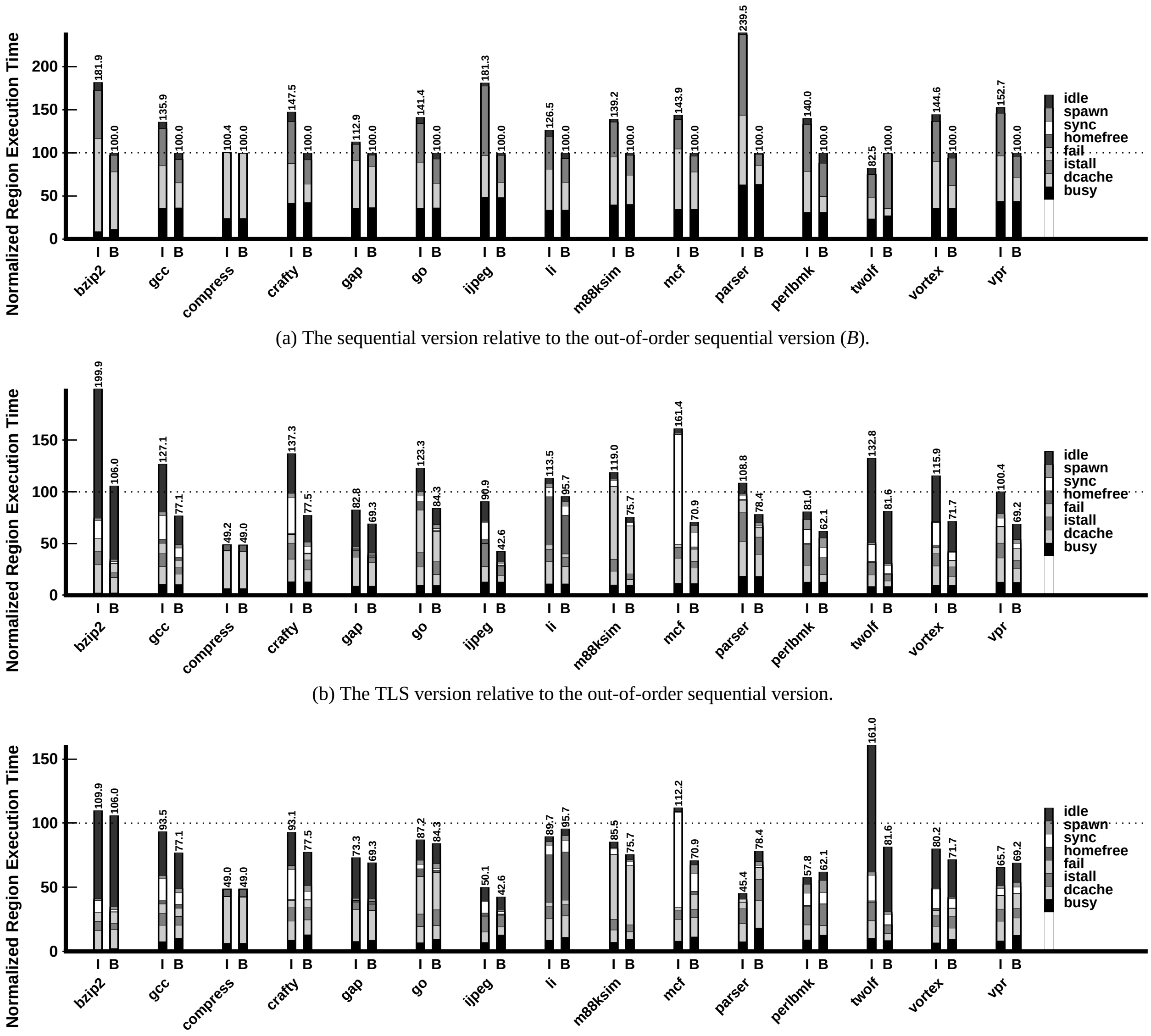


(a) The sequential version relative to the direct-mapped sequential version.

(b) The TLS version relative to the direct-mapped sequential version.

(c) The TLS version relative to the corresponding sequential version (for each associativity).

Figure 3.23. Varying data cache associativity from direct-mapped (1) to 2-way (for the *select* benchmarks). Note that our baseline architecture is 2-way set-associative.



(a) The sequential version relative to the out-of-order sequential version (*B*).

(b) The TLS version relative to the out-of-order sequential version.

(c) The TLS version relative to the corresponding sequential version (for each pipeline type).

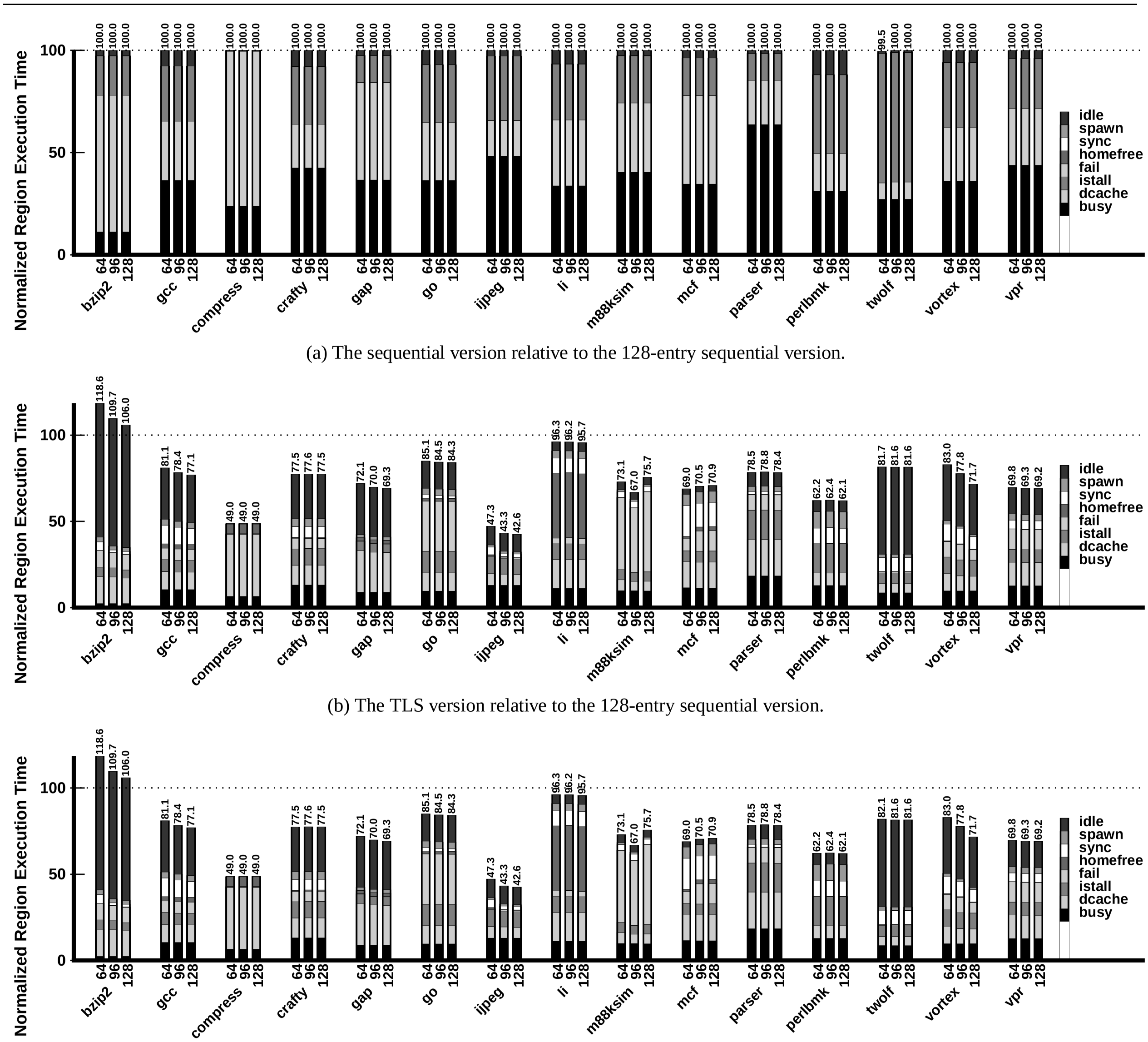
Figure 3.24. Impact of issuing memory references out-of-order. I **models an in-order-issue pipeline, and** B **(our baseline) models an out-of-order-issue pipeline.**

Figure 3.24 compares the speedup of TLS execution on an in-order-issue pipeline (*I*) to that of an out-of-order-issue pipeline (*B*), which is our baseline architecture. We observe that the out-of-order issue machine enhances TLS execution more so than sequential execution for 10 of the 15 applications. It is apparent that the biggest gain of out-of-order execution is in reducing synchronization (*sync*), most notably in MCF. When out-of-order execution is supported, the pipeline may issue around any stalled reference to the forwarding frame, thus decreasing the impact of the synchronization mechanisms on execution. This result emphasizes the complementary behavior of TLS techniques and out-of-order superscalar techniques.

Each generation of microprocessor tends to have an increasingly-large reorder buffer, so it is important to understand whether this trend benefits or hinders TLS execution. Figure 3.25 shows the impact of varying the size of the reorder buffer from 64 to 256 entries—our baseline architecture has a reorder buffer of 128 entries. Surprisingly, varying the size of the reorder buffer beyond 64 entries does not impact the performance of the sequential benchmarks, indicating that instruction issue is not a bottleneck for the speculative regions in those applications. Varying the reorder buffer size impacts performance for the TLS applications in three different ways. First, there are nine applications that are not affected—this indicates that the additional reorder buffer entries are not utilized by the TLS executions of the speculative regions that are parallelized. Second, for two applications (M88KSIM and MCF) the amount of failed speculation increases—this is due to a greater number of memory references that are issued out-of-order, which in turn can increase the odds of a data dependence violation. Third, in several cases performance improves for the TLS execution more than for the sequential execution (BZIP2, GCC, IJPEG, and VORTEX). These applications mostly benefit from decreased *sync* segments, which indicates less time spent stalled waiting for forwarded values. The larger reorder buffer size allows more instructions to be issued around such stalled waits, therefore decreasing the impact of that potential serialization. This important result demonstrates that increasing reorder buffer sizes and TLS execution are also complementary, and that the performance benefits of TLS cannot be achieved solely by increasing the size of the reorder buffer.

3.7 Implementation Alternatives

In this section we investigate design alternatives for some features of TLS hardware support. First, we compare the performance of designs with less aggressive hardware support than ours. Second, we discuss the issues involved in



(a) The sequential version relative to the 128-entry sequential version.

(b) The TLS version relative to the 128-entry sequential version.

(c) The TLS version relative to the corresponding sequential version (for each reorder buffer size).

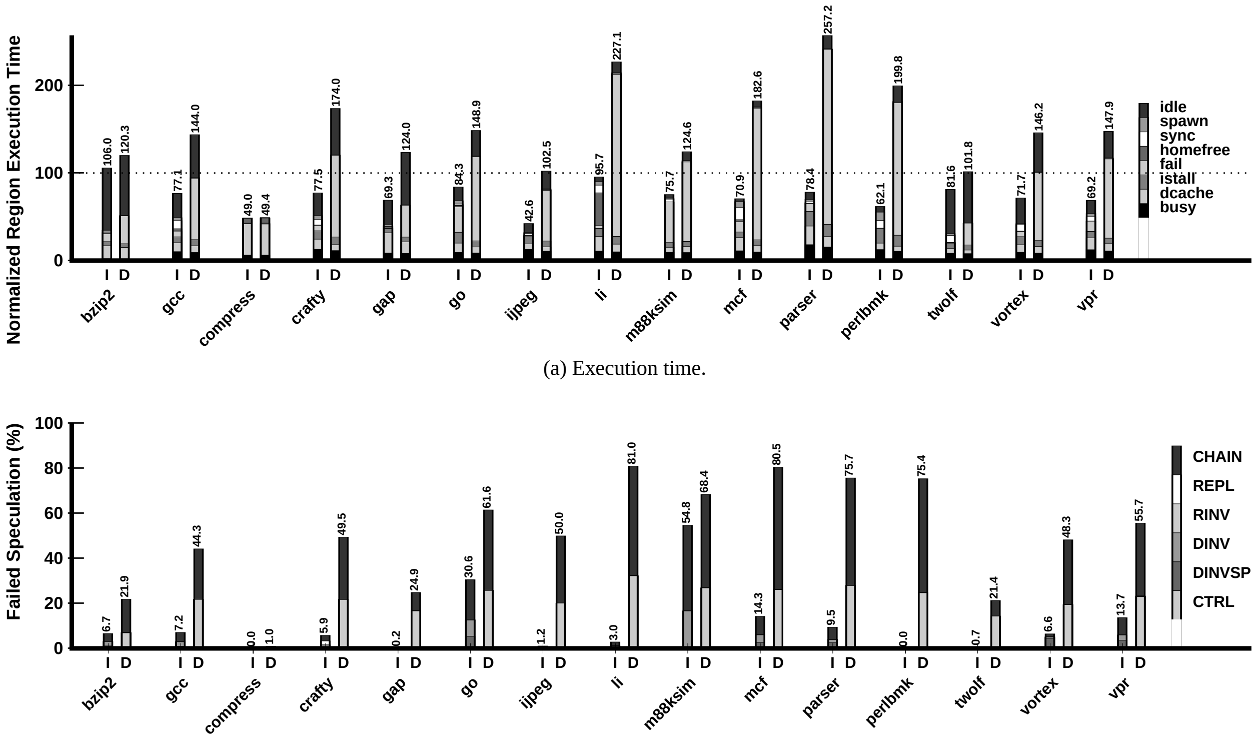
Figure 3.25. Impact of varying the reorder buffer size, from 64 to 256 entries. Note that our baseline architecture has a 128-entry reorder buffer.

tracking data dependences by extending a snoopy, write-through coherence scheme (as opposed to our invalidation-based, write-back coherence scheme). Second, we examine implementation alternatives for the forwarding frame. Third, we take a closer look at violation detection—whether violations should be detected by poll or interrupt, and whether it is beneficial to suspend an epoch rather than suffer from a violation due to loss of speculative state. Finally, we determine whether the homefree token should be a mechanism that is visible to the hardware, or whether a software-only implementation is sufficient.

3.7.1 Less Aggressive Designs

Here we briefly attempt to justify our point in the design space of TLS hardware support. While a range of complexity is possible, our scheme attempts to minimize the size and complexity of TLS-specific structures while maximizing performance. While more complex hardware support such as the highly specialized Multiscalar [27, 65] would likely provide improved performance at a higher cost, it is important to understand the performance advantages of our approach over schemes that are less complex. At the simplest end of this design space is deep uniprocessor speculation supported by a large instruction window. Another scheme that is less complex than ours is to simply use the load and store queues to buffer speculative data. In this section, we show that neither of these less-complex schemes is sufficient to obtain the performance improvements of our baseline scheme.

One of the biggest potential performance advantages of TLS is the exploitation of control independence [44]: in contrast to a single, extremely-large instruction window with deep uniprocessor speculation, the independent speculative threads of a TLS architecture allow logically-later threads to be sheltered from branch mispredicts in logically-earlier threads. But is there really a performance advantage to this inter-thread control independence? Figure 3.26 shows an approximation of the benefits of control independence. In the *D* experiment, any branch mispredict causes speculation to fail for all currently-running, logically-later epochs. Control-dependence (*D*) results in an overwhelming amount of failed speculation for every application except COMPRESS, while control-independence (*I*) shows an enormous improvement. Figure 3.26(b) shows the breakdown of time spent on failed speculation (similar to Figure 3.12), with the additional violation reason of *CTRL*, which represents those caused by control dependences. We observe that in every case, control dependences overwhelm every other form of violation, and only in one case (COMPRESS) is speedup achieved under this restricted model (because the speculative region in COMPRESS has highly-predictable



(a) Execution time.

(b) Failed speculation.

Figure 3.26. Benefits of control independence. I **is control independent (our baseline) and** D **is control dependent.**

branches). Hence control independence is an extremely important feature of TLS.

Another design that requires less hardware support than ours is to simply extend the load and store queues to buffer speculative state, rather than extending the first-level data caches. While the data caches can hold a greater amount of state, extending them to support TLS does require a non-trivial amount of real-estate. But are reasonably-sized load and store queues sufficient to capture all of the necessary speculative state?

Recall that we need to track which memory locations have been speculatively loaded, as well as buffer speculative modifications from memory. We can extend the load queue to track which loads are speculative (or extend the store queue to also buffer loads if there is no load queue), and extend the store queue to buffer speculative stores. Both queues need to retain this speculative state until the corresponding epoch receives the homefree token and commits. Should an epoch completely fill one of these queues, it can simply stall until it is homefree. Thus the comparison of our approach with this queue-based approach is solely a matter of cost versus performance.

We estimate the performance of such a queue-based approach by modeling a limited version of our speculative coherence scheme: the number of speculative loads and stores per epoch are limited to model reasonable-sized load

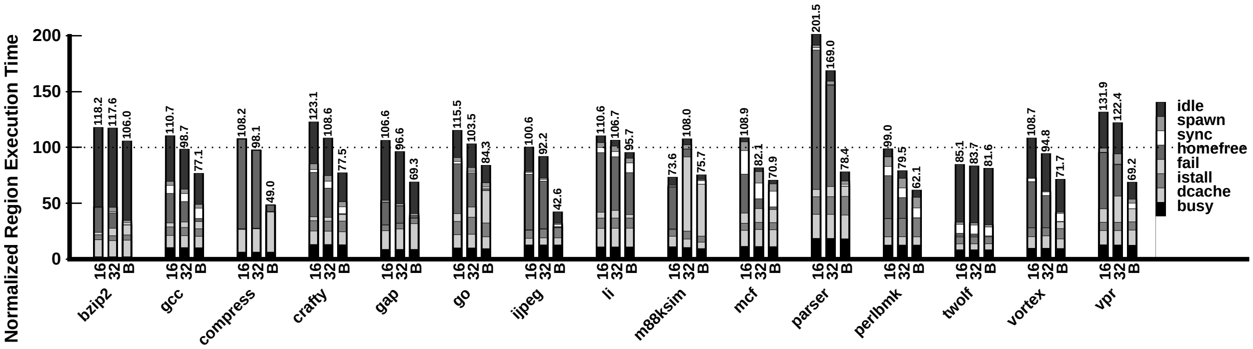


Figure 3.27. Comparison with hardware support that only uses the load/store queues as speculative buffers. *16* **models 16-entry load and store speculative buffers,** *32* **models 32-entry load and store speculative buffers, and** *B* **is our baseline hardware support (using the first-level data caches as speculative buffers).**

and store queues. Figure 3.27 compares the performance of our baseline hardware support (*B*) with that of two queue-based approaches, one with 16-entry load and store queues and one with 32-entry load and store queues. With 16-entry queues, 12 of the 15 applications slow down. Note that when any queue becomes full, the corresponding epoch stalls until it receives the homefree token, which enlarges the *homefree* segment in the result performance bar. Only in one case, M88KSIM, does the queue-based approach out-perform our baseline: the queue-based approach indirectly synchronizes dependences between epochs, drastically reducing the amount of failed speculation. With 32-entry queues, 7 of the 15 applications still slow down, and another 4 simply break even. For the remaining 4 applications, speedup is significantly less than that of our baseline architecture. These results indicate that a queue-based approach is not sufficient to capture interesting performance benefits for most applications.

3.7.2 Snoopy, Write-Through Coherence

In our approach to hardware support for tracking data dependences and buffering speculative state, we extend invalidation-based cache coherence. An alternative approach would be to extend a snoopy, write-through, update-based coherence scheme—an example of such a scheme is that used in the Hydra [34], which introduces speculative buffers between the write-through first-level caches and the unified second-level cache. In keeping with the philosophy of minimizing speculation-specific structures (such as the speculative buffers in Hydra), we instead propose that the second-level unified cache is itself extended to have speculative states and to buffer speculative modifications from regular memory. While a thorough evaluation of such an implementation of snoopy, write-through coherence is beyond the scope of this dissertation, the following discusses some of the challenges of such an approach.

The nature of an update-based coherence scheme with write-through first-level data caches is such that every write is propagated to the second-level cache and all other processors snoop the bus to update their own caches. To extend this behavior for TLS, we would piggy-back the epoch number along with each update. This way, only processors executing epochs that are logically-later than that of the originator of the update message would update their caches. This implementation would also facilitate *implicit forwarding*, as defined in Section 3.1.2 and evaluated later in Section 4.2.1.

There are three main challenges with the update-based approach. The first is that the commit operation must now be performed in both the first-level cache and also the unified, second-level cache. The second is the fact that the second-level cache is polluted with speculative data which must be invalidated when speculation fails, and fresh copies of those cache lines must be re-loaded from memory. The third difficulty is with supporting multiple speculative writers of the same cache line: in this case the second-level cache must support *replication* of cache lines, as will be described and evaluated for a first-level cache later in Section 4.2.3. The final difficulty is with the scalability of such a scheme. Sending updates to every processor in a multi-node system when there are a large number of nodes is not feasible. However, some combination of using snoopy, write-through coherence within each node and invalidation-based, write-back coherence between nodes is more reasonable.

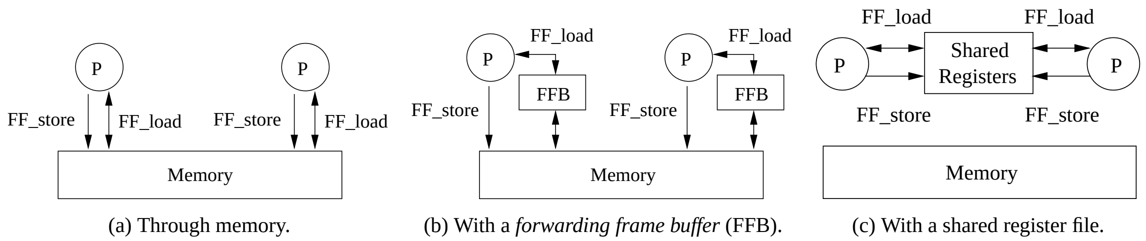
3.7.3 Implementing the Forwarding Frame

Our scheme for TLS support provides an architected *forwarding frame* where, as explained in Section 2.3.1, a certain portion of the stack is designated for synchronizing and forwarding values between epochs. Since the interface to the forwarding frame is fairly simple, we are free to explore alternatives for its actual implementation. Potential designs cover a spectrum of hardware complexity, ranging from shared register files to simple forwarding through the regular memory system. In this section, we explore some of these alternatives and their impact on performance.

Before we evaluate implementation alternatives, it is important to know some statistics on forwarding frame usage. First, the size of the forwarding frame is set by the compiler on a region-by-region basis, and can vary quite widely. According to the statistics in Table 3.7.3, any design with a fixed-size structure for forwarding values between epochs (such as a shared register file) should provide at least 30 entries, as well as a strategy for handling the case of overflow. One could imagine a design that includes several mechanisms for forwarding of varying complexity and speed, with a

Table 3.3. Forwarding Frame Sizes (in 8-byte words)

Application	*select*		*max-coverage*	
	Avg	**Max**	**Avg**	**Max**
BZIP2	6.0	8	4.9	8
GCC	6.2	20	7.0	32
COMPRESS	8.0	8	9.0	16
CRAFTY	7.3	8	9.3	20
GAP	8.0	8	4.8	8
GO	5.5	12	5.4	16
IJPEG	10.3	28	9.7	28
LI	4.0	4	8.0	8
M88KSIM	6.0	12	6.7	12
MCF	6.0	8	7.5	28
PARSER	5.6	8	6.2	24
PERLBMK	5.0	8	7.8	16
TWOLF	4.7	8	6.0	36
VORTEX	9.0	16	10.7	24
VPR	10.0	12	16.0	16
All	6.5	28	9.5	36



(a) Through memory. (b) With a *forwarding frame buffer* (FFB). (c) With a shared register file.

Figure 3.28. Implementation alternatives for the forwarding frame, where forwarding frame loads (FF_load) and stores (FF_store) are managed by different mechanisms.

smaller number of architected locations for fast communication as well as a slower method (such as regular memory) that is effectively unlimited in size (similar to procedure calling conventions).

Figure 3.28 illustrates the three implementation alternatives for the forwarding frame that we evaluate. Our baseline architecture assumes that the forwarding frame is implemented by the cache coherence scheme such that forwarding frame entries are stored in the first-level data cache, and are forwarded between epochs through the regular memory system (as shown in Figure 3.28(a)). Our compiler ensures that each forwarding frame entry is allocated to its own cache line to facilitate forwarding through regular coherence mechanisms which communicate at a cache line granularity. While this simple approach is somewhat wasteful of space in the first-level caches, we will show that it is quite effective.

Since forwarding frame values may be referenced frequently (especially since `gcc`, our back-end compiler, does not register allocate forwarding frame entries), the additional loads and stores may put a strain on the mechanisms for handling memory references. We evaluate the impact of two increasingly complex mechanisms for implementing the

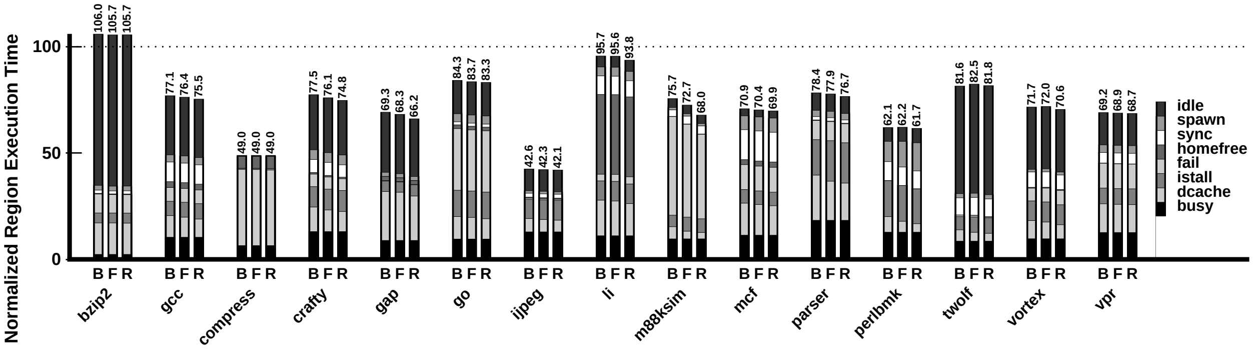


Figure 3.29. Impact of special hardware support for the forwarding frame. For B, all forwarding frame references are through regular memory (our baseline); for F, forwarding frame references are first loaded from regular memory but then saved in a *forwarding frame buffer*; and for R, all forwarding frame references are to a shared register file.

forwarding frame. First, as shown in Figure 3.28(b), we can use a *forwarding frame buffer* (FFB) to cache forwarding frame values. For this alternative, the forwarding frame is allocated in memory as it is for our baseline. However, the result of the first load of a forwarding frame location is cached in the FFB, and subsequent loads to the same location are satisfied by the FFB. Second, as shown in Figure 3.28(c), we can implement a shared register file for forwarding frame references, saving the data caches and memory from the burden of supporting forwarding frame accesses. In this case, we assume that the inter-processor communication latency for the register file is also 10 cycles.

In Figure 3.29, we evaluate several alternative implementations of the forwarding frame. In our baseline scheme (*B*), the forwarding frame is implemented using the memory system. Next, we measure the performance impact of a *forwarding frame buffer* (*F*). Looking at the figure, we see that the use of the forwarding frame buffer does improve execution time by decreasing the *dcache* portion of execution for several applications, although this gain is not dramatic. Finally, we evaluate the performance of a shared register file (*R*). We observe that again performance is improved slightly by reducing the *dcache* segment for many applications, but that the gain over our baseline scheme is not significant. These results are important since they indicate that allocating the forwarding frame in regular memory and having it reside in the first-level data caches does not cause conflicts, and does not have a significant negative impact on performance.

3.7.4 Handling Violations

A key aspect of TLS hardware support is the detection of data dependence violations. Once a violation is detected, there are several possibilities for how to proceed. In some cases, the violation itself can be avoided by immediately

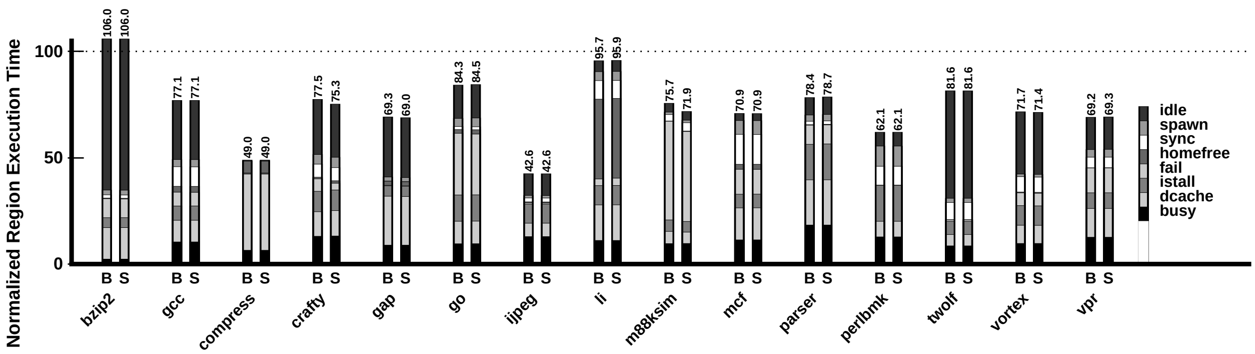
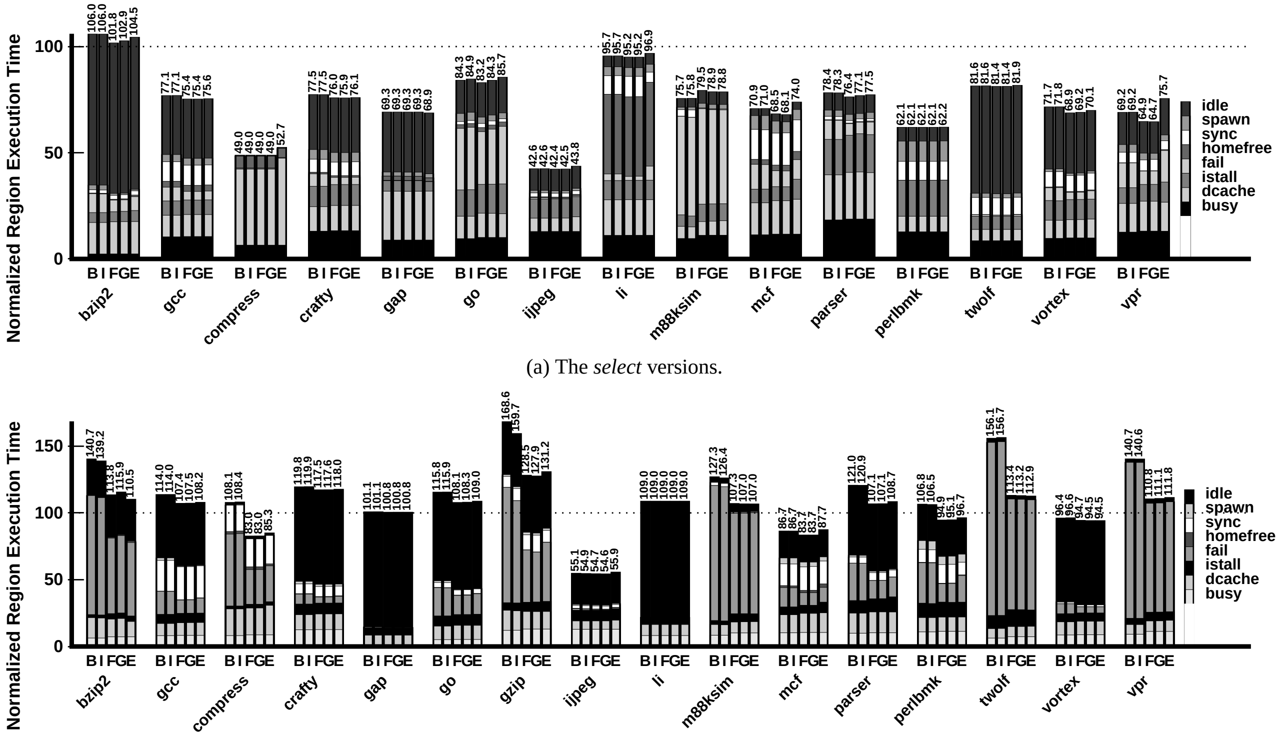


Figure 3.30. Impact of support for avoiding any violation due to cache line replacement by immediately suspending the offending epoch until it becomes homefree. B **is our baseline hardware support, and** S **suspends any epoch that attempts to replace a speculative cache line until it becomes homefree.**

suspending execution until the epoch becomes homefree. When a violation cannot be avoided, there are still several options for how to notify the software that recovery is necessary and for when recovery should begin. We evaluate several of these alternatives in this section.

Recall that if a speculatively accessed cache line is replaced, speculation must fail because we can no longer track dependence violations. In our baseline scheme, if an epoch is about to evict a speculative line from the cache, we simply let it proceed and signal a dependence violation. Since one epoch is always guaranteed to be non-speculative, this scheme will not deadlock (since the non-speculative or *homefree* epoch always makes forward progress). Alternatively, we could *suspend* the epoch that triggered the replacement until it becomes *homefree*, at which point we can safely allow the replacement to occur since the cache line is no longer speculative and we are not losing any speculative state.

In Figure 3.30, we compare the performance impact of support for suspending (*S*) with that of our baseline (*B*). Recall from Figure 3.12 that only CRAFTY and VORTEX suffer from a significant number of violations due to cache line replacement, although these are responsible for only a small portion of overall execution time. As expected, they both benefit slightly from this suspend feature. While not visible in Figure 3.30, M88KSIM does suffer from replacement violations (which account for 0.003% of cycles wasted on failed speculation for that application), and hence also benefits from this support—although some of the observed benefit is from fortuitous changes in timing that result in less overall failed speculation. Overall, the ability to avoid violations by suspending is not a big performance win. We will discuss further uses of the suspend feature later in Chapter 4.



(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 3.31. Impact of various violation notification and recovery strategies. B is our baseline strategy which polls for a violation at the end of an epoch, and squashed epochs re-spawn; I modifies B such that squashed epochs store their initial state and restart independently, without having to respawn; F modifies B such that violations are notified immediately by an interrupt; G combines both I and F; and E modifies G by not having speculative coherence messages.

Next, we evaluate several different implementations for the notification of violations and recovery. In our baseline scheme, an epoch is notified of any violation at the end of its execution through a polling mechanism. Alternatively, a violated epoch could be squashed immediately through a user-level interrupt. Such an interrupt would not invoke the operating system, but simply trigger the mechanisms which recover from failed speculation as well as a user-level handler for indicating to software that speculation has failed. One design issue is whether an epoch should store its initial parameters (either in memory or in some special structure) or instead be *re-spawned* by its parent epoch and have its initial parameters re-sent.

We examine these possibilities in Figure 3.31. The first experiment (*B*) shows the performance of our baseline hardware support—with polling violation detection and re-spawn recovery. The next experiment (*I*) modifies our baseline by allowing epochs to store initial state locally; this way, violated epochs can restart independently without having to re-spawn. Surprisingly, this support does not have a large impact on performance—only for the *max-coverage* version of GZIP is the improvement substantial. Next we measure the impact of instant violation notification through interrupts both in isolation (*F*) and also when combined with local storage for initial state (*G*). For the *select* versions, five applications are improved significantly by interrupt violation detection, but do not improve further with the local storage for initial state. For seven of the *max-coverage* benchmarks the improvement is dramatic, likely due to the large amount of failed speculation in this version of the applications. This result indicates that interrupt-based violation notification is important for limiting the negative performance impact of frequent failed speculation. However, we have not included this support as part our baseline since its complexity is non-trivial, requiring a new form of interrupt and support for restoring initial state when recovering from failed speculation.

Finally, recall from Section 3.5.2 that one purpose of speculative coherence messages is to provide early notification of violations. In the final experiment (*E*), we further modify the previous experiment (*G*) by eliding speculative messages from our coherence scheme. For the *select* benchmarks we observe that most of the benefit of early detection is lost when speculative coherence messages are unsupported. The performance of the *max-coverage* benchmarks also suffers, but to a lesser extent. Hence interrupt-based violation detection and speculative coherence messages are only beneficial when implemented in tandem.

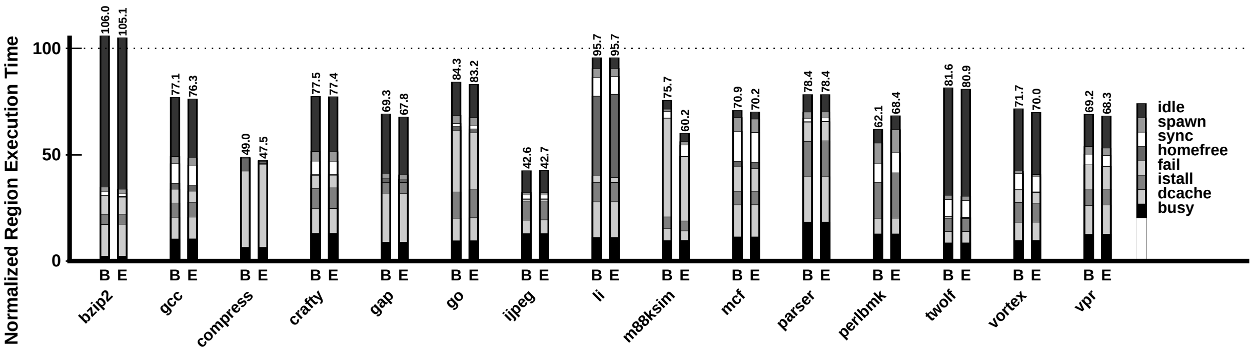


Figure 3.32. Benefits of a hardware-visible homefree token. B **(our baseline) models a software-only homefree token, while** E **models a hardware-visible homefree token.**

3.7.5 Implementation of the Homefree Token

Another important design issue is whether the homefree token should be visible to hardware, or simply be implemented with normal software-based synchronization in regular memory. More specifically, a *hardware-visible* homefree token implies that hardware is aware of the arrival of the homefree token and at that point can begin committing the current epoch's speculative modifications to memory, further hiding the latency of the commit operation. Figure 3.32 quantifies the performance impact of a hardware-visible homefree token. This feature will only be beneficial if the homefree token frequently arrives before the current epoch is complete—allowing an early commit to effectively hide the latency of committing speculative modifications to memory. This technique provides a significant benefit for M88KSIM, and provides a slight improvement for ten other applications. For COMPRESS, this technique eliminates all time spent waiting for the homefree token to arrive. For the other applications that improve, the improvement is mainly in reduced failed speculation—since speculative modifications are being committed to memory earlier, there are fewer violated data dependences between epochs and hence less failed speculation in those cases. Only one other application (PERLBMK) performs moderately worse due to an increase in instruction stall time (*istall*).

3.8 Chapter Summary

We have introduced a speculative cache coherence scheme that allows the compiler to automatically parallelize general-purpose applications and to exploit an arbitrary number of processors on a single chip. Our approach extends the architecture of a generic chip-multiprocessor without adding any large or centralized TLS-specific structures, and without hindering the performance of non-speculative workloads. Of 15 benchmark applications, our baseline architecture and coherence scheme improves program performance for one application by two-fold, for three other applications by more than 10%, and provides more modest improvements for four other applications. A deep analysis of our scheme shows that our implementation of TLS support is efficient, and that our mechanisms for supporting speculation are not a bottleneck.

A closer look at our hardware support and speculative coherence scheme resulted in many important observations. We found that support for multiple writers is necessary for good performance for most general-purpose applications studied, and that a simplified version of the coherence scheme without speculative coherence messages is nearly as effective as the original. Analyzing the sensitivity of our scheme to various architectural parameters, we found an expensive inter-processor communication mechanism to be unnecessary so long as a less-expensive mechanism with a latency of no more than 20 cycles can be implemented. We also discovered that TLS execution is sensitive to neither crossbar bandwidth nor the number of data reference handlers, indicating that the the memory system is not a bottleneck. However, varying the sizes of the data caches demonstrated that 8KB caches are insufficient, although 64KB caches do not offer a significant improvement over 32KB caches. Finally, we showed that TLS techniques are complementary to out-of-order superscalar techniques, and that performance benefits of control independence enjoyed by TLS cannot be achieved simply by increasing the reorder buffer size of a uniprocessor.

We explored alternative designs for many aspects of our TLS hardware support. We showed that less aggressive designs, namely deep uniprocessor speculation and TLS support using only the load/store-queues, are insufficient to capture the same performance improvements as our approach. We demonstrated that it is sufficient to allocate the forwarding frame in regular memory and having it reside in the first-level data caches, rather than adding new local or shared register files. While examining various violation notification techniques, we discovered that interrupt-based violation notification is important for exploiting TLS in applications that do not actually contain much parallelism, and that this support increases the importance of speculative coherence messages. Finally, we demonstrated that a software-only implementation of the homefree token is sufficient, although a hardware-visible homefree token does yield additional benefit.

In the next chapter, we will evaluate the ability of our speculative coherence scheme to scale both down and up: from a chip-multiprocessor where the first-level data cache is shared, to large machines that use many chip-multiprocessors as building blocks.

Chapter 4

Support for Scalable Thread-Level Speculation

4.1 Introduction

The previous chapter introduced a speculative coherence scheme that empowers the compiler to automatically-parallelize general-purpose programs to exploit chip-multiprocessors. This implementation scales well within a chip from two to at least eight processors; performance is primarily limited by the amount of parallelism that is actually available in the benchmark applications. However, our original goal was to support TLS in *any* scale of machine. Since our scheme is built on top of standard invalidation-based cache coherence, it scales both up to large-scale multiprocessors as well as down to multithreaded processors. Given this scalable foundation for supporting TLS execution, what are the key issues for improving the efficiency of speculative execution at these different scales?

The goals of this chapter are twofold. First, we will evaluate how well our cache coherence scheme scales down by evaluating its performance within a chip for processors that share a cache; we also explore possibilities for enhancing the performance of TLS with these architectures. Second, we evaluate how well our cache coherence scheme scales up to high-end multiprocessor systems (e.g., the SGI Origin [47]) composed of traditional processors or perhaps using chip-multiprocessors as building blocks. Similarly, we explore ways to improve the performance of TLS for these larger-scale machines.

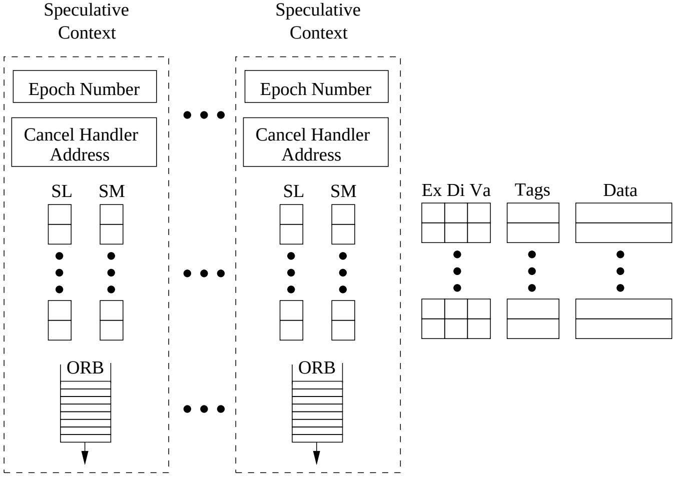


Figure 4.1. Hardware support for multiple epoch contexts in a single cache.

4.2 Support for a Shared Data Cache

In this section we describe and evaluate support for TLS in a shared data cache. This support for multiple speculative contexts within a single cache is valuable for three reasons. First, support for multiple speculative contexts allows us to implement TLS with *simultaneous multithreading* (SMT) [70] and other shared-cache multithreaded architectures. Second, we can use multiple speculative contexts to allow a single processor to switch to a new epoch when the current epoch is suspended (e.g., when waiting for the homefree token). Finally, we may want to maintain speculative state across OS-level context switches so that we can support TLS in a multiprogramming environment.[1]

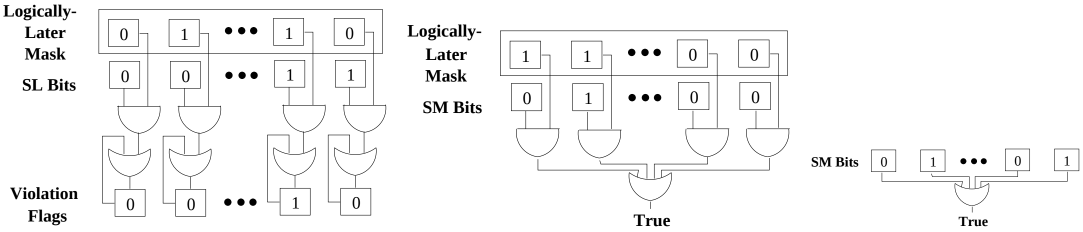
We begin by describing how our implementation of speculative state from Section 3.3 can be extended to support multiple speculative contexts. We then evaluate this support and then explore ways to avoid failed speculation due to conflicts in the shared cache.

4.2.1 Implementation

In our basic coherence scheme, two epochs from the same program may both access the same cache line except in two cases: (i) two epochs must not modify the same cache line, and (ii) an epoch must not read from a cache line that has been speculatively-modified by a logically-later epoch. We can trivially enforce these constraints by simply squashing the logically-later epoch whenever a constraint is about to be violated.

Figure 4.1 shows how we can support TLS in a shared cache by implementing multiple speculative contexts. The

[1] For now we assume that any system interrupt will cause all speculation to fail—evaluation of OS-level context-switching is beyond the scope of this thesis.



(a) Data dependence violation detection.

(b) Read-conflict detection.

(c) Write-conflict detection.

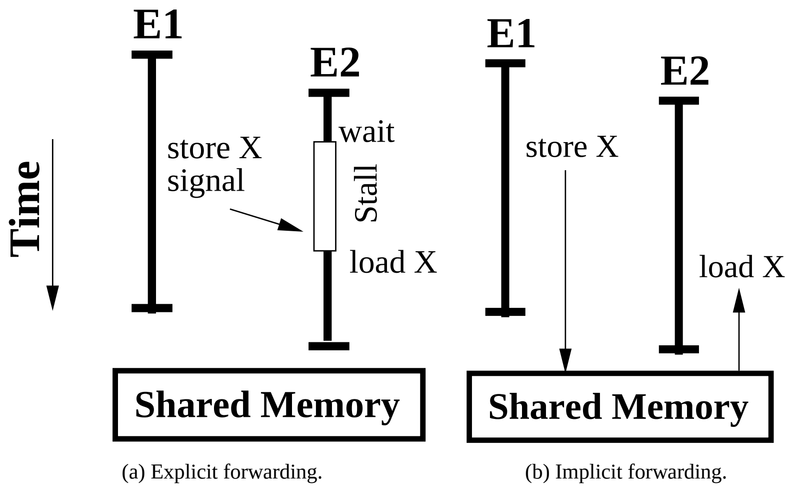
Figure 4.2. Support for efficient epoch number comparison.

exclusive (*Ex*), dirty (*Di*), and valid (*Va*) bits for each cache line are shared between all speculative contexts, but each speculative context has its own speculatively-loaded (*SL*) and speculatively-modified (*SM*) bits. This state allows us to track which epochs have speculatively loaded or modified any cache line, and allows us to track data dependences between epochs as well as check for conflicts.

Since epoch contexts in a shared-cache architecture are implemented in a common structure (as opposed to the distributed implementation for private caches), it is wasteful to frequently recompute their relative ordering by comparing epoch numbers on every memory reference. Instead we can *precompute* and store the relative ordering between all active local epochs. A convenient method of storing this information is in a *logically-later mask*. This mask has one bit per local speculative context, and each bit is set if the corresponding speculative context is currently executing an epoch that is *logically-later* than the epoch in question. We maintain a logically-later mask for each speculative context and update it whenever an epoch is spawned or exits.

As shown in Figure 4.2(a), we can use the *logically-later mask* to detect data dependence violations. If the active epoch stores to a location, then any logically-later epoch which has already speculatively loaded that same location has committed a violation. We can detect a violation by taking the bit-wise `AND` of the logically-later mask with the *SL* bits for the appropriate cache line and `OR`'ing the result with the violation flag for each epoch.

In addition to detection of data dependence violations, we also need to track read and write conflicts in the shared cache. A conflict occurs when two epochs access the same cache line in an incompatible way, and is resolved by squashing the logically-later epoch (a *conflict violation*). For our initial shared-cache implementation, the following two access patterns are incompatible (these are formalized in the full description of our speculative coherence scheme given in Appendix A).



(a) Explicit forwarding.

(b) Implicit forwarding.

Figure 4.3. Explicit vs implicit forwarding.

1. If an epoch speculatively modifies a cache line, only that epoch or a logically-later epoch may read that cache line afterwards. If a logically-earlier epoch attempts to read the cache line, a read-conflict violation results.

2. Only one epoch may speculatively modify a given cache line. If an epoch attempts to speculatively modify a cache line that has already been speculatively modified by a different epoch, a write-conflict violation results.

We can use the logically-later masks to determine whether any load or store will result in a read or write-conflict violation, as illustrated in Figures 4.2(b) and 4.2(c). Recall that a read-conflict miss occurs when the active epoch attempts to execute a speculative load but a logically-later epoch has already modified that same cache line. This condition may be checked by taking the `AND` of the logically-later mask with the *speculatively-modified* (*SM*) bits for the appropriate cache line and checking the `OR` of the results. A write-conflict miss occurs when the active epoch executes a speculative store and any other epoch has already speculatively stored that cache line. We can check for this case by taking the `OR` of the speculatively-modified (*SM*) bits (excluding the bit belonging to the epoch in question) as shown in Figure 4.2(b), and proceed by squashing the logically-later epoch.

The data dependence tracking implemented by our shared-cache support for TLS differs from the private-cache support described in Chapter 3 in the following two ways. First, our shared-cache support allows us to *implicitly forward* speculative modifications between two properly ordered epochs. In Figure 4.3, we differentiate between explicit and implicit forwarding. With explicit forwarding, as is supported in our private-cache scheme, the compiler inserts explicit `wait` and `signal` primitives which communicate a value between epochs through the *forwarding* *frame*. In contrast, *implicit forwarding* (which is not supported in our private-cache scheme) allows a value to be communicated between a store from one epoch and a load from a logically-later epoch that happen to execute in order. Our private-cache design supports only explicit forwarding because its distributed nature makes implicit forwarding extremely difficult to implement.[2] However, implicit forwarding is trivial to support in a shared cache: we simply allow an epoch to speculatively load from a cache line that has been speculatively-modified by a logically-earlier epoch. Note that if the logically-earlier epoch then speculatively modifies that cache line again, a write-conflict violation will result. Since support for implicit forwarding is trivial to implement in a shared cache, we include implicit forwarding in our baseline design.

The second major difference between shared and private-cache architectures is that we cannot easily allow two epochs to modify the same cache line—in this respect, our shared-cache design is less aggressive than our private-cache design which has such support for *multiple writers* (see Section 3.5.1). We will describe how our baseline shared-cache design can be extended to support multiple writers in Section 4.2.3.

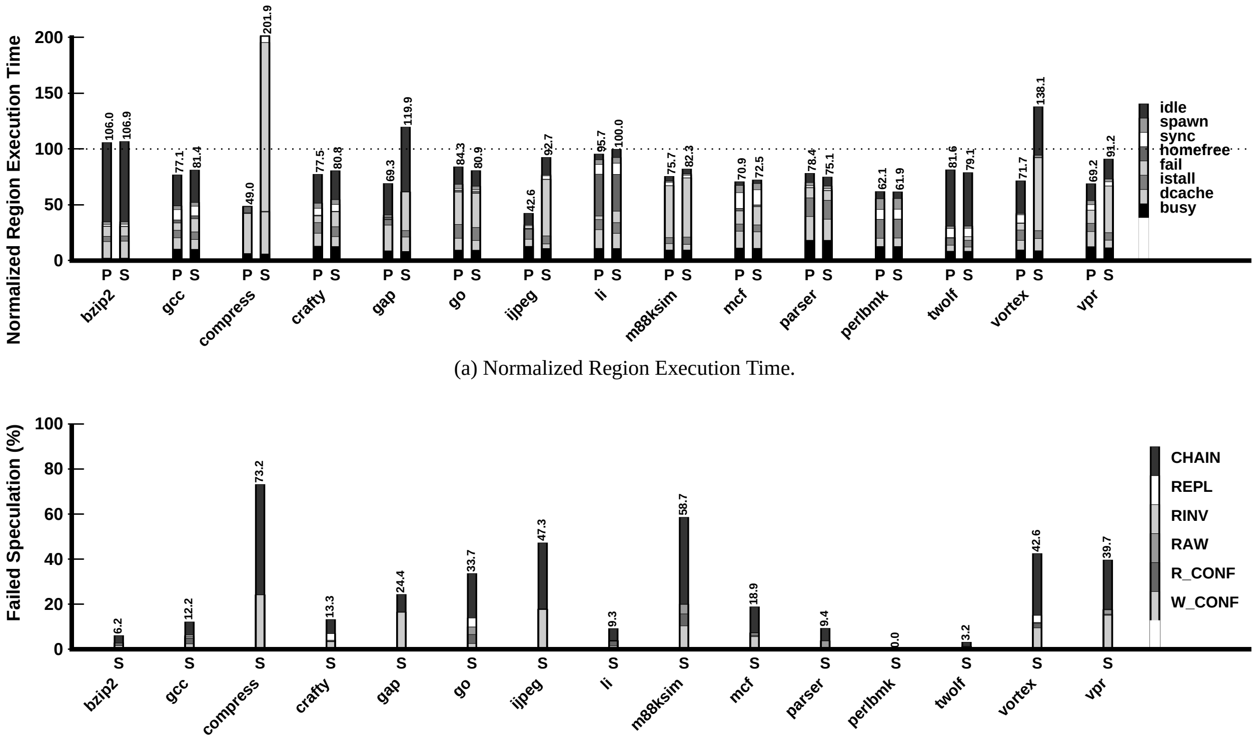
As we will demonstrate next, these simple extensions provide effective support for TLS in shared-cache architectures.

4.2.2 Performance of Shared Data Cache Support for TLS

We begin our evaluation by comparing the performance of both private-cache and shared-cache support for TLS, as shown in Figure 4.4(a). *P* shows speculative execution on a 4-processor CMP with private caches, and *S* shows speculative execution on a 4-processor CMP with a shared first-level data cache. To facilitate comparison, the shared cache is the same size and associativity as one of the private caches (32KB, 2-way set associative). For 10 of the 15 applications, the performance with a shared cache is similar to the performance with private caches; the remaining 5 applications (COMPRESS, GAP, IJPEG, VORTEX, and VPR) perform significantly worse with a shared cache due to increased failed speculation.

Figure 4.4(b) shows the percentage of time wasted on failed speculation for each application, broken down into the reasons why speculation failed. The first three segments are similar to those from the breakdown for private-cache architectures (e.g., Figure 3.12): *CHAIN* violations represent time spent on epochs that were squashed because a

[2] Speculative modifications would have to be broadcast to all logically-later epochs, or an epoch would have to poll the caches of all logically-earlier epochs for the most up-to-date value on every load



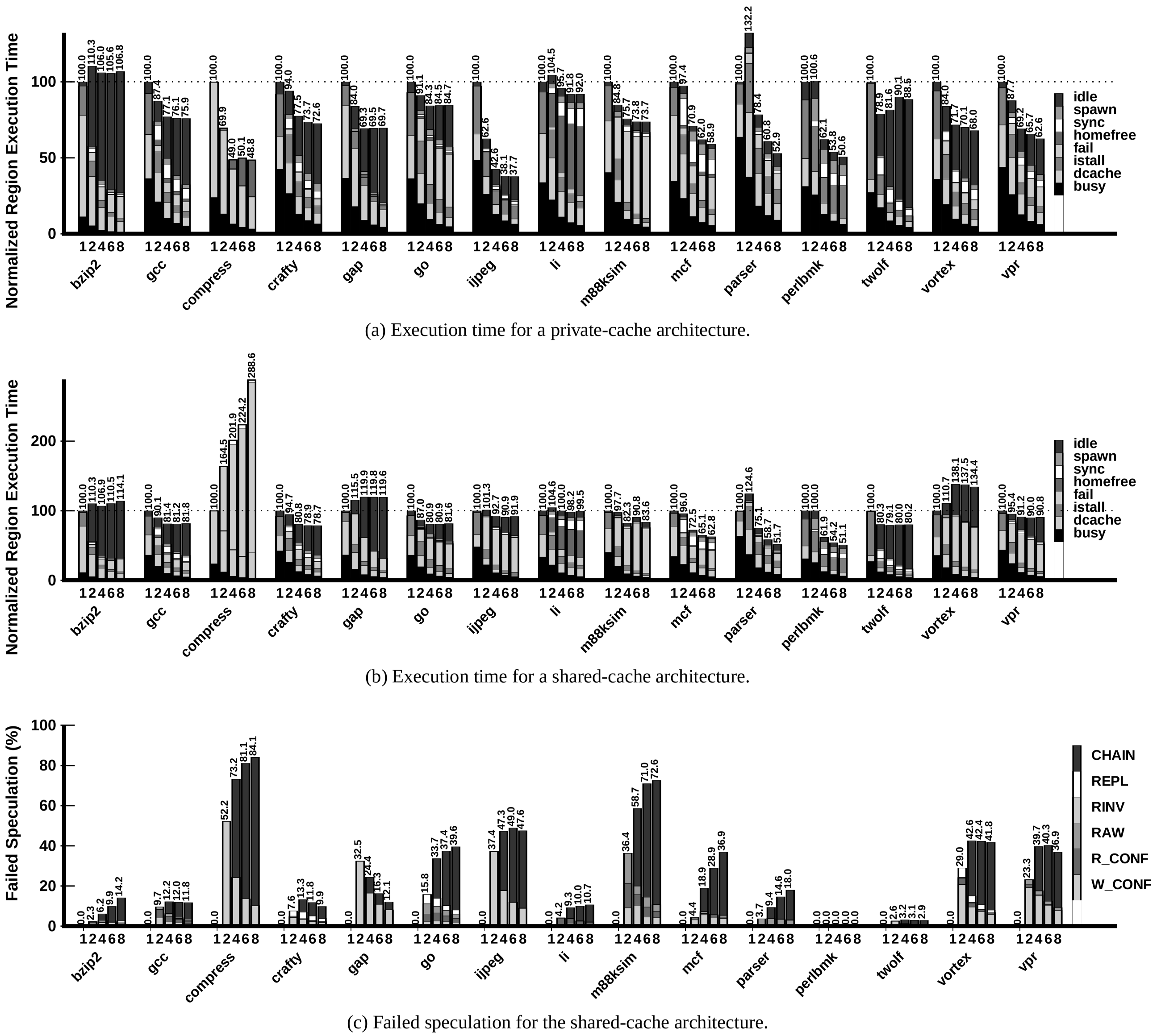
(a) Normalized Region Execution Time.

(b) Percentage of time wasted on failed speculation for a shared-cache.

Figure 4.4. Region performance of the *select* benchmarks on both private-cache and shared-cache architectures. *P* is speculatively executed on a 4-processor CMP with private caches, and *S* is speculatively executed on a 4-processor CMP with a shared cache.

logically-earlier epoch was previously squashed, while (*REPL* and *RINV*) represent violations caused by replacement in either the first-level data caches or the shared unified cache respectively.[3] The remaining segments represent violations raised by the new shared-cache TLS mechanisms: the *RAW* segment represents read-after-write data dependence violations, and the *R_CONF* and *W_CONF* segments represent read and write conflicts respectively. It is apparent that write conflicts (*W_CONF*) account for the vast majority of the failed speculation in the shared-cache TLS support. Even though the set-associative first-level data cache is only 2-way set-associative, CRAFTY, GO, and VORTEX are the only three applications for which replacement (*REPL*) is a significant component of time lost to failed speculation.

For most applications, our baseline shared-cache hardware support is sufficient to maintain the performance of private caches; however, for several applications the impact of conflict violations is severe. We will investigate ways to tolerate these conflicts later in Section 4.2.3.



(a) Execution time for a private-cache architecture.

(b) Execution time for a shared-cache architecture.

(c) Failed speculation for the shared-cache architecture.

Figure 4.5. Varying the number of processors for both private and shared-cache architectures. Note that the shared cache is the same size as each of the private caches (32KB).

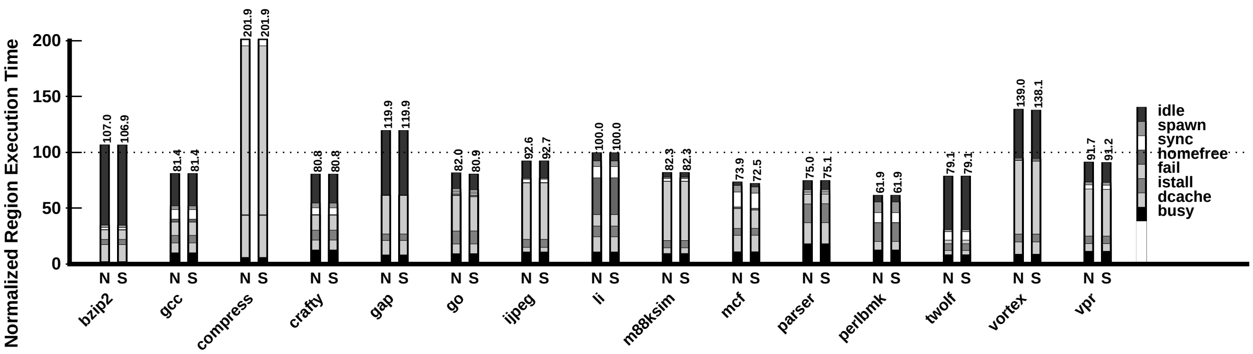


Figure 4.6. Benefits of implicit forwarding in a shared cache. *N* **does not support implicit forwarding, while** *S* **(our shared-cache baseline) does.**

Scaling Within a Chip for a Shared-Cache Architecture

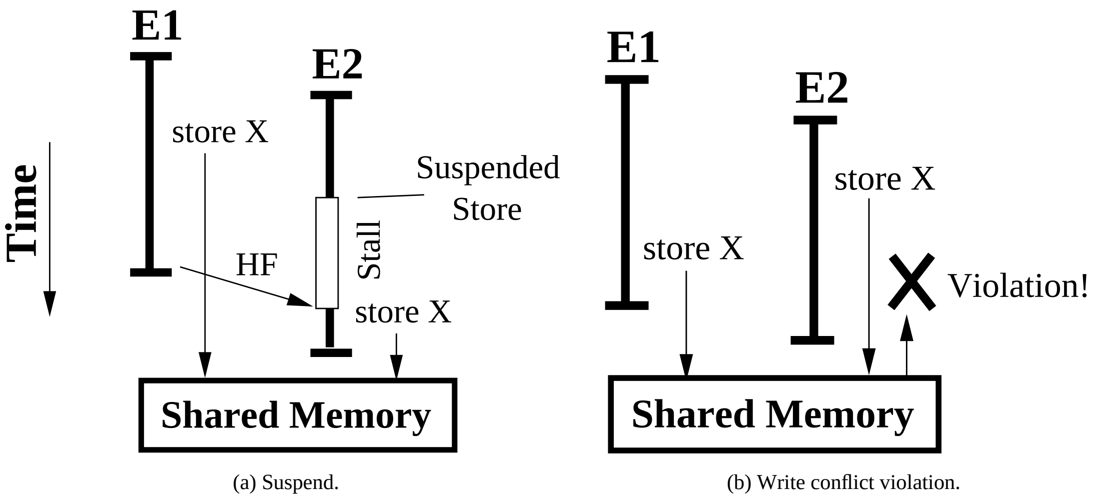
Next we compare how both private-cache and shared-cache architectures scale within a chip by varying the number of processors from 2 to 8, as shown in Figure 4.5. The scaling behavior of the benchmarks is similar for the two different architectures except for COMPRESS, GAP, IJPEG, VORTEX and VPR—for these applications failed speculation prevents scaling for shared-cache architectures. To investigate further, Figure 4.5(c) shows the percentage of time wasted on failed speculation for the shared-cache design. Again, we observe that write conflicts (*W_CONF*) are the main cause of failed speculation for most applications—we will further investigate conflicts in Section 4.2.3.

Impact of Implicit Forwarding

In Section 3.1.2 we estimated that support for implicit forwarding was not worth the implementation complexity for private-cache architectures. For shared-cache designs, providing this support is relatively straightforward since the speculative state is implemented in a common structure (as opposed to the distributed implementation for private caches). However, it is interesting to quantify the benefits of such support. In the shared-cache experiments in Figure 4.6, the *N* experiment does not include support for implicit forwarding while the *S* experiment (our shared-cache baseline) does. It is apparent that support for implicit forwarding does not have a significant impact on performance other than for MCF, which improves slightly due to a decrease in failed speculation. Hence our estimate that support for implicit forwarding is not worth the complexity was correct.

We have shown how to implement support for TLS in a shared data cache architecture , and that performance for

[3] Recall that we do not need special support to choose which cache line to evict from an associative set: the existing LRU (least recently used) mechanism ensures that any non-speculative cache line is evicted before a speculative one.



(a) Suspend.

(b) Write conflict violation.

Figure 4.7. Two epochs that store the same cache line. In (a), suspension of epoch 2 allows it to proceed later. In (b), suspension cannot help, and epoch 2 is violated due to the write conflict.

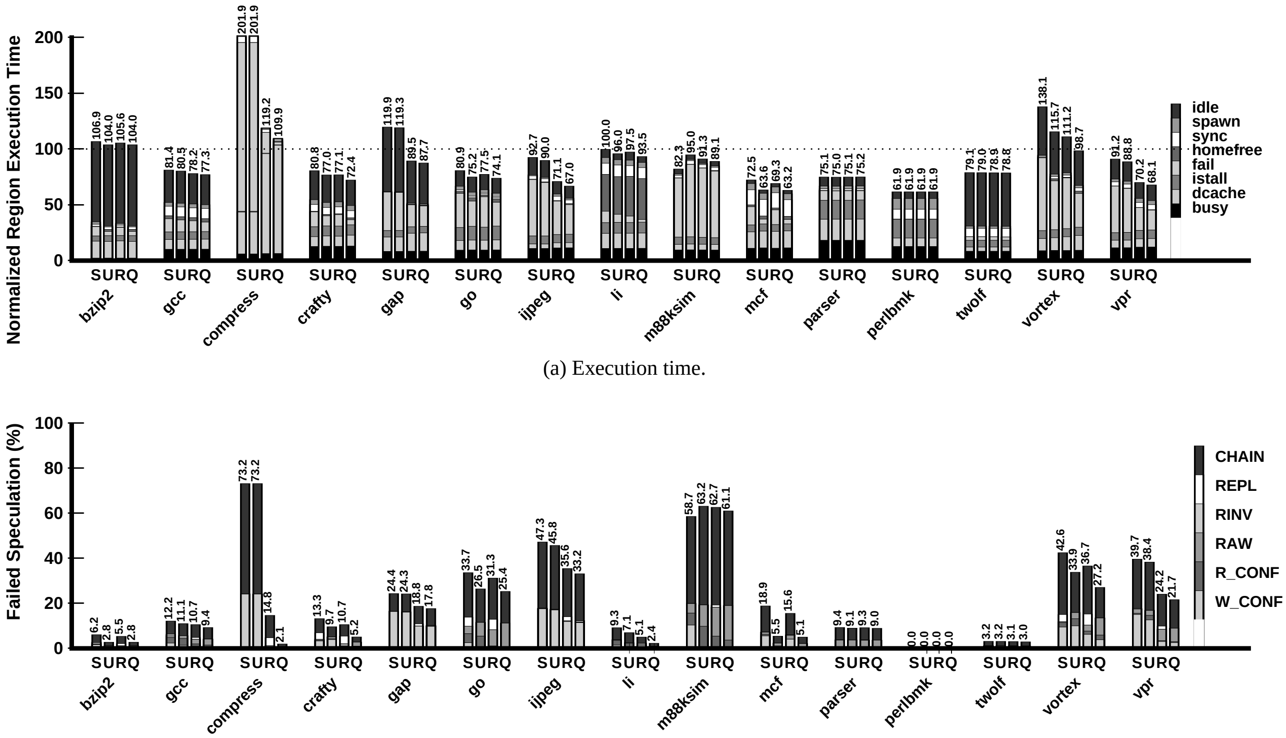
most applications is comparable to that of a private-cache architecture, although generally not as good. We observed that an increase in failed speculation due to read and write conflicts negates the potential performance improvement from the increased locality of the shared cache architecture (as compared with the private-cache architecture); hence we next investigate ways to tolerate these conflicts.

4.2.3 Tolerating Read and Write Conflicts

In this section, we evaluate two methods for tolerating read and write conflicts in a shared-cache design. First, we investigate support for *suspending* an epoch that is about to cause a conflict. Second, we evaluate support for *cache line replication*; this support is more costly but allows speculative execution to proceed. Third, we analyze the performance of these two techniques when combined. Finally, we measure the impact of increasing the associativity of the shared cache.

Suspending Epochs

Rather than handling read or write conflicts by squashing the logically-later epoch, we can instead *suspend* that epoch until it becomes homefree. Only in certain cases can an epoch be suspended. For example, consider two epochs that both attempt to write to the same cache line as shown in Figure 4.7. In Figure 4.7(a) *epoch 1* (E1) writes first, then *epoch 2* (E2) is suspended when it attempts to write until it is passed the homefree token, at which point it can proceed. In contrast, in Figure 4.7(b) *epoch 2* writes first, and when *epoch 1* writes a a write conflict is triggered and



(a) Execution time.

(b) Failed speculation.

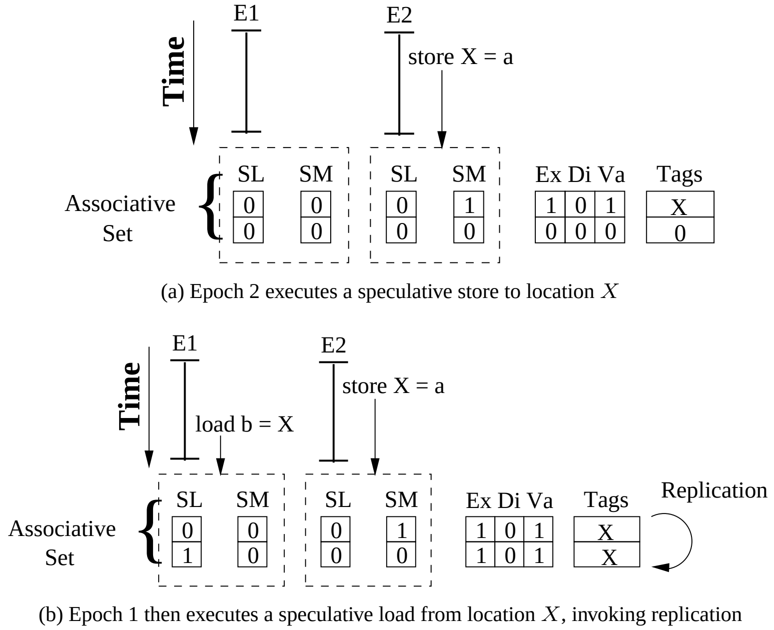
Figure 4.8. Impact of suspending violations for replacement and conflicts. S **is the baseline 4-processor shared-cache architecture,** U **builds on** S **by tolerating conflicts and replacement through suspension of the logically-later epoch,** R **builds on** S **by tolerating conflicts through replication, and** Q **supports both suspension and replication.**

epoch 2 is squashed. There are two requirements to avoid deadlock when suspending an epoch: (i) that exactly one epoch is always homefree, and (ii) any suspended epoch that receives the homefree token is unsuspended at that point.

In Figure 4.8, we evaluate performance when both replacement and certain read/write conflicts cause the logically-later epoch to be suspended rather than squashed. We show performance on our shared-cache baseline architecture (*S*), and an augmented baseline where epochs are suspended rather than squashed whenever possible (*U*). Suspension eliminates a significant amount of failed speculation due to replacement and conflicts for several applications, including BZIP2, CRAFTY, GO, MCF, and VORTEX;the resulting improvement in performance is significant for GO, MCF, and VORTEX. Only M88KSIM performs worse: write-conflict violations have merely exposed read-conflict and read-after-write (*RAW*) violations. Overall, this support is worthwhile and also straightforward to implement.

Cache Line Replication

Another technique for tolerating read and write conflicts is cache line replication. Rather than squashing the conflicting epoch, the epoch can proceed by replicating the appropriate cache line: if the cache line is not yet speculatively-



(a) Epoch 2 executes a speculative store to location X

(b) Epoch 1 then executes a speculative load from location X, invoking replication

Figure 4.9. Example of cache line replication.

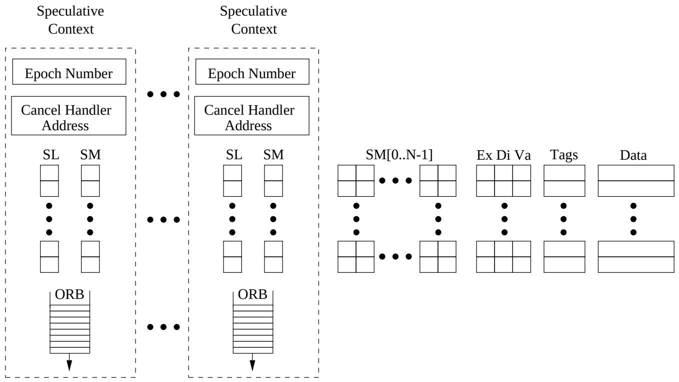


Figure 4.10. Hardware support for multiple writers in a shared cache that also supports replication.

modified, then it may be copied directly; if the cache line is speculatively-modified, then the replicated copy is obtained from the external memory system. Once replicated, both copies of the cache line are kept in the same associative set of the shared cache. The owner of a given cache line can be determined by checking the *SM* and *SL* bits—in other words, the *SM* and *SL* bits are considered part of the tag match. If all entries in an associative set are consumed, then replication fails and the logically-latest epoch owning a cache line in that set is suspended or squashed.

Figure 4.9 shows an example of cache line replication. In the figure, only the speculative state of the appropriate associative set is shown. In Figure 4.9(a), *epoch 2* (E2) executes a speculative store to location X, and the *SM* bit for that cache line is set. Next, in Figure 4.9(b), *epoch 1* executes a speculative load from the same location (X) resulting in a read conflict (since *epoch 1* must not read the speculative modifications of *epoch 2*). Rather than squashing *epoch 2*, we can use the available entry in the associative set to store a replicated copy of location X. Once replication is supported, we can also extend our implementation to support multiple writers by adding *fine-grain SM* bits [66]—as shown in Figure 4.10, only one group of fine-grain SM bits is necessary per cache line since only one epoch may modify a given cache line.

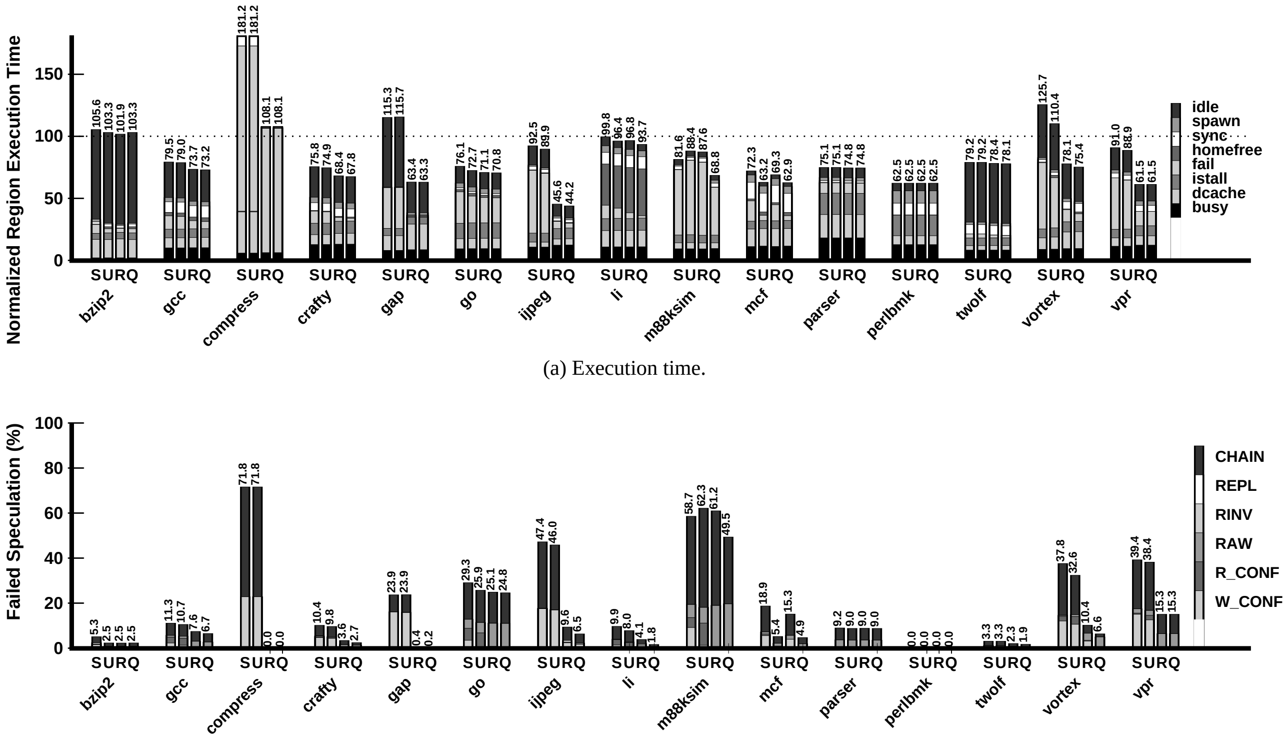
In Figure 4.8 the *R* experiment builds on the baseline *S* with support for cache line replication. This support has a significant positive impact on the performance of, GAP, IJPEG, VORTEX, VPR, and especially on COMPRESS for which performance improves by 82.7%. In all of these cases the amount of failed speculation has been significantly reduced. From Figure 4.8(b), which shows the percentage of execution time wasted on failed speculation and the corresponding breakdown, we see that replication does increase tolerance of write conflicts (*W_CONF*) and read conflicts (*R_CONF*). Reducing the occurrence of write conflicts significantly lowers the amount of failed speculation for COMPRESS, CRAFTY, GAP, IJPEG, LI, VORTEX and VPR, but merely exposes more true data dependence violations (*RAW*) for GO and M88KSIM. Although this support is relatively costly to implement (since we need to be able to store multiple versions of the same cache line in a single associative set), the benefits are significant.

Combining the Techniques

In the *Q* experiment in Figure 4.8, we evaluate the combination of both the suspension of epochs and cache line replication. Although replication (*R*) is more effective than suspension (*U*) for most benchmarks, compared with either technique in isolation we observe that the combination of the two techniques (*Q*) captures the best performance of either in every case. Furthermore, for 7 applications (COMPRESS, CRAFTY, GAP, IJPEG, LI, VORTEX, and VPR) this combination is complementary, achieving better performance than either technique alone.

4.2.4 Impact of Increasing Associativity

Increased associativity is usually desirable for shared-cache architectures, although there is a point where the increase in hit latency negates further benefit. Hence we want to ensure that our scheme for supporting TLS in a shared cache can also benefit from increased associativity—in particular, whether support for cache line replication can capitalize on the increased opportunity for storing replicated copies. In Figure 4.11 we repeat the experiments for suspension and replication (shown in Figure 4.8) for a shared cache with an associativity of 4 ways (as opposed



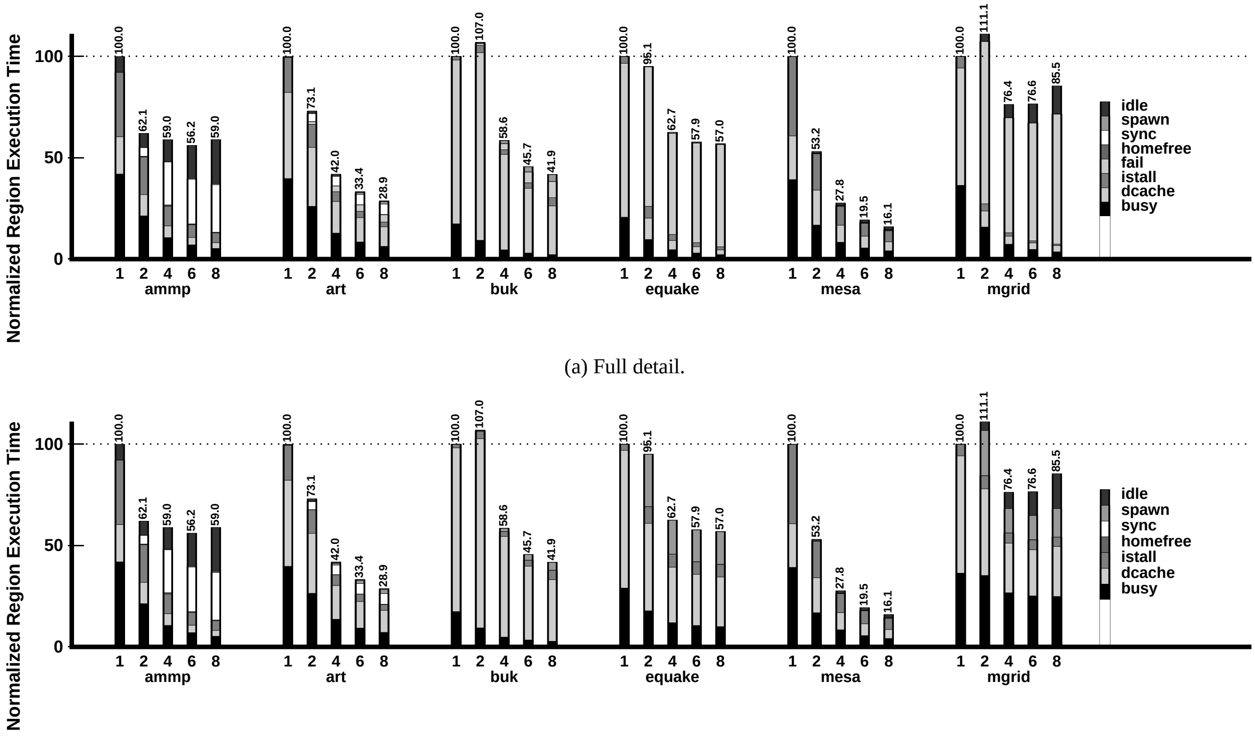
(a) Execution time.

(b) Failed speculation.

Figure 4.11. Impact of suspending violations for replacement and conflicts when the the shared data cache is 4-way set-associativity (as opposed to 2-ways). S is the baseline 4-processor shared-cache architecture, U builds on S by tolerating conflicts and replacement through suspension of the logically-later epoch, R builds on S by tolerating conflicts through replication, and Q supports both suspension and replication.

to 2 ways, as used until this point): we maintain the original hit latency, and re-evaluate the sequential execution (to which all experiments are normalized) on a 4-way set-associative cache as well. Comparing with Figure 4.8, we observe that the performance of the baseline (S) with increased associativity is improved in most cases. Suspension and replacement are even more effective at tolerating conflicts and reducing failed speculation for most applications. With support for both suspension and replication (Q), failed speculation is nearly or entirely eliminated for COMPRESS and GAP. These results confirm that our shared-cache support for TLS can benefit from increased associativity.

We now understand how to support TLS effectively within a single chip, whether the underlying architecture is a chip-multiprocessor where each speculative thread has a corresponding private first-level data cache, or a chip-multiprocessor or simultaneously-multithreaded processor where threads share a cache. We now turn our attention to the problem of supporting TLS on larger scale machines composed of multiple processor chips, where each chip itself may have one or more processors.



(a) Full detail.

(b) Incorporating failed speculation.

Figure 4.12. Region performance of the *select* **version of the floating point benchmarks when scaling (varying the number of processors) within a chip.**

4.3 Scaling Beyond Chip Boundaries

Our scheme as described in Chapter 3 provides a framework for scaling to large machines since it is built upon invalidation-based cache coherence which itself works on a wide-variety of distributed architectures. In this section we evaluate the ability of our coherence scheme for TLS to scale up to multiprocessor systems composed of chip-multiprocessors.

4.3.1 Performance of Floating Point Benchmark Applications

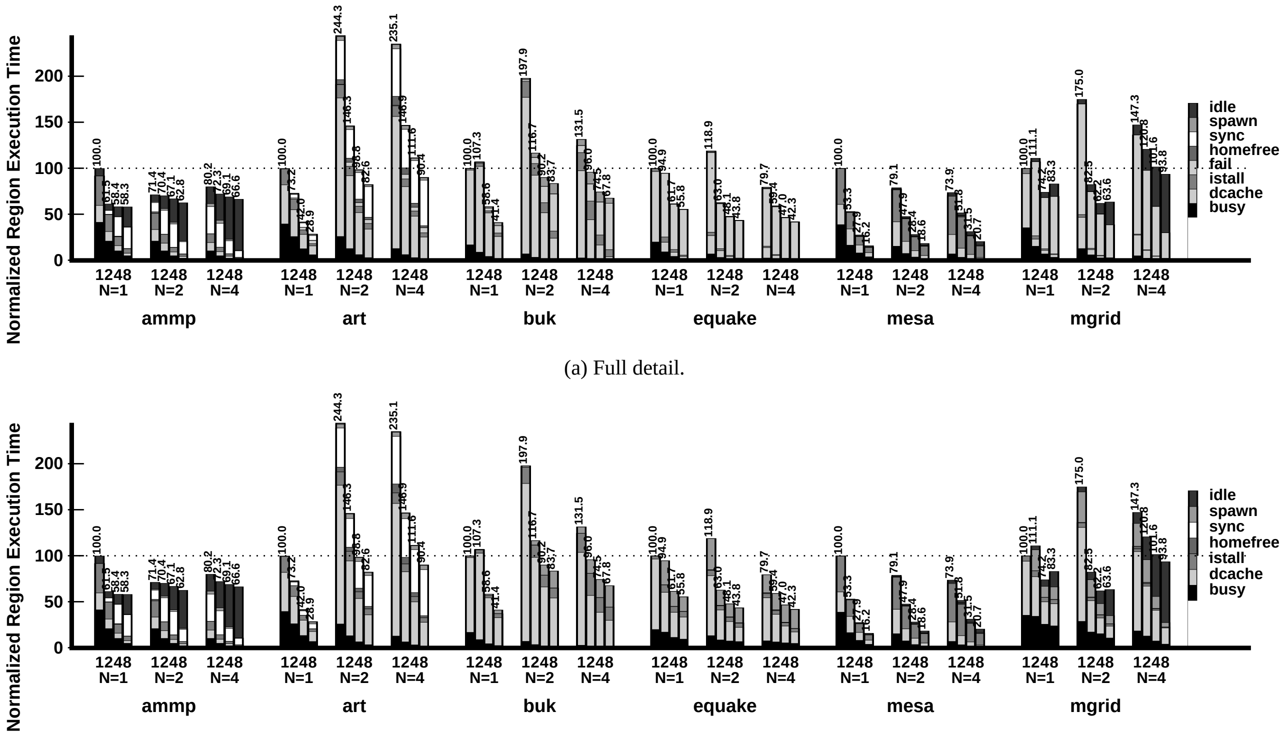
In Chapter 3 we observed that our support for TLS allows many speculatively-parallelized general-purpose applications to scale within a single chip. However, these programs generally do not exhibit the coarse-grain parallelism required to scale well beyond chip boundaries. To evaluate the performance of our hardware support for TLS when scaling beyond a single chip, we use the SPECfp2000 [16] floating point benchmarks described and analyzed in Section 2.5.1. Compared with general-purpose codes, these scientific applications have greater available parallelism and can potentially scale well beyond a single chip. They also tend to have more computation and less complex control flow than general-purpose applications, allowing parallel execution to hide more latency. To the best of our knowledge, floating point benchmarks are used in the evaluation of all related work on scalable support for TLS [12, 24, 32, 59, 60, 63, 77, 79].

Before we evaluate the performance of multi-chip architectures, we first measure the scalability of the floating point applications within a single chip. Figure 4.12 shows region execution time as we vary the number of processors from one to eight. ART, BUK, and MESA scale very well within a single chip showing continually improved performance as we increase the number of processors; ART suffers from both failed speculation and synchronization overheads, and BUK is mostly limited by data cache misses (*dcache*). While AMMP does not suffer from failed speculation, the increasing synchronization (*sync*) and idle segments indicate limited available parallelism for that application. Both EQUAKE and MGRID suffer from large amounts of failed speculation; while speedup for EQUAKE reaches a limit at 6 processors, MGRID actually performs worse beyond 4 processors. While not all of these applications are well suited to multi-chip execution, they demonstrate a wide variety of behaviors that will aid in a thorough evaluation of the scalability of our hardware support.

4.3.2 Scaling Up to Multi-Node Architectures

Given a fixed total number of processors, there are both advantages and disadvantages to splitting those processors across multiple nodes. One advantage is that the total amount of secondary cache storage increases (since there is a fixed amount per chip). On the other hand, disadvantages include an increase in the average cost of inter-processor communication, and decreased data locality. Figure 4.13 shows the region performance of the floating point benchmarks on multiprocessor architectures with varying numbers of processors and nodes. For each benchmark we simulate 1, 2, and 4 nodes (N) with a varying number of processors per node. Each bar is normalized to the execution time of the the sequential version (1 node with 1 processor). Each chip-multiprocessor node is part of a cache-coherent non-uniform memory access (CC-NUMA), distributed shared memory (DSM) mesh network with 2 kilobyte pages that are distributed in a round-robin fashion across the memories in the DSM network. The communication latency between nodes for all speculative events (spawning an epoch, explicit forwarding, passing the homefree token) is 100 processor cycles. All other configuration parameters are identical to those described in Table 2.6.

The results for a single node are similar to those from Figure 4.12; any slight differences are due to the addition



(a) Full detail.

(b) Incorporating failed speculation.

Figure 4.13. Region performance of the *select* version of the floating point benchmarks on multiprocessor architectures with varying numbers of processors and nodes. For each benchmark we simulate 1, 2, and 4 nodes (N) of 1, 2, 4, and 8 processors per node.

of the DSM model. For architectures with two nodes and one processor per node, only AMMP and MESA speed up, indicating that only certain applications—when speculatively parallelized—can exploit conventional multiprocessors possessing only one processor per node. As we increase the number of processors for two-node architectures, performance improves for all six applications; only with 8 processors does ART speed up on a two-node machine.

Performance on four-node architectures is similar to that of two-node architectures for all applications except for MGRID, which no longer speeds up due to an overwhelming amount of failed speculation and idle time. In most cases, the four-node machines do not perform as well as machines with fewer nodes. However, these results indicate that applications respond well to an increase in the number of processors per node: to exploit speculative parallelism across multiple chips, we need a certain amount of parallelism per node to tolerate the high inter-node communication latencies. For EQUAKE, the best overall performance is achieved with a multi-node architecture (4 nodes of 8 processors). For a fixed number of processors per node, both MESA and MGRID benefit from the addition of a second node. This result is important, since current chip multiprocessors such as the IBM Power4 [38] can be packaged in clusters of

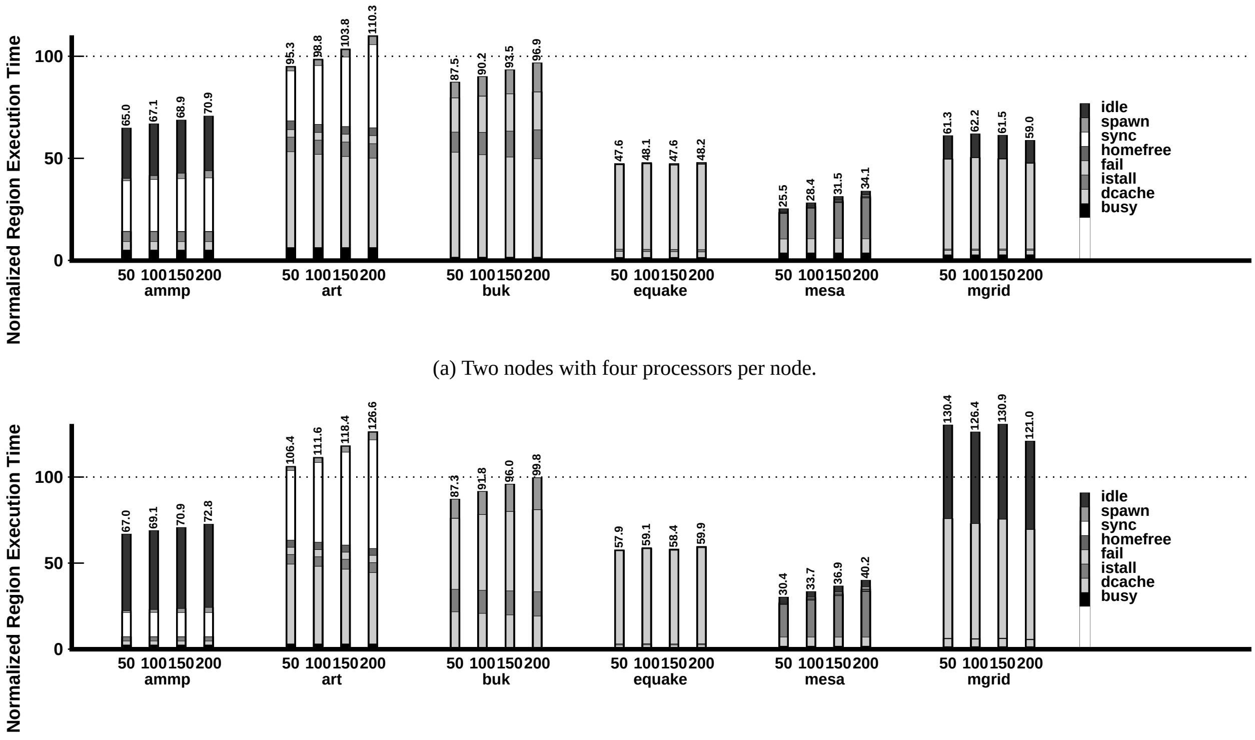
2-processor nodes (chips) with up to 4 nodes incorporated in a multi-chip module.

Several factors limit parallelism for these applications. First, AMMP and MGRID only have 11.2 and 18.2 epochs per region instance on average (see Table 2.3); this means that they will not benefit from additional processors beyond 11 and 18 respectively. As evidence of this, both applications exhibit an increasing amount of idle time when the number of processors is greater than the average number of epochs per region instance. Second, AMMP and ART spend a significant amount of time stalled on synchronization (*sync*), while BUK,EQUAKE, and MGRID suffer from large amounts of failed speculation (*fail*). Third, ART has the most difficulty tolerating inter-chip latency since it has relatively small epochs (176.4 instructions per epoch on average, as seen earlier in Table 2.6) compared to the other applications (MESA has the second smallest epochs, with 291.4 instructions per epoch on average); ART may benefit from unrolling more than 8 times, which is the maximum unrolling that we considered. Neither flushing the ORB nor passing the homefree token are bottlenecks for any application. However, from Figure 4.13(b) we observe that the memory system (*dcache*) is a performance bottleneck for ART, EQUAKE, and MGRID.

In the next two sections we investigate the sensitivity of TLS to inter-node communication latency for multinode architectures, and then explore ways to improve the memory system behavior of multi-node architectures through improved page management.

4.3.3 Sensitivity to Inter-Node Communication Latency

In Section 3.6.1, we investigated the sensitivity of TLS execution to the communication latency between processors within a single node or chip, and found that some general-purpose applications are quite sensitive. For multi-node architectures, it is important to understand the sensitivity of TLS execution to the communication latency between nodes. While *multi-chip module* technology (as used in the IBM Power4 [38]) can provide fast inter-chip communication by incorporating several processor chips in a single package, it is not always cost-effective to do so. Figure 4.14 shows the impact of varying the communication latency between nodes for TLS events (spawn, forwarding values, passing the homefree token) from 50 to 200 cycles. Our baseline architecture has an inter-node communication latency of 100 cycles. We observe that ART and MESA are somewhat sensitive to communication latency, while the other three benchmarks are not. In Figure 4.14(b), the performance of MGRID actually improves as the communication latency increases: in this case, the increased latency decreases parallel overlap and indirectly synchronizes some of the data



(a) Two nodes with four processors per node.

(b) Four nodes with four processors per node.

Figure 4.14. Impact of varying the communication latency between nodes (chips) from 50 to 200 cycles; note that for our baseline architecture it is 100 cycles.

dependences that cause speculation to fail for this benchmark. Overall, these results suggest that an inter-node communication latency of up to 200 cycles would still allow TLS to improve the performance of numeric applications on multi-node architectures.

4.3.4 Impact of Page Layout

Looking at Figure 4.13(b), we see that time spent servicing data cache misses (*dcache*) represents a significant portion of execution time for every benchmark, especially for those that do not scale well for certain configurations (ART, EQUAKE, and MGRID). Although other communication-related components are also significant (such as spawn and synchronization (*sync*) times), we cannot easily adjust the communication latency between nodes in a real system—however, in Chapter 5, we investigate ways to improve the efficiency of value communication between speculative threads. Hence, in this section we instead focus on the performance of the memory system when scaling up to multi-node architectures.

In Figure 4.15 we show the breakdown of all cycles spent servicing misses (in the memory system) for the floating

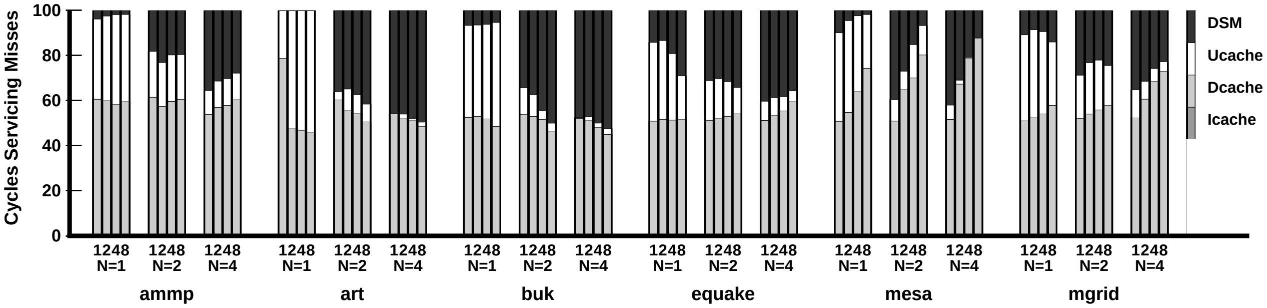
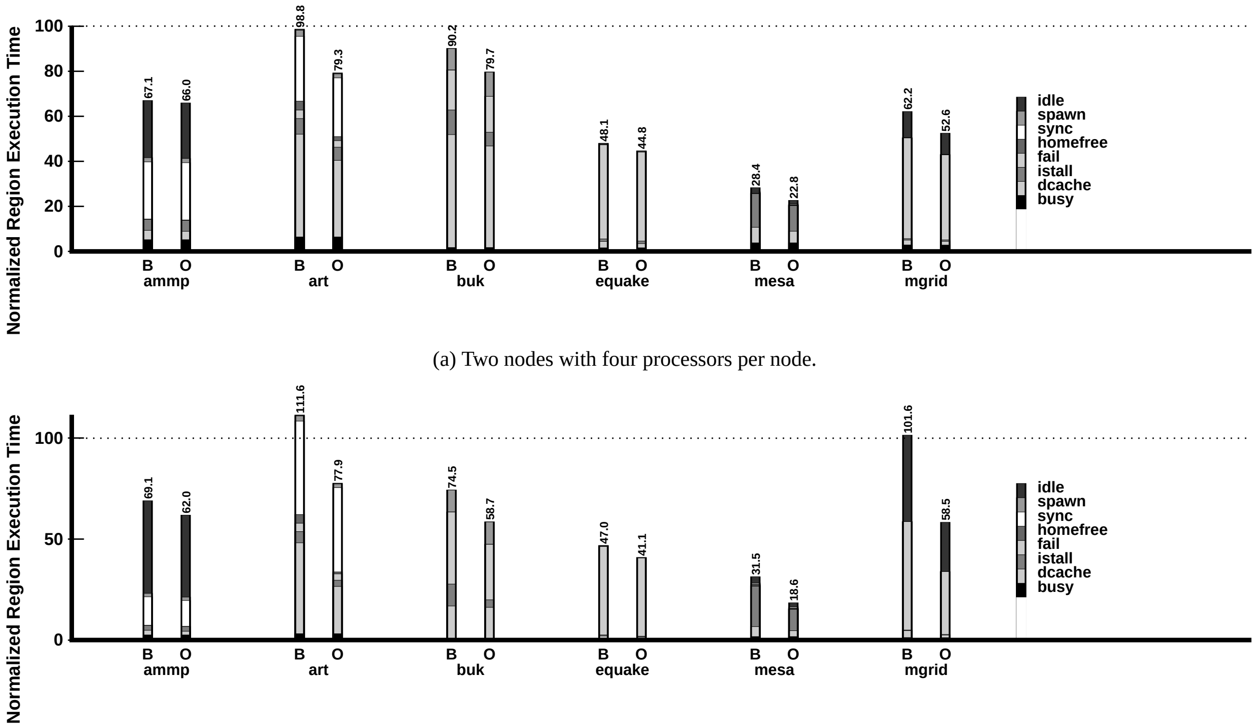


Figure 4.15. Breakdown of cycles spent servicing misses (in the memory system) for varying numbers of nodes (N) and processors per node. *DSM* **represents cycles spent on accessing both local and remote memory in the distributed shared memory system;** *Ucache* **represents cycles spent on contention and transmission in the inter-connection network (crossbar) between the data and instruction caches and the unified cache, as well as cycles spent on fill and contention in the unified cache itself;** *Dcache* **represents cycles spent for fill and contention in the data cache while servicing data cache misses; and** *Icache* **represents cycles spent on both filling cache lines and contention in the instruction cache while servicing instruction cache misses.**

point benchmarks. Note that this is different than the *dcache* segment measured in previous breakdown graphs. Each bar is broken into four segments: cycles spent on both filling cache lines and contention in the instruction cache while servicing instruction cache misses (*Icache*); cycles spent for fill and contention in the data cache while servicing data cache misses (*Dcache*); cycles spent on contention and transmission in the inter-connection network (crossbar) between the data and instruction caches and the unified cache, as well as cycles spent on fill and contention in the unified cache itself (*Ucache*); and cycles spent on accessing both local and remote memory in the distributed shared memory system (*DSM*). Since the total number of cycles spent servicing misses can vary widely as we increase the number of processors and nodes, each bar is normalized to 100% so that we can more easily observe the underlying trends.

We note that the instruction cache time (*Icache*) is negligible, while each application spends close to half of all memory system cycles in the data cache (*Dcache*). For each benchmark, the fraction of cycles spent in the data cache (*Dcache* for a given number of processors is nearly constant as we vary the number of nodes. In contrast, the fraction of cycles spent in the distributed shared memory system (*DSM*) increases significantly for every benchmark as the number of nodes increases, indicating that this is a limitation on scalability. One way to attack this problem is in the *layout* of pages in the DSM system.

In Figure 4.16, we investigate the impact of the layout of pages in memory. For brevity we analyze only two and four node architectures, each with four processors per node. The *B* bar represents the “baseline” data layout from Figure 4.13 where pages are assigned in a round-robin fashion amongst the DSM nodes. To evaluate the potential for



(a) Two nodes with four processors per node.

(b) Four nodes with four processors per node.

Figure 4.16. Impact of the DSM page layout. *B* is the "baseline" data layout from Figure 4.13, and *O* models an oracle migration strategy to estimate the maximum potential benefit of improved page allocation.

improvement, the *O* experiment models an oracle page migration strategy where each page migrates to the appropriate node just in time to be referenced, such that data is always found in local rather than remote memory. This experiment indicates that there is still significant room for improvement in page allocation for every application except for AMMP. These results indicate that TLS execution prefers a page allocation mechanism that supports frequent page migration [73]—such techniques can be implemented simply through extra support in the operating system as well as performance counters that monitor cache misses.

4.4 Chapter Summary

While previous approaches to TLS hardware support one of the two levels of scaling (either within a chip or in a system composed of multiple chips), our hardware support for TLS is unique because it scales seamlessly both within and beyond chip boundaries. This ability to scale both up and down allows our scheme to apply to a wide variety of multithreaded processors and larger systems that use those processors as building blocks.

Our evaluation of support for TLS in shared-cache architectures showed that performance is similar to that of private-cache architectures, since the increased cache locality of a shared-cache architecture is balanced with an increase in failed speculation due to conflicts. We supported our previous claim that support for implicit forwarding does not have a large impact on performance, although it is trivial to support in a shared-cache approach. We also showed that two techniques for tolerating read and write conflicts—suspending conflicting epochs and replicating cache lines—can significantly lower the amount of failed speculation in the benchmark applications.

Through an analysis of SPECfp2000 floating point benchmarks we demonstrated that some applications can speed up on multi-node architectures, by as much as 4.8 on a 4-node system with 8 processors per node. However, the resulting performance for other applications is limited by (i) the number of epochs per region instance (i.e. the amount of parallel work), (ii) synchronization and failed speculation, and (iii) the layout of data in the DSM system. We found that TLS can tolerate an inter-node communication latency of up to 200 cycles for multi-node architectures. We also showed that a simple page migration scheme can improve performance, and that further improvement is possible.

In the next chapter we investigate ways to further improve the efficiency of speculative execution by attacking one of the most significant performance bottlenecks: the communication of values between speculative threads.

Chapter 5

Improving Value Communication

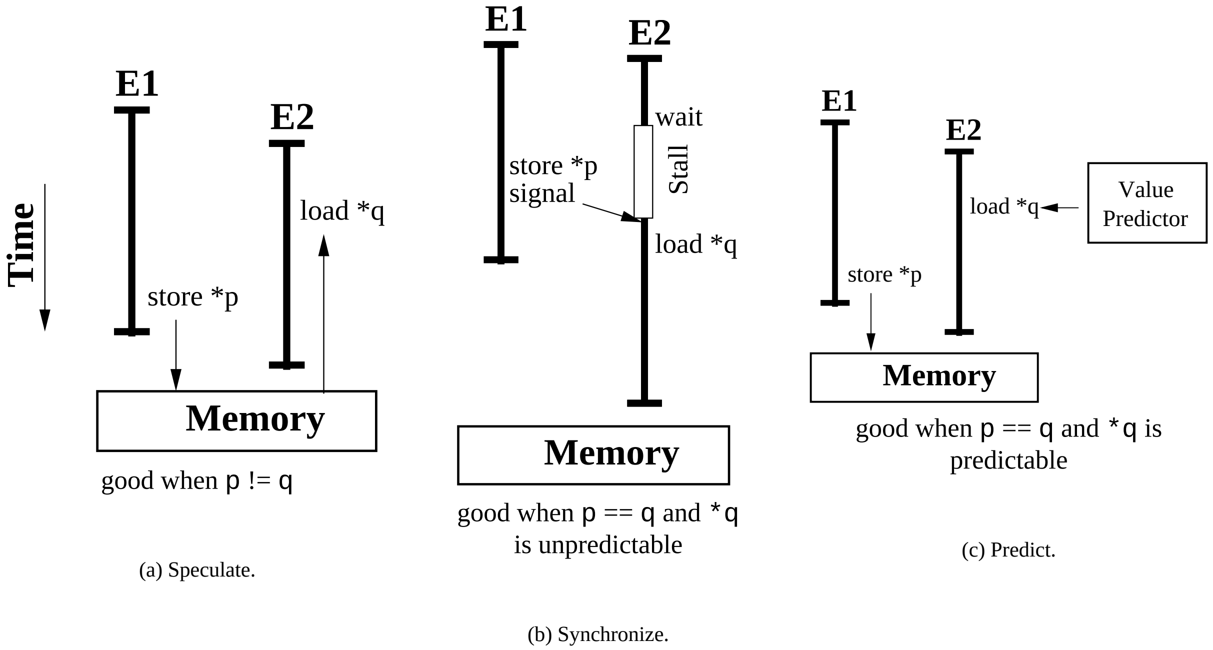
5.1 Introduction

For thread-level speculation, a key bottleneck to good performance lies in the three different ways to communicate a value between speculative threads: speculation, synchronization, and prediction. The difficult part is deciding how and when to apply each method. In this chapter we show how to apply value prediction, dynamic synchronization, and hardware instruction prioritization to improve value communication and hence performance for TLS.

5.1.1 The Importance of Value Communication for Thread-Level Speculation

In the context of TLS, value communication refers to the satisfaction of any true (read-after-write) dependence between *epochs* (sequential chunks of work performed speculatively in parallel). From the compiler's perspective, there are two ways to communicate the value of a given variable. First, the compiler may speculate that the variable is not modified (Figure 5.1(a)). However, if at run-time the variable actually *is* modified then the underlying hardware ensures that the misspeculated epoch is re-executed with the proper value. This method only works well when the variable is modified infrequently, since the cost of misspeculation is high. Second, if the variable is frequently modified, then the compiler may instead synchronize and forward the value[1] between epochs (Figure 5.1(b)). Since a parallelized region of code contains many variables, the compiler employs a combination of speculation and synchronization as appropriate.

[1] This is also known as *DOACROSS* [17, 57] parallelization.



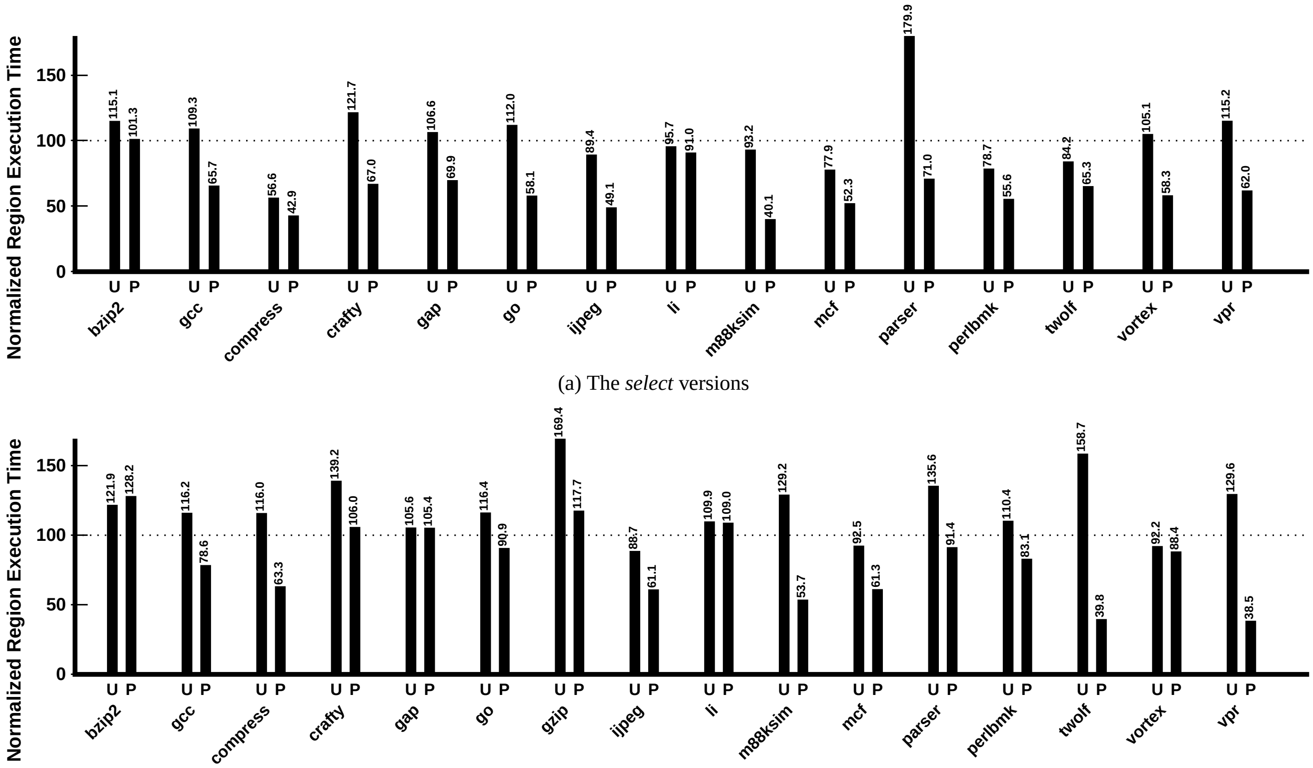
(a) Speculate.

(b) Synchronize.

(c) Predict.

Figure 5.1. A memory value may be communicated between two epochs (E1 and E2) through (a) speculation, (b) synchronization, or (c) prediction.

To further improve upon static compile-time choices between speculating or synchronizing for specific memory accesses, we can exploit dynamic run-time behavior to make value communication more efficient. For example, we might exploit a form of *value prediction* [4, 28, 49, 51, 62, 64, 74], as illustrated in Figure 5.1(c). To get a sense of the potential upside of enhancing value communication under TLS, let us briefly consider the ideal case. From a performance perspective, the ideal case would correspond to a value predictor that could perfectly predict the value of any inter-thread dependence. In such a case, speculation would never fail and synchronization would never stall. While this perfect-prediction scenario is unrealistic, it does allow us to quantify the potential impact of improving value communication in TLS. Figure 5.2 shows the impact of perfect prediction on the performance of both the *select* and *max-coverage* benchmarks (described in Chapter 2), evaluated on a 4-processor CMP that implements our TLS scheme as described in Chapter 3. Each bar shows the total execution time of all speculatively-parallelized regions of code, normalized to that of the corresponding original sequential versions of these same codes. As we see in Figure 5.2, efficient value communication often makes the difference between speeding up and slowing down relative to the original sequential code. Hence this is clearly an important area for applying compiler and hardware optimizations.



(a) The *select* versions

(b) The *max-coverage* versions

Figure 5.2. Potential impact of optimizing value communication. Relative to the normalized, original sequential version, U shows the unoptized speculative version and P shows perfect prediction of all inter-thread data dependences.

5.1.2 Techniques for Improving Value Communication

Given the importance of efficient value communication in TLS, what solutions can we implement to approach the ideal results of Figure 5.2? Figure 5.1 shows the spectrum of possibilities: i.e. speculate, synchronize, or predict. In our baseline scheme, the compiler synchronizes dependences that it expects to occur frequently (by explicitly “forwarding” their values between successive epochs) [76], and speculates on everything else. How can we use hardware to improve on this approach? Hardware support for efficient *speculation* is addressed in Chapters 3 and 4. Therefore our focus in this chapter is how to exploit and enhance the remaining spectrum of possibilities (i.e. *prediction* and *synchronization*) such that they are complementary to speculation within TLS. In particular, we explore the following techniques:

Value Prediction: We can exploit *value prediction* by having the consumer of a potential dependence use a predicted value instead, as illustrated in Figure 5.1(c). After the epoch completes, it will compare the predicted value with the actual value; if the values differ, then the normal speculation recovery mechanism will be invoked to squash and restart the epoch with the correct value. We explore using value prediction as a replacement for both

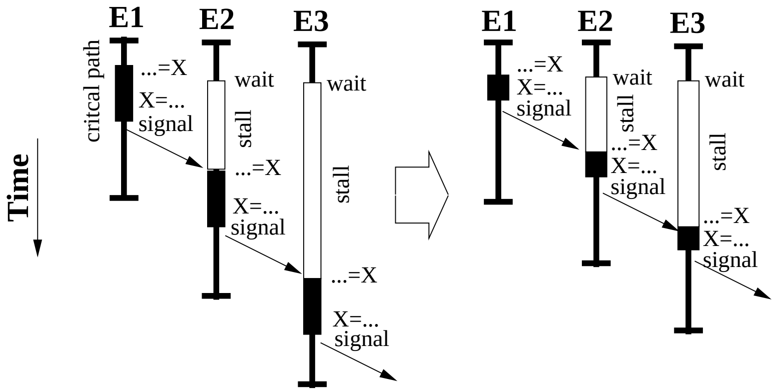


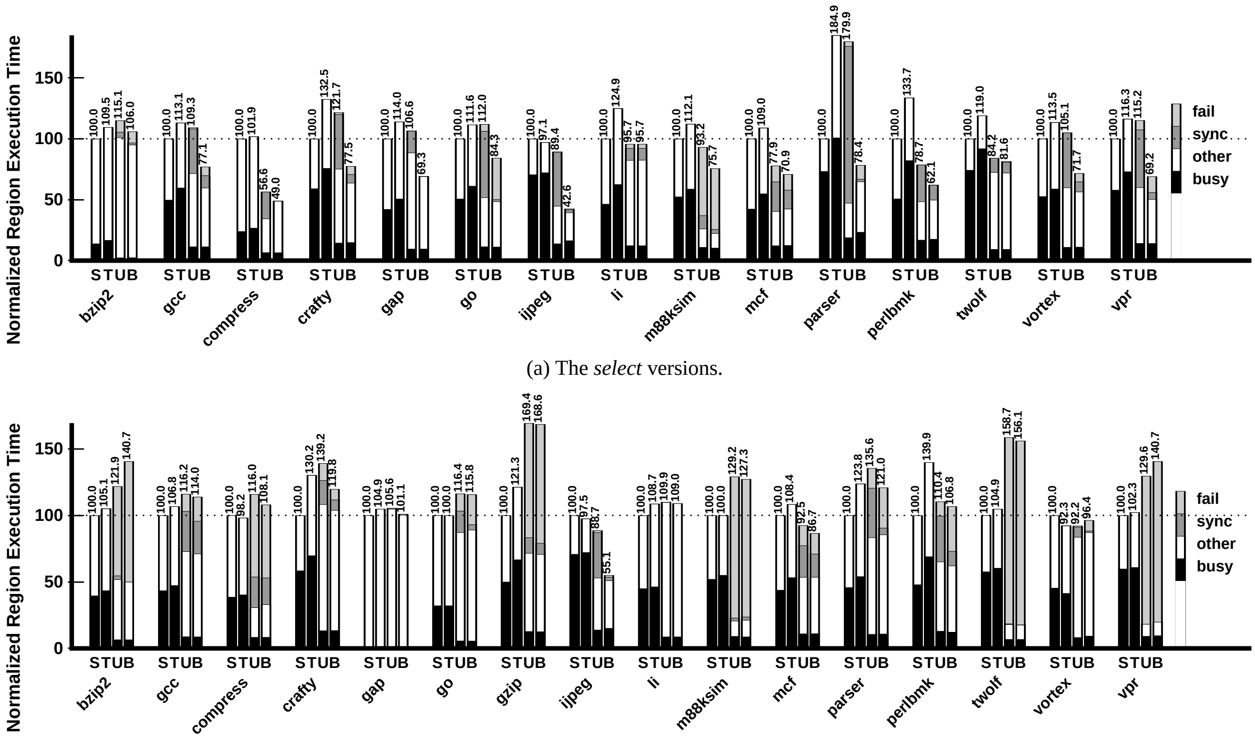
Figure 5.3. Reducing the critical forwarding path.

speculation and *synchronization*. In the former case (which we refer to later as "*memory value prediction*"), successful value prediction avoids the cost of recovery from an unsuccessful speculative load. In the latter case (which we refer to later as "*forwarded value prediction*"), successful prediction avoids the need to stall waiting for synchronization. Because the implementation issues and performance impact differ for these two cases, we evaluate them separately.

Silent Stores: An interesting program phenomenon is that many stores have no real side-effect since they overwrite memory with the same value that is already there. These stores are called *silent stores* [48], and we can exploit them when they occur to avoid failed speculation. Although one can view silent stores as a form of value prediction, the mechanism to exploit them is radically different from what is shown in Figure 5.1(c) since the changes occur at the *producer* of a communicated value, rather than the consumer.

Hardware-Inserted Dynamic Synchronization: In cases where the compiler decided to speculate that a potential true (read-after-write) dependence between speculative threads was not likely to occur, but where the dependence does in fact occur frequently and the communicated value is unpredictable, the best option would be to explicitly synchronize the threads (Figure 5.1(b)) to avoid the full cost of failed speculation. However, since the compiler did not recognize that such synchronization would be useful, another option is for the *hardware* to automatically switch from speculating to synchronizing when it dynamically detects such bad cases.

Reducing the Critical Forwarding Path: Once synchronization is introduced to explicitly forward values across epochs, it creates a dependence chain across the threads that may ultimately limit the parallel speedup. We



(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 5.4. Performance impact of our TLS compiler. For each experiment, we show normalized region execution time scaled to the number of processors multiplied by the number of cycles (smaller is better). S is the sequential version, T is the TLS version run sequentially. There are two versions of TLS code run in parallel: U and B are without and with compiler scheduling of the critical forwarding path, respectively. Each bar shows a breakdown of how time is being spent.

can potentially improve performance in such cases by using scheduling techniques to reduce the critical path between the first use and last definition of the dependent value, as illustrated in Figure 5.3. We implement both compiler and hardware methods for reducing the critical forwarding path.

5.2 A Closer Look at Improving Value Communication

In this section, we evaluate the impact of compiler optimization on performance and then show the potential for further improvement by hardware techniques.

5.2.1 Impact of Compiler Optimization

We begin by analyzing the performance impact of our compiler on TLS execution. Figure 5.4 shows the performance on a 4-processor chip-multiprocessor. For each application in Figure 5.4, the leftmost bar (*S*) is the original sequential version of the code, and the next bar (*T*) is the TLS version of the code run on a single processor. For each experiment, we show region execution time normalized to the sequential case (*S*); hence bars that are less than 100 are speeding up, and bars that are larger than 100 are slowing down, relative to the sequential version. Comparing the TLS version run sequentially (*T*) with the original sequential version (*S*) isolates the overhead of TLS transformation; across all *select* benchmarks, this overhead averages 20.0%. When we run the TLS code in parallel, it must overcome this overhead in order to achieve an overall speedup.

Each bar in Figure 5.4 is broken down into four segments explaining what happened during all potential graduation slots.[2] The *fail* segment represents all slots wasted on failed thread-level speculation, and the remaining three segments represent slots spent on successful speculation. The *busy* segment is the number of slots where instructions graduate; the *sync* portion represents slots spent waiting for synchronization for a forwarded location; the *other* segment is all other slots where instructions cannot graduate (including the slots where the reorder buffer is empty, data cache misses, and pipeline hazards). While this breakdown is less detailed than that used in previous chapters, it allows us to focus on the two sources of overhead that we attack in this chapter: synchronization (*sync*) and failed speculation (*fail*).

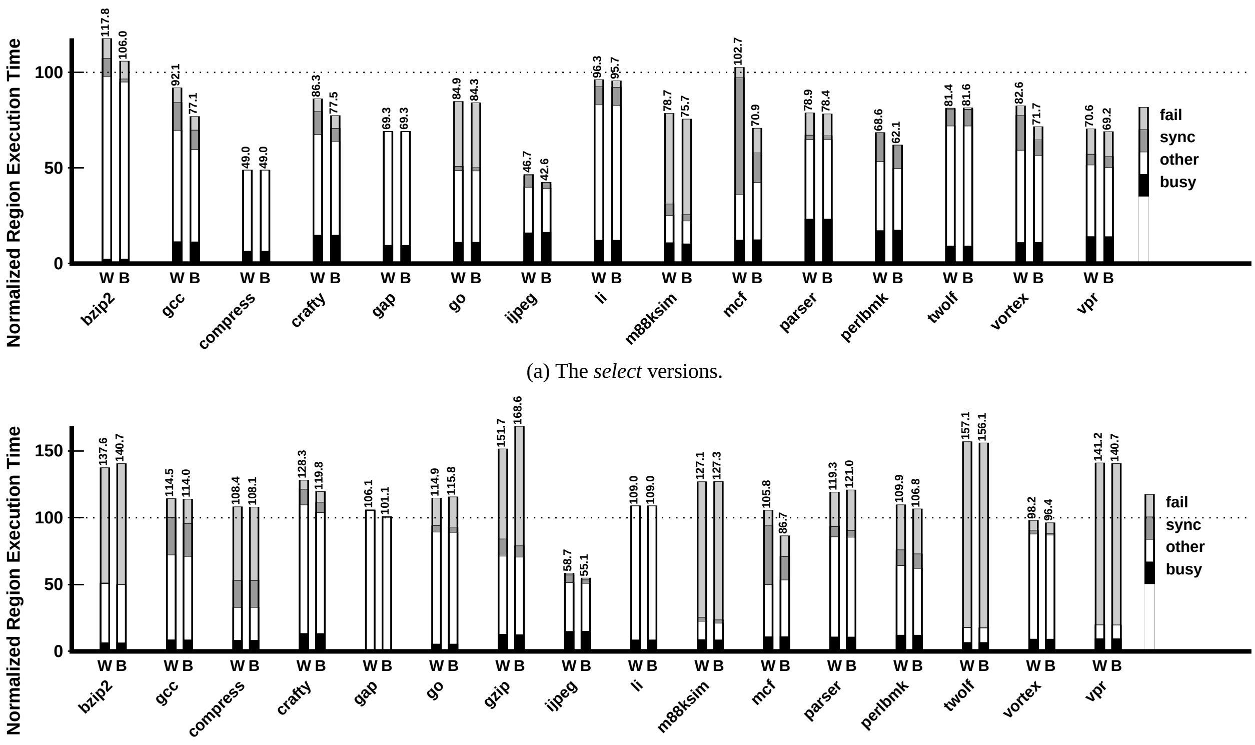
We consider two versions of TLS code running in parallel: the *B* case includes all of the compiler optimizations described earlier in Section 2.4, and the *U* case is the same minus the aggressive compiler scheduling to reduce the critical forwarding path. In nearly every case, the "unoptimized"[3] version (*U*) slows down with respect to the sequential version. The additional impediments include decreased data cache locality, synchronization, and failed speculation. Many benchmarks spend a significant amount of time synchronizing on forwarded values (as shown by the *sync* portion). If the compiler optimizes forwarded values by removing dependences due to certain loop induction variables and scheduling the critical path (*B*, our baseline), we observe that the performance of every *select* benchmark except for BZIP2, LI, and TWOLF improves substantially through decreased synchronization (*sync*), indicating that this is a crucial optimization.

5.2.2 Importance of Issuing Around `Wait` Instructions

For standard consistency models (such as release consistency [18]) a synchronization operation normally stalls further memory operations (i.e. through a memory barrier) until the synchronization operation is complete. However, for TLS support (which implements uniprocessor consistency with respect to ordered speculative threads) synchronization operations are quite different. TLS hardware associates each `wait` (i.e. `wait_for_value()`) instruction

[2] The number of graduation slots is the product of: (i) the issue width (4 in this case), (ii) the number of cycles, and (iii) the number of processors.

[3] Note that the "unoptimized" case still includes the `gcc` "`-O3`" flag, and is optimized in every way except for the aggressive critical forwarding path scheduling.



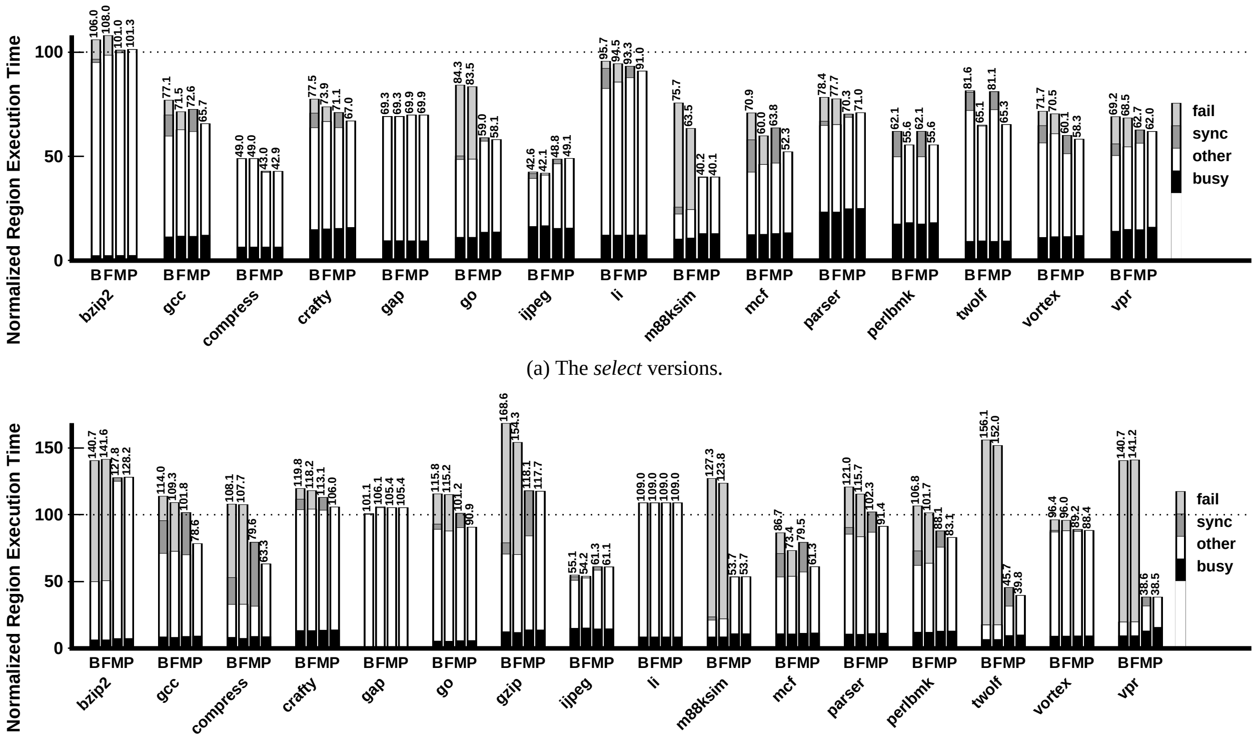
(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 5.5. Impact of issuing around `wait` instructions. For W, instructions cannot issue out-of-order with respect to a blocked `wait` instruction, while in B (our baseline) they can.

with the corresponding load instruction (which actually loads the value from the forwarding frame) by matching the forwarding frame offset (see Section 2.3.2). This association allows the load instruction to be blocked while other instructions are issued around it. Since the forwarded value is communicated directly between speculative threads through a mechanism that is logically separated from regular memory (i.e. the forwarding frame), consistency issues related to this form of synchronization are avoided.

In Figure 5.5 we evaluate the impact of the ability to issue instructions out-of-order with respect to a `wait` instruction (and the corresponding computation chain dependent on that forwarded value). For the *select* benchmarks, issuing around `wait` instructions provides a significant improvement for six applications (BZIP2, GCC, CRAFTY, MCF, PERLBMK, and VORTEX). For the *max-coverage* benchmarks, CRAFTY and MCF improve while four applications perform slightly worse: for the *max-coverage* benchmarks, increasing parallel overlap through decreased synchronization wait time indirectly increases the amount of failed speculation. The *select* benchmarks improve by an average of 6.3% while the *max-coverage* benchmarks only improve by an average of 1.3%. Overall, this support is both beneficial and straightforward to implement.



(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 5.6. Potential for improved value communication. B **is our baseline,** M **shows perfect prediction of memory values,** F **shows perfect prediction of forwarded values, and** P **shows perfect prediction of both forwarded and memory values.**

5.2.3 The Potential for Further Improvement by Hardware

To illustrate that the performance of many of our benchmarks are limited by the efficiency of value communication, we show in Figure 5.6 the impact of ideal prediction on performance, starting with the baseline (*B*) from Figure 5.4 which includes compiler scheduling of the critical forwarding path. In the *F* experiment we see the impact of perfect prediction of forwarded values (those explicitly communicated between epochs on the forwarding frame). In effect, this means that there will be no time spent waiting for synchronization of forwarded values. GCC, CRAFTY, M88KSIM, MCF, PERLBMK, and TWOLF show a substantial improvement, since the baseline experiment (*B*) shows these benchmarks to be somewhat limited by synchronization; the remaining *select* benchmarks do not improve significantly. The *max-coverage* benchmarks are relatively unaffected since they tend to be dominated by failed speculation.

In the *M* experiment, we measure the impact of perfect memory value prediction, such that no epoch will suffer from failed speculation. In this case, we see a great improvement in most benchmarks. The *select* versions of GCC, CRAFTY, MCF, PERLBMK, TWOLF, and VORTEX show a significant synchronization portion for the *M* experiment, indicating that synchronization is still a limiting factor for these applications; the *max-coverage* versions of GCC, COMPRESS, MCF, PERLBMK, and TWOLF show similar results.

Finally, in the *P* experiment we evaluate the impact of perfect prediction of both memory and forwarded value values—this is the same experiment shown in Figure 5.2. In this case, the *select* versions of GCC, CRAFTY, MCF, as well as the *max-coverage* versions of GCC, COMPRESS, MCF, PARSER, TWOLF show a significant increased benefit compared to either technique alone, while the other benchmarks show only modest improvements. Evidently, given our compiler support for improving synchronization [76], avoiding failed speculation is the main bottleneck to good performance; however, further improvement of synchronization may still be possible for some benchmarks. Also, if we cannot fully eliminate failed speculation then improving synchronization will still be important.

5.3 Techniques for When Prediction Is Best

For a frequently-occurring cross-epoch dependence that the compiler has decided to synchronize or speculate upon, an attractive alternative is to instead predict the value, eliminating synchronization stall time or failed speculation. In this section we investigate techniques for predicting values for TLS. We begin with a comparison of related work, followed by a description of the issues for predicting values for TLS (as compared with uniprocessor value prediction). We then investigate prediction of memory values and forwarded values, and the optimization of silent stores.

5.3.1 Related Work

Value prediction in the context of a uniprocessor is fairly well understood [28, 49, 64, 74], while value prediction for thread-speculative architectures is relatively new. Gonzalez *et al.* [51] evaluated the potential for value prediction when speculating at a thread-level on the innermost loops from SPECint95 [16] benchmarks, and concluded that predicting synchronized (forwarded) register dependences provided the greatest benefit, and that predicting memory values did not offer much additional benefit. The opposite is true of the results presented in this chapter for two reasons. First, our compiler has correctly scheduled easily-predictable but frequently-synchronized loop-induction variables so that they cannot cause dependence violations, and has also scheduled the code paths of forwarded values to minimize the impact of synchronization on performance. Second, we have selected much larger regions of code to speculate on, resulting in a greater number of memory dependences between threads. Cintra *et al.* [13] investigated the impact of value prediction after the compiler has optimized loop induction variables in floating-point applications. Several other works evaluate the impact of value prediction without such compiler optimization. Oplinger *et al.* [56] evaluate the potential benefits to TLS of memory, register, and procedure return value prediction, and Akkary *et al.* [4] and Rotenberg *et al.* [62] also describe designs that include value prediction.

5.3.2 Design Issues and Experimental Framework

Predicting values for TLS has similar issues to predicting values in the midst of branch speculation, but at a larger scale. With branch speculation, we do not want to update the value predictor for any loads on the mispredicted path. Also, when a value is mispredicted we need only squash a relatively small number of instructions, so the cost of misprediction is not large. Similarly, in TLS we only want to update the predictor for values predicted in successful epochs, but this will require either a larger amount of buffering or the ability to back-up and restore the state of the value predictor. Furthermore, the cost of a misprediction is high for TLS: the entire epoch must be re-executed if a value is mispredicted because a prediction cannot be verified until the end of the epoch when all modifications by previous epochs have been made visible.[4] Finally, for TLS we require that each epoch has a logically-separate value predictor. For SMT or other shared-pipeline speculation scheme, this does not mean that each requires a physically separate value predictor, but that the prediction entries must be kept separate by incorporating the epoch context identifier into the indexing function. This is necessary since multiple epochs may need to simultaneously predict different versions of the same location.

We model an aggressive hybrid predictor that combines a 1K×3-entry context predictor with a 1K-entry stride predictor, using 2-bit, up/down, saturating confidence counters to select between the two predictors. We found that the number of mispredictions can be minimized by simply predicting only when the prediction confidence is at the maximum value. Finding the smallest and simplest predictor that produces good results is beyond the scope of this dissertation. It is important to note that we also model misprediction by re-executing any epoch that has used a mispredicted-value. Finally, a misprediction is not detected until the end of the epoch when the prediction is verified.

5.3.3 Memory Value Prediction

One potential way to eliminate data dependence violations between speculative threads is through the prediction of memory values. But which loads should we predict? A simple approach would be to predict every load for which the predictor is confident. Previous work [9, 23] shows that focusing prediction on critical path instructions is important for

[4] Some schemes support selective-squashing of instructions that have used a mispredicted value [4], but this requires a large amount of buffering.

Table 5.1. Memory value prediction statistics for the *select* benchmarks.

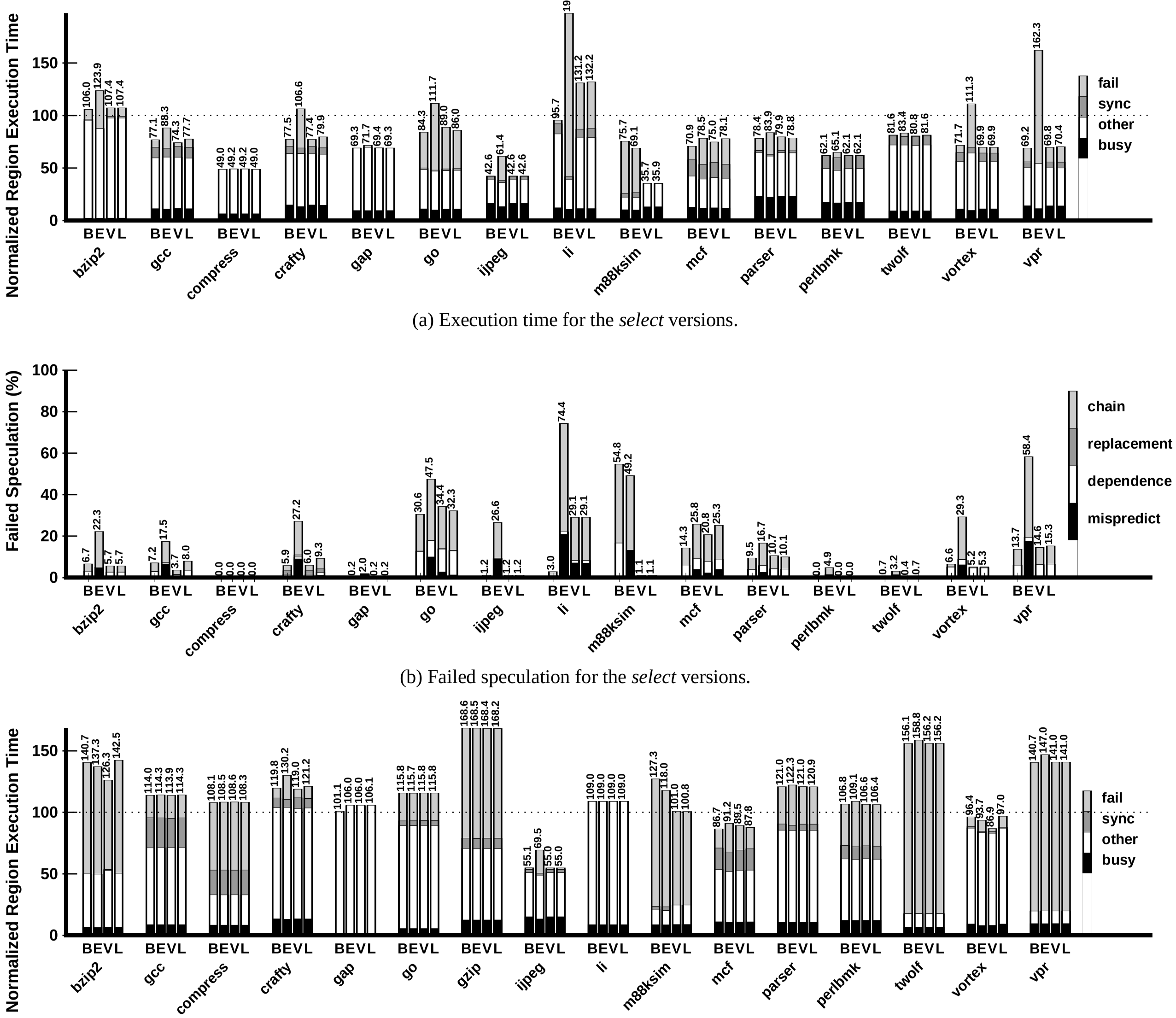
Application	Avg. Exposed Loads per Epoch	Incorrectly Predicted	Correctly Predicted	Not Confident	Avg. Unique Violating Loads Per Processor (Across all Regions)
BZIP2	31.9	4.0%	63.9%	32.0%	9.2
GCC	16.1	2.4%	68.3%	29.1%	139.8
COMPRESS	0.0	0.0%	60.0%	40.0%	0.0
CRAFTY	16.5	3.8%	53.9%	42.2%	7.8
GAP	15.5	0.3%	72.8%	26.7%	4.2
GO	33.4	2.6%	53.7%	43.6%	180.8
IJPEG	39.8	1.3%	41.4%	57.2%	12.5
LI	11.9	10.9%	48.7%	40.2%	7.8
M88KSIM	14.3	2.2%	69.6%	28.0%	1.0
MCF	12.1	1.1%	56.2%	42.6%	16.2
PARSER	38.9	0.2%	94.7%	05.0%	71.0
PERLBMK	6.0	0.2%	97.0%	02.6%	0.0
TWOLF	4.3	4.7%	67.8%	27.3%	3.0
VORTEX	44.9	2.5%	60.4%	36.9%	4.8
VPR	34.8	6.7%	53.7%	39.4%	10.5

uniprocessor value prediction when modeling realistic misprediction penalties. Similarly, the cost of misprediction in TLS is very high, so we instead want to focus only on the loads that can potentially cause misspeculation. Fortunately, this information is available from the speculative cache line state: only loads that are *exposed*[5] can cause speculation to fail. Since our scheme tracks which words have been speculatively-modified in each cache line, we can decide whether a given load is exposed. Furthermore, there is no point predicting a load if the speculatively-loaded (SL) bit is already set for the corresponding cache line, since any failed speculation will not be averted; for this study, we will not consider such loads to be exposed.

Table 5.1 shows some statistics for the *select* benchmarks for exposed loads and their predictability, using the predictor described above. The exposed loads in eight applications are quite predictable (correctly predicted more than 60% of the time), while the remaining benchmarks also provide a significant fraction of correct predictions. We see that the amount of misprediction is small—an average of 2.9% across all *select* benchmarks, with LI having the worst misprediction rate at 10.9%. The rates of correct prediction are quite high, averaging 64.1% across all benchmarks. Hence we expect memory value prediction to work well.

In Figure 5.7, the *E* experiment shows the impact of predicting all exposed loads for which the predictor is confident. In almost every case for the *select* benchmarks, performance is worse due to an increased amount of failed speculation caused by misprediction; these benchmarks perform 23.5% worse on average. The problem is that despite several

[5] A load that is not proceeded in the same epoch by a store to the same location is considered to be exposed [3].

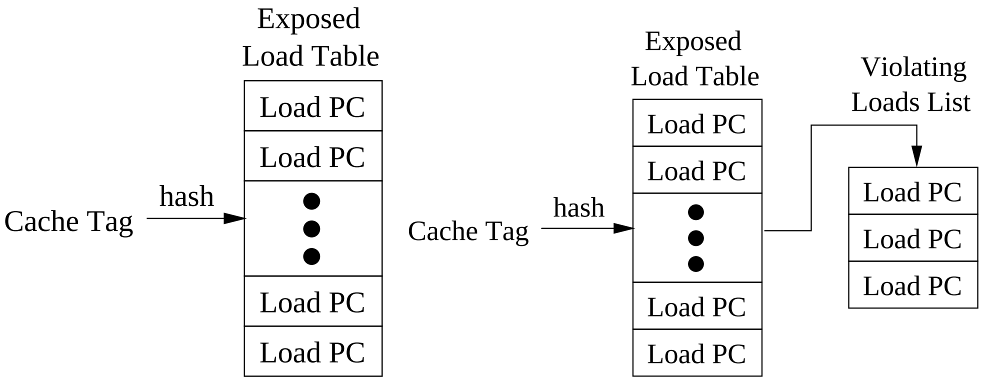


(a) Execution time for the *select* versions.

(b) Failed speculation for the *select* versions.

(c) Execution time for the *max-coverage* versions.

Figure 5.7. Performance and failed speculation with memory value prediction. B **is the baseline experiment,** E **predicts all** *exposed* **loads,** V **only predicts loads that have caused violations using an exposed load table and a violating loads list that are both unlimited in size, and** L **refines** V **with tables/lists that are realistic in size.**



(a) Exposed load table, updated on every exposed load.

(b) Violating loads list, updated on every dependence violation.

Figure 5.8. Two mechanisms used to throttle memory value prediction: the exposed load table and violating loads list.

correct predictions for a given epoch, it only takes a single misprediction to cause speculation to fail for that epoch and all active logically-later epochs as well. Figure 5.7(b) shows a breakdown of the percentage of time wasted on failed speculation: *chain* represents time spent on epochs that were squashed because logically-earlier epochs were previously squashed; (*replacement* represent violations caused by replacement in either the first-level data caches or the shared unified cache; *dependence* represents speculation that fails due to violation of a true (read-after-write) data dependence; and *mispredict* represents failed speculation due to mispredicted memory values. It is evident that misprediction of memory values (*mispredict*) is the cause of performance loss. The *max-coverage* benchmarks also suffer an increase in failed speculation for this experiment (*E*), although it is less severe (only 3.7% worse on average). For both sets of benchmarks, only the performance of M88KSIM improves.

Rather than predict all exposed loads, we can be even more selective by only predicting loads that are likely to cause dependence violations. We can track these loads with the following two devices. First, for each speculative context, we keep a hash table (called the *exposed load table*) that is indexed by the cache line tag and stores the PC of the most recent corresponding exposed load, as shown in Figure 5.8(a). This table is reset when an epoch commits. Second, whenever a dependence violation occurs it is associated with a cache line, so we can use the cache line tag to index the exposed load table and retrieve the PC of the offending load, as shown in Figure 5.8(b). Hence we can keep a list of load PCs (also for each speculative context) which have caused violations (the *violating loads list*), and can now use this list to decide which loads we should predict. In Table 5.1 we see that the average number of unique violating loads (by PC) per processor for each benchmark is not large, averaging 31.2 across all benchmarks.

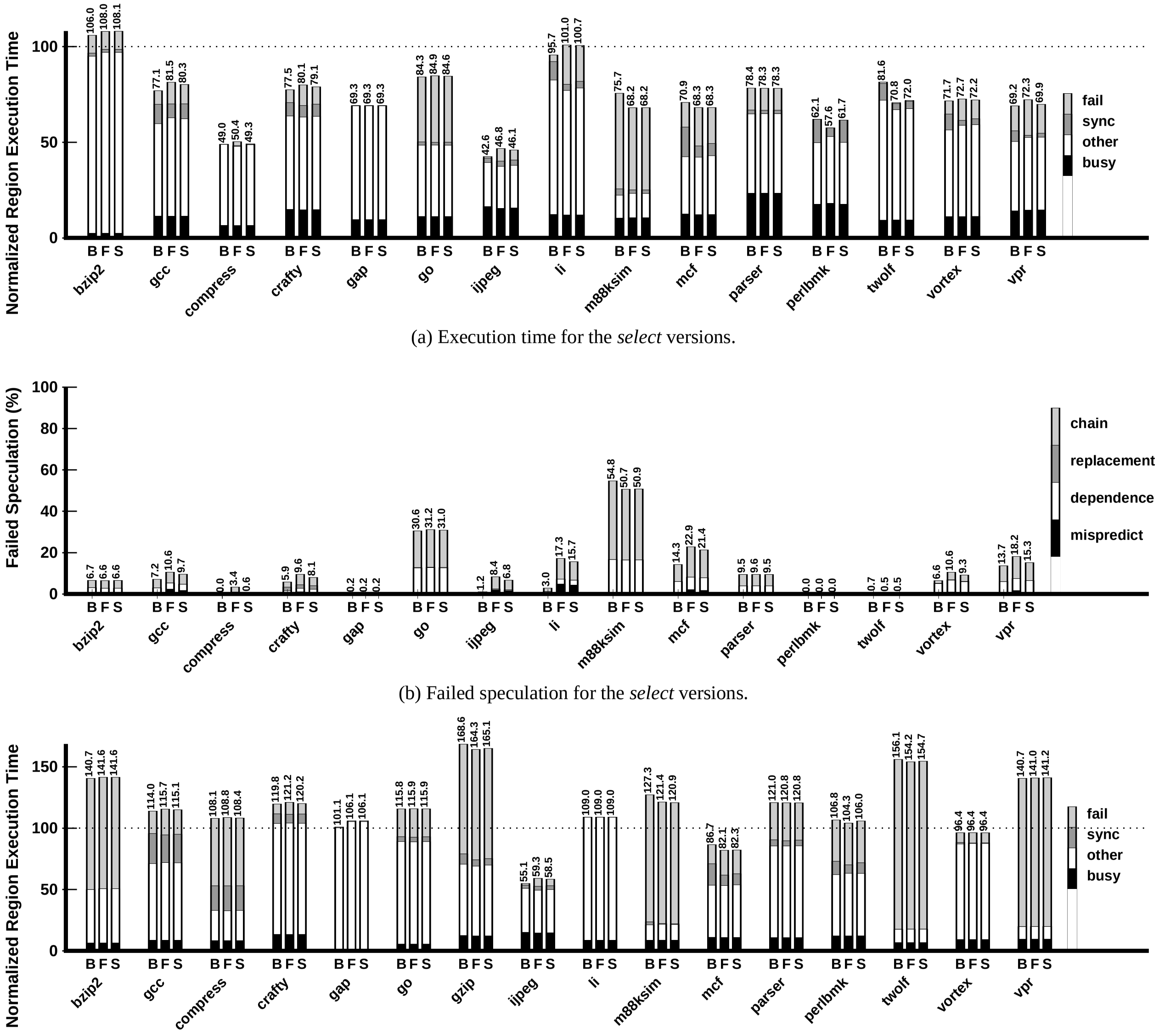
In Figure 5.7, the *V* experiment shows the impact of predicting only loads that have caused violations, as given by the violating loads list. For this experiment, both the exposed load tables and violating loads lists are unlimited in size. These mechanisms throttle prediction for the *select* benchmarks, although average performance is still overall 0.2% worse than that of the baseline. The performance of most benchmarks is relatively unaffected, except for M88KSIM which performs 52.8% better, and LI which performs 37.0% worse than the baseline. The *max-coverage* benchmarks perform 2.7% better on average than the corresponding baseline, with only BZIP2, M88KSIM, and VORTEX performing significantly better.

In the *L* experiment in Figure 5.7, we model more realistic storage sizes for our throttling mechanisms, such that both the exposed-load tables and violating loads lists have only 32 entries each. Since the exposed-loads table is implemented with a hashing index function, any conflicting cache tag index simply over-writes that entry, such that the table saves only the most recent exposed loads. Similarly, the violating loads list only saves the most recent violating loads. We observe that the performance of the *V* experiment is maintained except for the *max-coverage* versions of BZIP2 and VORTEX. Hence with proper throttling, memory value prediction can be effective for some applications; for the *select* benchmarks, loads that cause violations are not easily predictable, suggesting that synchronization may be a better option for speculative regions with little speculation. Memory value prediction is more effective for the *max-coverage* benchmarks (which have a greater amount of failed speculation), suggesting that value prediction can help increase the number of regions of code for which speculative parallelization is effective.

Having explored value prediction for the sake of avoiding failed speculation, we now turn our attention to using value prediction to mitigate the performance impact of explicit synchronization.

5.3.4 Prediction of Forwarded Values

Recall that forwarded values are those that are frequently modified and hence their accesses are synchronized and forwarded between epochs (using the forwarding frame) by the compiler. Similar to memory values, we can also predict forwarded values. However, while we predict memory values to decrease the amount of failed speculation, we predict forwarded values to decrease the amount of time spent synchronizing. Table 5.2 shows some statistics for the prediction of forwarded values for the *select* benchmarks, using the predictor described previously. We observe that the fraction of incorrect predictions for forwarded values (averaging 3.8% across all benchmarks) is somewhat



(a) Execution time for the *select* versions.

(b) Failed speculation for the *select* versions.

(c) Execution time for the *max-coverage* versions.

Figure 5.9. Performance of forwarded value prediction. B is the baseline experiment, F predicts all forwarded values S predicts forwarded values that have caused stalls.

Table 5.2. Forwarded value prediction statistics for the *select* benchmarks.

Application	Incorrectly Predicted	Correctly Predicted	Not Confident	Average Number of "Waiting" Loads
BZIP2	0.4%	22.8%	76.7%	0.9
GCC	6.7%	34.5%	58.6%	2.0
COMPRESS	2.4%	90.3%	7.2%	0.2
CRAFTY	3.6%	6.3%	90.0%	2.1
GAP	0.3%	0.0%	99.6%	0.0
GO	2.0%	12.4%	85.5%	0.9
IJPEG	7.7%	32.4%	59.8%	2.8
LI	13.1%	39.2%	47.6%	1.0
M88KSIM	8.0%	64.1%	27.8%	0.7
MCF	9.3%	52.5%	38.0%	1.4
PARSER	0.5%	19.3%	80.0%	0.2
PERLBMK	0.1%	98.8%	1.1%	0.1
TWOLF	0.1%	14.8%	85.1%	1.5
VORTEX	0.7%	71.1%	28.1%	2.3
VPR	2.6%	35.6%	61.7%	2.6

higher than for memory values (as shown in Table 5.1) while the fraction of correct predictions for forwarded values (averaging 39.6%) is not as high.

In Figure 5.9, the *F* experiment shows the impact of predicting forwarded values: for the *select* benchmarks the amount of time spent waiting for synchronization is significantly reduced for M88KSIM, PERLBMK, and TWOLF, although the amount of failed speculation for GCC, CRAFTY, IJPEG LI, VORTEX, and VPR increases. From Figure 5.9(b) we see that this increase is indeed due to the misprediction of forwarded values (*mispredict*) for GCC, LI, IJPEG, MCF, and VORTEX. The remaining benchmarks (LI, VORTEX, and VPR) suffer an increase in failed speculation due to violated data dependences (*dependence*): dependences which previously were indirectly synchronized by value forwarding are now exposed due to the increased overlap resulting from successfully predicted forwarded values. This observation suggests that prediction of forwarded values will perform better when used in concert with other techniques for reducing failed speculation. Overall, forwarded value prediction yields an average improvement of 0.1% for the *select* benchmarks and 0.3% for the *max-coverage* benchmarks.

To throttle forwarded value prediction, we apply a similar technique to that used for memory value prediction: we track which forwarded loads that cause the pipeline to stall waiting for synchronization (*waiting loads*), and only predict those values. Similar to violating loads, these loads can be tracked through the use of a small list of PCs: in Table 5.2 we see that the number of unique waiting loads per benchmark is quite small, averaging 1.2 loads across all benchmarks. The *S* bars shows the results of this experiment, which for the *select* benchmarks maintains the benefits of the *F* experiment for M88KSIM and TWOLF and also helps mitigate the negative impact on IJPEG, VORTEX, and

Table 5.3. Percent of dynamic, non-stack stores that are silent (for the *select* benchmarks).

Application	Dynamic, Non-Stack Silent Stores
BZIP2	4.3%
CC1	8.7%
COMPRESS95	94.8%
CRAFTY	3.3%
GAP	0.2%
GO	1.4%
IJPEG	30.7%
LI	0.0%
M88KSIM	19.4%
MCF	10.7%
PARSER	1.4%
PERLBMK	1.3%
TWOLF	0.6%
VORTEX	53.7%
VPR	5.9%

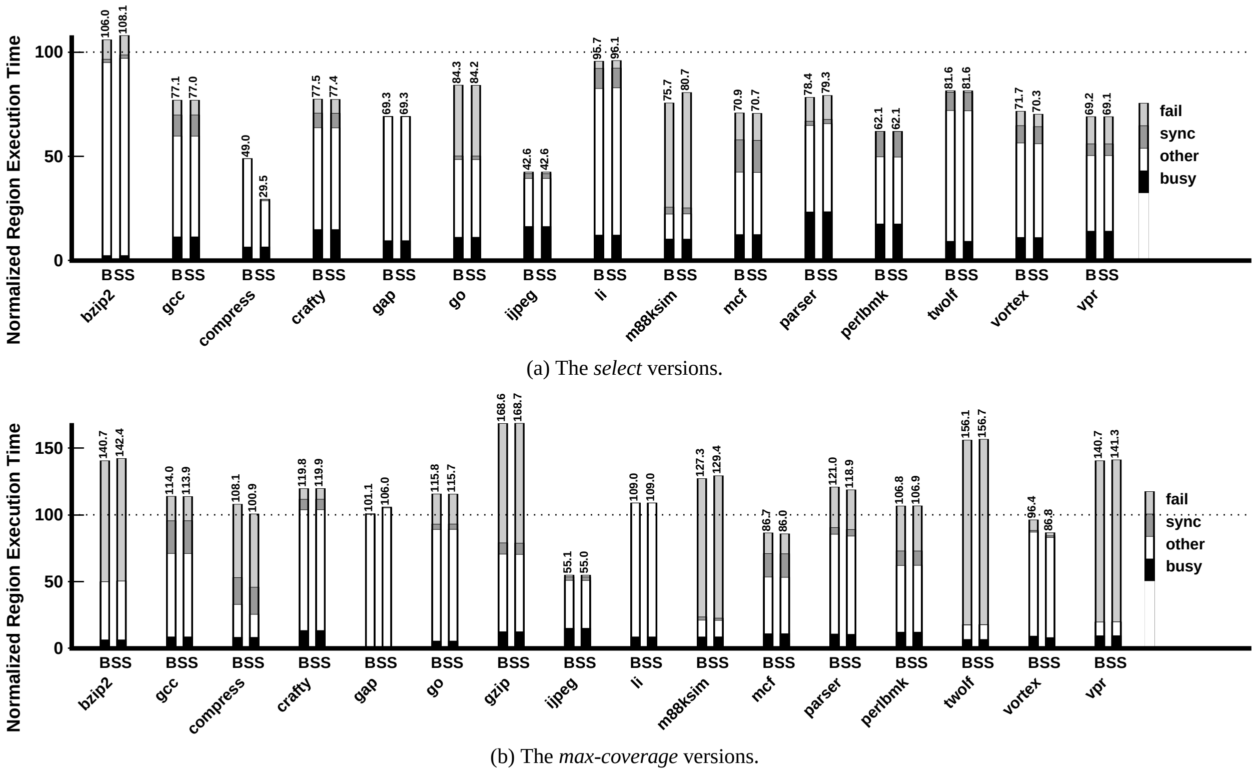
VPR. Overall, throttled forwarded value prediction yields an average improvement of 0.2% for the *select* benchmarks and the same 0.3% improvement for the *max-coverage* benchmarks. Similar to memory value prediction, for some applications the prediction of forwarded values is an effective way to reduce the amount of time spent synchronizing, but does not have a large impact on most applications.

5.3.5 Silent Stores

Often, a store does not actually modify the value of its target location. In other words, the value of the location before the store is the same as the value of the location after the store. This occurrence is known as a *silent store* [48], and was first exploited to reduce coherence traffic. A store that is expected to be silent is replaced with a load, and the loaded value is compared with the value to be stored. If they are not the same, then the store is executed after all, otherwise we save the coherence traffic of gaining exclusive access to the cache line and eliminate future update traffic. This same technique can be applied to TLS to avoid data dependence violations so that a dependent store-load pair can be made independent if the store is actually silent.

In Table 5.3 we see that silent stores are abundant in the *select* benchmarks, ranging from 4% to 80% of all dynamic non-stack stores within speculative regions, and averaging 15.8% across all benchmarks. However, what matters is whether the stores which cause dependence violations are silent.

The mechanism for exploiting silent stores for TLS is somewhat different than that used for exploiting silent stores in a uniprocessor. Replacing a potentially-silent store with a load could cause the underlying speculative coherence scheme to detect a read-after-write dependence violation if a logically-earlier epoch stores to the same cache line. Instead, we convert a potentially-silent store to a prefetch (which is ignored by the data dependence tracking mechanism),



(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 5.10. Performance of silent stores optimization. B **is the baseline experiment, and** S **optimizes silent stores.**

and verify the location later via a load when the epoch becomes homefree.

Figure 5.10 shows that optimizing silent stores results in a substantial improvement for COMPRESS (40.0%), a slight improvement for most benchmarks, and an increase in failed speculation for BZIP2 and M88KSIM. Exploiting silent stores yields an improvement of 0.9% for the *select* benchmarks and 0.6% for the *max-coverage* benchmarks, Compared with using value prediction to avoid potential memory dependences (as we explored earlier in Section 5.3.3), the silent stores approach requires significantly less hardware support (e.g., no value predictor is needed). Hence this appears to be an attractive technique for enhancing TLS performance.

5.4 Techniques for When Synchronization Is Best

For frequently-occurring cross-epoch dependences with unpredictable values, the only remaining alternative is to synchronize. In this section we investigate techniques for dynamic synchronization of dependences, and also prioritization of the critical forwarding path between synchronized accesses.

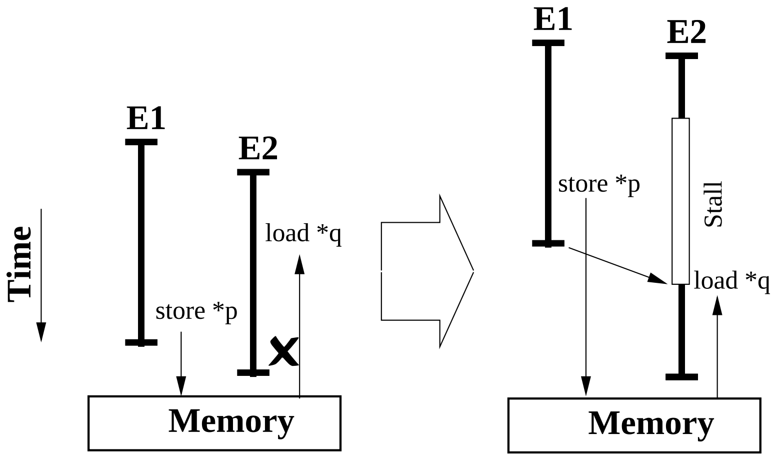


Figure 5.11. Dynamic synchronization, which avoids failed speculation (left) by stalling the appropriate load until the previous epoch completes (right).

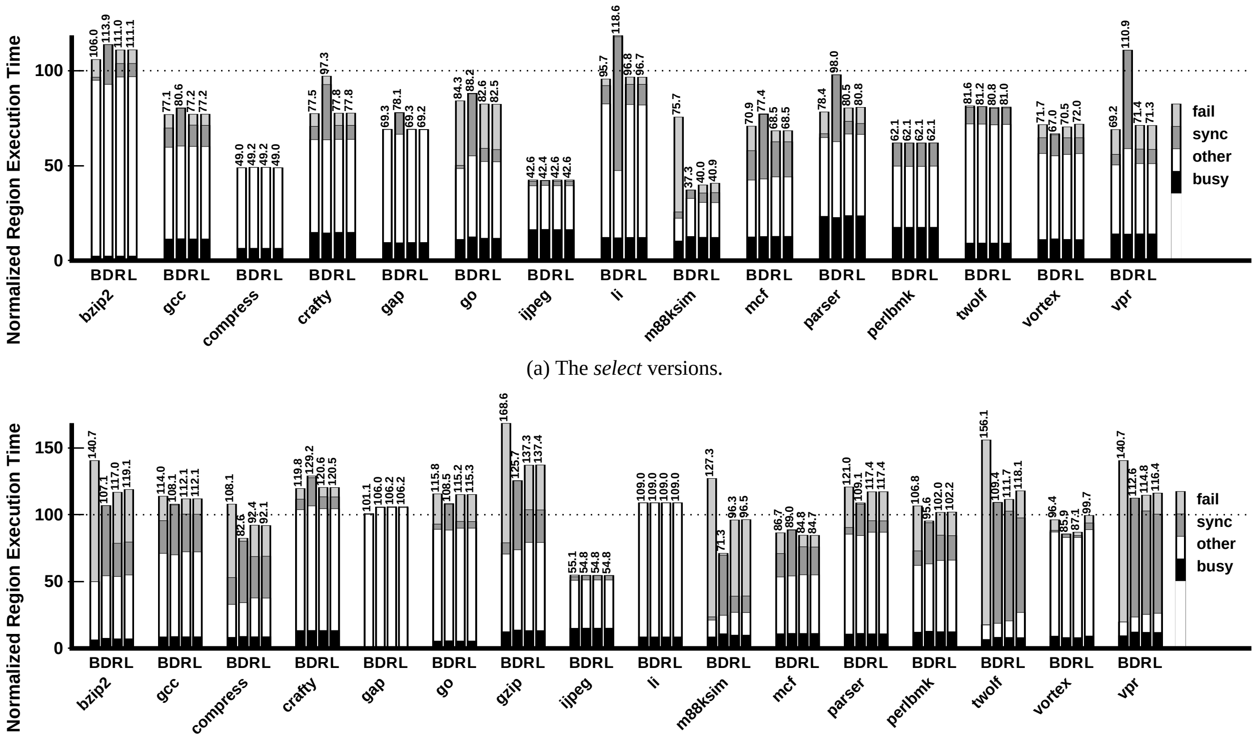
5.4.1 Hardware-Inserted Dynamic Synchronization

For many of our benchmarks, failed speculation is a significant performance limitation (especially for the *max-coverage* benchmarks); and as we observed in Section 5.3, prediction alone cannot eliminate all dependence violations. Our compiler has already inserted synchronization for local variables with cross-epoch data dependences (such as loop induction variables). Many dependences still remain, as demonstrated in Section 5.2.3, which can be synchronized dynamically by hardware.

Related Work

Dynamic synchronization has been applied to both uniprocessor and multiprocessor domains. Chrysos *et al.* [11] present a design for dynamically synchronizing dependent store-load pairs within the context of an out-of-order issue uniprocessor pipeline, and Moshovos *et al.* [54] investigate the dynamic synchronization of dependent store-load pairs in the context of the Multiscalar architecture [27, 65].

Both of these works differ from ours because they have the ability to forward a value directly from the store to the load in a dynamically-synchronized store-load pair: this is not difficult in a uniprocessor since the store and load issue from the same pipeline; for a multiprocessor like the Multiscalar, this requires that the memory location in question is implicitly forwarded from the producer to the consumer—functionality that is provided by the Multiscalar's *address-resolution buffer* [27]. As discussed and quantified in Sections 3.1.2 and 4.2.2, our scheme does not provide such support for implicit forwarding since it is not worth the complexity.



(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 5.12. Performance of dynamic synchronization. B **is the baseline experiment,** D **automatically synchronizes all violating loads (using a exposed load tables and violating loads lists of unlimited size),** R **builds on** D **by periodically resetting the violating loads list, and** L **refines** R **with tables that are realistic in size.**

Design Issues

Figure 5.11 illustrates how we dynamically synchronize two epochs. We obviously do not want to synchronize every load; instead, when a load is likely to cause a dependence violation, we can prevent speculation from failing by stalling the load until the previous epoch is complete: at that point, all modifications by logically-earlier epochs will be visible and the load can safely issue. We use the *exposed load table* and *violating loads list* described in Section 5.3.3 to identify the loads most likely to cause a violation that should therefore be synchronized. Again, this dynamic synchronization technique is not as aggressive as that described by Moshovos *et al.* [54] because our design does not support implicit forwarding.

Performance

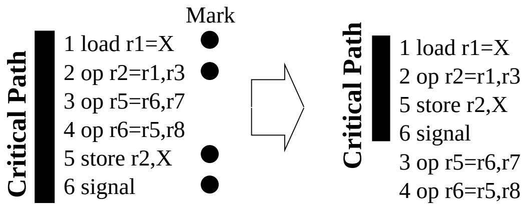
In Figure 5.12, experiment *D* shows the performance of dynamic synchronization where we have synchronized every load in the violating loads list. For now, we model tables/lists that are unlimited in size. For the *select* benchmarks we see that failed speculation has been replaced with synchronization as expected, resulting in improved performance for M88KSIM and VORTEX. However, nine benchmarks are now over-synchronized: we have unwittingly replaced successful speculation with synchronization as well, since some cases have infrequent dependence violations which trigger the dynamic synchronization mechanism. Overall, this technique has a negative performance impact, slowing down all benchmarks by an average of 6.0%.

For the *max-coverage* benchmarks, dynamic synchronization *D* results in a significant improvement for more than half of the benchmarks, and an average improvement of 16.4% across all benchmarks. COMPRESS, M88KSIM, and PERLBMK now speed up with this technique. Hence dynamic synchronization is important for limiting the negative performance impact of poorly performing regions that are selected for speculative parallelization by the compiler.

The fraction of speculation that fails for different speculative regions is a continuum—from speculation that fails rarely to constantly. A good scheme for dynamic synchronization must adapt to all of these cases by aggressively synchronizing those with frequent failed speculation and selectively synchronizing those with intermittent failed speculation, without resulting in the over-synchronization evident in the *D* experiment. To achieve this behavior, we periodically reset the violating loads list in experiment *R* (a period of 16 epochs on a single processor, not counting those for which speculation failed, was found to work well through experimentation), resulting in an improvement over the baseline (*B*) of 2.1% for the *select benchmarks*, and 11.8% for the *max-coverage* benchmarks. In the *L* experiment, we model tables/lists that are more realistic in size (exposed-load tables and violating loads lists that have 32 entries each). The resulting performance improvement over the baseline (*B*) is 1.9% for the *select benchmarks*, and 10.3% for the *max-coverage* benchmarks. In summary, dynamic synchronization is a promising technique for improving the performance of TLS.

5.4.2 Prioritizing the Critical Forwarding Path

In Section 5.3.4 we observed that even after aggressive prediction of *forwarded* values, synchronization is still an impediment to good speedup for some benchmarks. We call the instructions between the first use and the last definition of a forwarded value the *critical forwarding path*. When it is not possible to eliminate synchronization, an alternative is to instead prioritize *critical instructions* to help reduce the size and hence performance impact of the critical forwarding path. Our compiler already performs this optimization to the best of its ability, but there may be more that can be done dynamically by hardware at run-time. To the best of our knowledge, this is the first evaluation



(a) Prioritization.

Application	Issued Insts That Are High Priority and Issued Early	Improvement in Avg. Start-to-Signal Time (cycles)		
		Unprioritized	Prioritized	Speedup
BZIP2	4.4%	88.7	94.0	0.94
CC1	14.7%	136.8	127.0	1.08
COMPRESS95	8.2%	11.3	11.2	1.01
CRAFTY	8.9%	71.6	70.1	1.02
GAP	2.7%	116.8	111.5	1.05
GO	10.1%	101.7	100.0	1.02
IJPEG	12.2%	40.9	38.4	1.07
LI	1.6%	44.6	43.6	1.02
M88KSIM	5.9%	74.0	72.3	1.02
MCF	4.0%	75.6	69.1	1.09
PARSER	4.8%	145.4	138.7	1.05
PERLBMK	0.0%	35.5	35.3	1.01
TWOLF	7.2%	88.8	87.8	1.01
VORTEX	6.5%	126.7	148.8	0.85
VPR	20.3%	102.1	102.1	1.00

(b) Statistics.

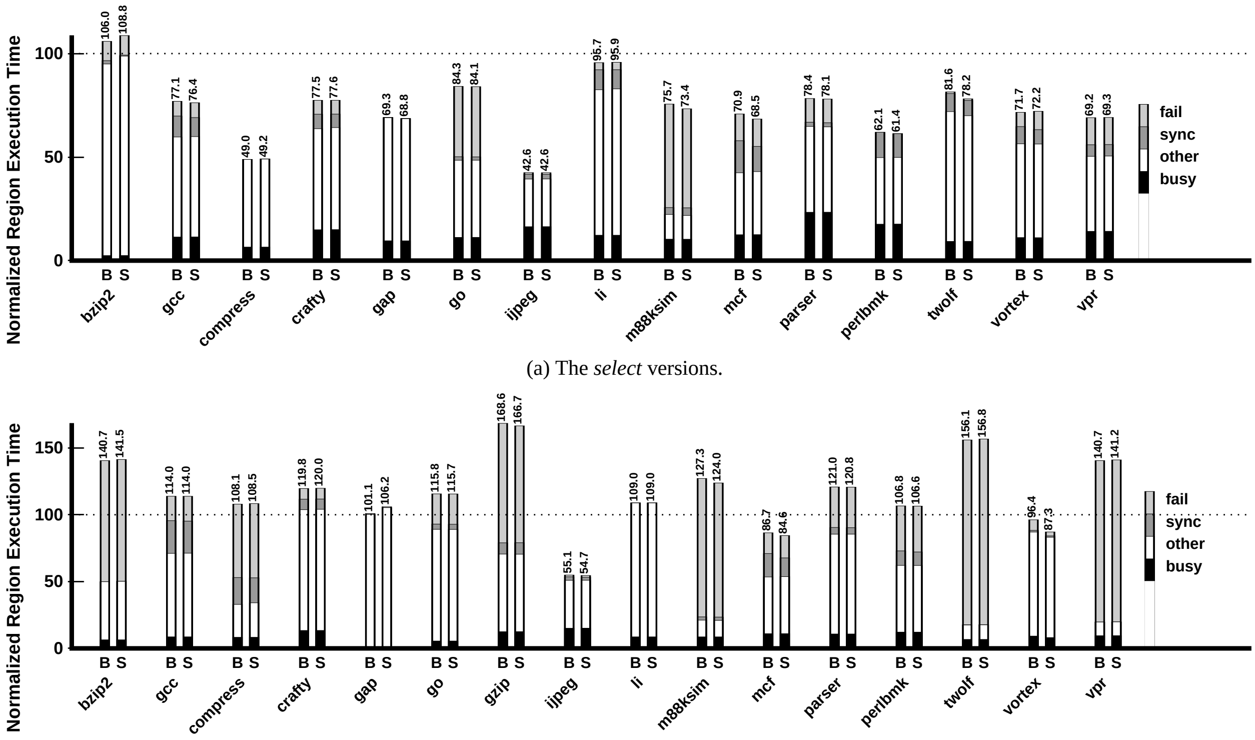
Figure 5.13. Prioritization of the critical forwarding path. We show (a) our algorithm, where we mark the instructions on the input chain of the critical store and the pipeline's issue logic gives them high priority; (b) some statistics, namely the fraction of issued instructions that are given high priority by our algorithm and issue early, and also the improvement in the average number of cycles from the start of the epoch until each signal.

of a hardware scheme for prioritizing the critical forwarding path for TLS.

Our hardware prioritization algorithm works as shown in Figure 5.13(a). We mark all instructions with register outputs on the input-chain of the critical store. We also track the critical forwarding path through memory, so that a critical load also depends on the store which produced the value for the given memory location. Ideally, we would also mark any instructions on the input-chain of an unpredictable conditional branch as being on the critical forwarding path, but this beyond the scope of this dissertation. The pipeline issue logic then gives priority to marked instructions so that the associated signal may be issued as early as possible. This algorithm could be implemented using techniques described by Fields *et al.* [23], but for now we focus on the potential impact.

The impact of prioritizing the critical forwarding path is shown in Figure 5.14. Note that we model a 128-entry reorder buffer (see Table 2.6), so the issue logic has significant opportunity to reorder prioritized instructions. Despite this fact, the impact on most applications is small. For the *select* benchmarks, the amount of synchronization time (*sync*) is reduced for GCC, MCF, TWOLF, and VORTEX. For VORTEX, however, this savings is negated by an increase in failed speculation (again, due to the corresponding increase in parallel overlap that exposes more inter-epoch data dependences). On average, both the *select* and *max-coverage* benchmarks improve by 0.6%.

To clarify the impact of prioritization, Figure 5.13(b) shows the fraction of issued instructions that are given high priority by our algorithm and also issue early, which averages 7.4% across all *select* benchmarks. Figure 5.13(b) also shows the change in the average number of cycles from the start of an epoch to the issue of each signal, for which the



(a) The *select* versions.

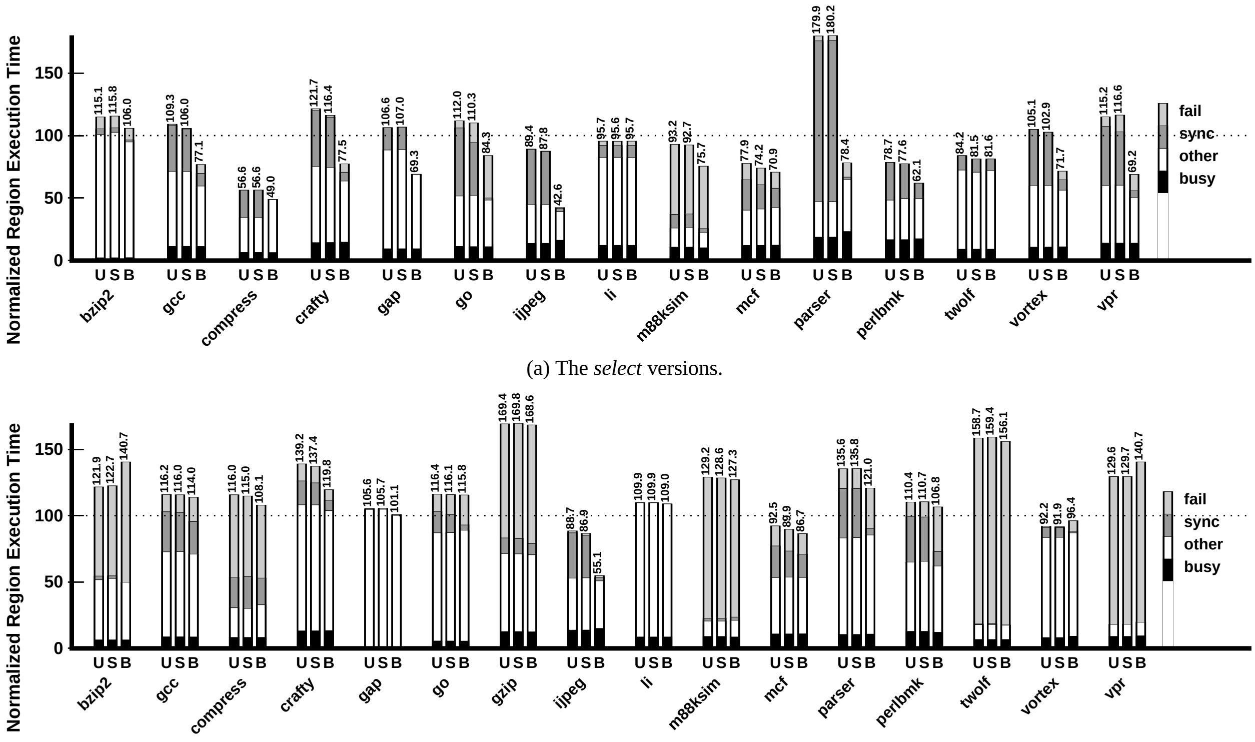
(b) The *max-coverage* versions.

Figure 5.14. Performance impact of prioritizing the critical forwarding path: B **is the baseline experiment, and** S **prioritizes the critical forwarding path.**

results are mixed: five benchmarks are improved by at least 5%, while 10 benchmarks are improved less than 5%.

As demonstrated in Section 5.2.1, our TLS compiler dramatically reduces the impact of synchronization by scheduling the critical forwarding path, limiting the remaining opportunity for hardware scheduling to have impact. We evaluate the impact of hardware prioritization without aggressive compiler scheduling in Figure 5.15. We observe that hardware prioritization (*S*) has more of an impact than before (when compiler scheduling was applied), the improvement averaging 1.4% across all *select* benchmarks. However, hardware prioritization is as effective as compiler scheduling (*B*) only for MCF; in most cases, compiler scheduling is much more effective.

Since the performance impact of hardware prioritization of the critical forwarding path is modest compared to the scheduling done by the compiler, the potential complexity of its implementation is not worth the cost. Hence scheduling the critical forwarding path for forwarded local variables is a task best suited to the compiler.



(a) The *select* versions.

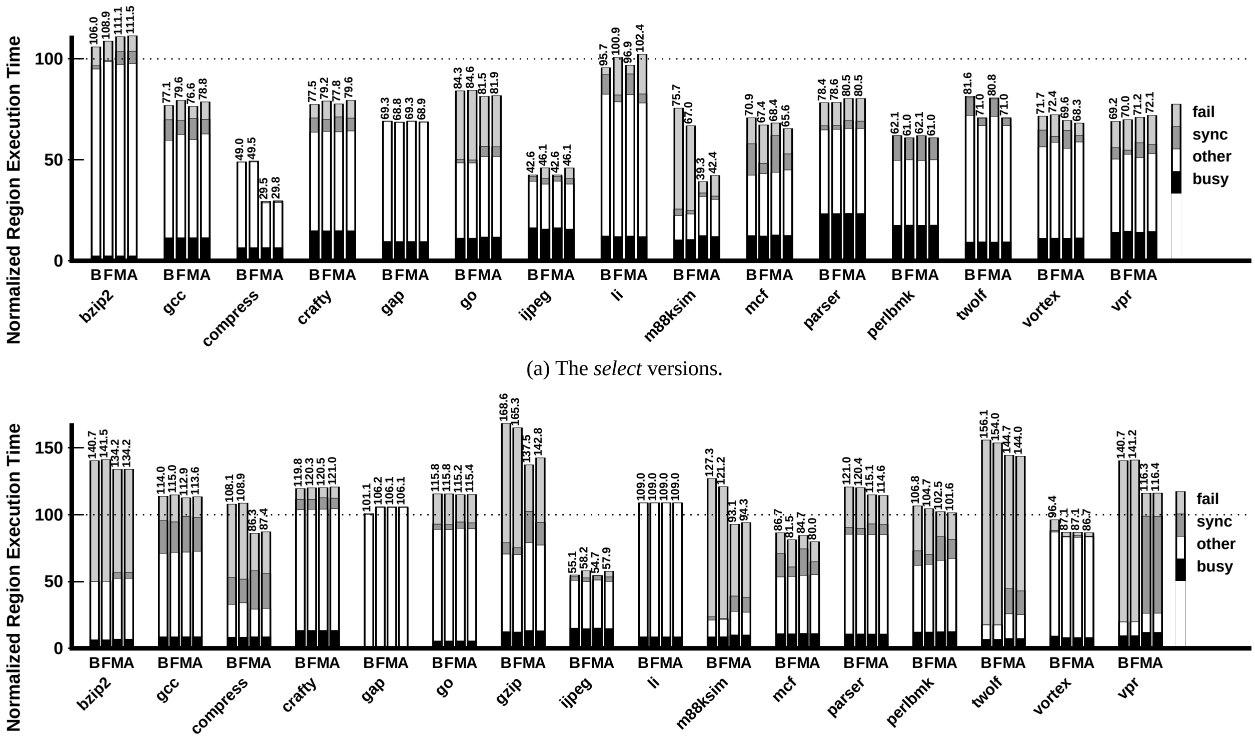
(b) The *max-coverage* versions.

Figure 5.15. Performance impact of prioritizing the critical forwarding path when it has not been scheduled by the compiler: *U* has not been scheduled by the compiler, *S* builds on *U* by prioritizing the critical forwarding path, and *B* is scheduled by the compiler.

5.5 Combining the Techniques

In this section, we evaluate the impact of all of our techniques combined. Most techniques are orthogonal in their operation with the exception of memory value prediction and dynamic synchronization: we only want to dynamically synchronize on memory values that are unpredictable. This cooperative behavior is implemented by having the dynamic synchronization logic check the prediction confidence for the load in question, and synchronizing only when confidence is low.

Figure 5.16 shows the performance impact of combining the various techniques, where the *B* experiment is our baseline, the *F* experiment represents the two techniques for improving communication of forwarded values (forwarded value prediction and scheduling the critical forwarding path), the *M* experiment represents the three techniques for improving the communication of memory values (memory value prediction, exploiting silent stores, and dynamic synchronization), and the *A* experiment shows all techniques combined. We model the realistic table/list sizes described in Section 5.3.3. For the *select* benchmarks, improving communication of forwarded values (*F*) is only effective for



(a) The *select* versions.

(b) The *max-coverage* versions.

Figure 5.16. Performance of all techniques combined. B **is the baseline experiment,** F **models techniques for optimizing forwarded values,** M **models techniques for optimizing memory values, and** A **models all techniques.**

M88KSIM and TWOLF, and yields a 0.4% improvement across all benchmarks (for the *max-coverage* benchmarks the average improvement is 0.8%). In contrast, improving communication of memory values is quite effective, resulting in significant performance gains for the *select* versions of COMPRESS and M88KSIM, as well as the *max-coverage* versions of COMPRESS, GZIP, M88KSIM, and VPR. On average, the *select* applications improve by 5.8% and the *max-coverage* applications improve by 7.0%. Finally, we observe that these techniques can be complementary: for example, the performance when all techniques are combined (*A*) is better than that of either group of techniques in isolation (i.e. *F* or *M*) for the *select* versions of MCF and VORTEX, and the *max-coverage* versions of PARSER and PERLBMK. A closer look at these two cases reveals that improving communication of forwarded values reduces (*F*) synchronization time (*sync*) but increases the amount of failed speculation (*fail*)—increasing the amount of parallel overlap also increases the number of out-of-order data dependences between epochs. Similarly, reducing the number of dependence violations by improving the communication of memory values (*M*) simply exposes more synchronization stall (*sync*). When both sets of techniques are combined, we attack both synchronization and failed speculation at the same time, resulting in improved performance for the two benchmarks.

Table 5.4. Summary of techniques for improving value communication.

Technique	Improves/Applies To	Requires Throttling	Complexity	Average Improvement		Maximum Improvement	
				select	*max-cov.*	*select*	*max-cov.*
forwarded value prediction	synchronization/forwarded values	yes	moderate	0.2%	0.3%	11.8%	5.0%
hardware prioritization	synchronization/forwarded values	no	high	0.6%	0.6%	4.1%	9.4%
memory value prediction	failed speculation/memory values	yes	moderate	-0.2%	2.7%	52.8%	20.6%
silent stores	failed speculation/memory values	no	low	0.9%	0.6%	40.0%	10.0%
dynamic synchronization	failed speculation/memory values	yes	low	1.9%	10.3%	46.0%	24.4%
all forwarded	synchronization/forwarded values	-	-	0.4%	0.8%	13.0%	9.7%
all memory	failed speculation/memory values	-	-	5.8%	7.0%	48.0%	26.9%
all	value communication/all values	-	-	5.4%	6.7%	44.0%	25.9%

5.6 Chapter Summary

We have shown that improving value communication in TLS can yield large performance benefits, and examined the techniques for taking advantage of this fact. Our analysis provides several important lessons. First, we discovered that prediction cannot be applied liberally when the cost of misprediction is high: predictors must be throttled to target only those dependences that limit performance. We observed that silent stores are prevalent, that replacing them with prefetches and loads can improve the performance of TLS execution, and requires significantly less hardware support than load value prediction (e.g., no value predictor is needed). We found that dynamic synchronization improves performance for many applications and can also mitigate the negative impact of poorly-performing speculative regions. We found that hardware prioritization to reduce the critical forwarding path does not work well, even though a significant number of instructions can be reordered.

Table 5.4 summarizes each technique and its respective performance impact. Of the five techniques for improving value communication, forwarded and memory value prediction and dynamic synchronization require careful throttling, while hardware prioritization and silent stores do not. Hardware prioritization is the most complex technique, while forwarded and memory value prediction are moderately complex (requiring predictors, tag lists and value tables), and silent stores and dynamic synchronization are the least complex. Averaging across all benchmarks, we observe that the techniques for improving the communication of memory values are more effective than those for forwarded values for both the *select* and *max-coverage* benchmark sets. Every technique has a significant impact on at least one application, with maximum improvements ranging from 4.1% to 42.8% for the *select* benchmarks, and 5.0% to 24.4% on the *max-coverage* benchmarks.

Looking at the performance of techniques when combined, we see evidence of complementary behavior in several cases. For the *max-coverage* benchmarks, the techniques that apply to forwarded values achieve a greater average speedup when combined compared to either technique in isolation. Similarly, the techniques that apply to memory values achieve a greater average speedup when combined for the *select* benchmarks. The maximum improvement is also greater for combined techniques than isolated techniques in many cases. Finally, in all cases the performance with all techniques combined is not as good as that of the memory value techniques alone, indicating that the techniques for forwarded values are interfering. Overall, we conclude that hardware techniques for improving the communication of *memory* values are effective, while improving the communication of *forwarded* local variables is a task best suited to the compiler.

Chapter 6

Conclusions

Architectures that naturally support multithreading (such as chip-multiprocessors) have become increasingly commonplace over the past decade, and this trend will likely continue in the near future. However, only workloads composed of parallel threads can take advantage of such processors. This dissertation proposes support for *Thread-Level Speculation* (TLS) which empowers the compiler to optimistically create parallel threads despite uncertainty as to whether those threads are actually independent; it puts forth the thesis that hardware support for thread-level speculation that is simple and efficient, and that scales to a wide range of multithreaded architectures can empower the compiler to improve the performance of sequential applications. This thesis is validated through a detailed evaluation of SPEC [16] integer and floating-point benchmarks, generated by a feedback-directed, fully-automated compiler, and measured on realistic, simulated hardware.

In the next section, we review in detail the contributions of this dissertation. We then discuss possible directions for future work, and finally conclude by summarizing the most important lessons to be taken from this research.

6.1 Contributions

This dissertation makes contributions in three major areas. The first is the proposal of a cooperative approach to TLS that capitalizes on the strengths of both the compiler and hardware. The second is the design and detailed evaluation of a unified scheme for TLS hardware support. Third, this dissertation presents a comprehensive evaluation of techniques for improving value communication between speculative threads.

6.1.1 A Cooperative Approach to TLS

In contrast with many previous approaches to TLS, this dissertation contributes a cooperative approach that exploits the respective strengths of compiler and hardware and ventures to redefine the interface between them. The compiler understands the structure of an entire program, but cannot easily predict its run-time behavior. In contrast, hardware operates on only limited windows of instructions but can observe all of the details of dynamic execution. Through new architected instructions that allow software to manage TLS execution, we free hardware from the burden of breaking programs into threads and tracking register dependences between them, while empowering the compiler to optimistically parallelize programs. This cooperative approach has many advantages over those that used either software or hardware in isolation, allowing the implementation of aggressive optimizations in the compiler, and minimizing the complexity of the underlying hardware support.

6.1.2 Unified Hardware Support for TLS

Previous approaches to TLS hardware support apply to either speculation within a chip-multiprocessor (or simultaneously-multithreaded processor) or to a larger system composed of multiple uniprocessor systems. The hardware support for TLS presented in this dissertation is unique because it scales seamlessly both within and beyond chip boundaries—allowing this single unified design to apply to a wide variety of multithreaded processors and larger systems that use those processors as building blocks. We demonstrate that tracking data dependences by extending invalidation-based cache coherence and using first-level data caches to buffer speculative state is both elegant and efficient, and that less aggressive designs, namely deep uniprocessor speculation and TLS support using only the load/store-queues, are insufficient to capture the same performance improvements as our approach. This dissertation also contributes a thorough evaluation of a wide range of architectural scales—chip-multiprocessors with varying numbers of processors and cache organizations, and larger multi-chip systems—as well as a detailed exploration of design alternatives and sensitivity to various architectural parameters.

6.1.3 A Comprehensive Evaluation of Techniques for Improving Value Communication Between Speculative Threads

This dissertation provides a comprehensive evaluation of techniques for enhancing value communication within a system that supports thread-level speculation, and demonstrates that many of them can result in significant performance gains. While these techniques are evaluated within the context of the implementation of TLS described in this dissertation, we expect to see similar trends within other TLS environments since the results are largely dependent on application behavior rather than the details of how speculation support is implemented. An important contribution is that these techniques are evaluated *after* the compiler has eliminated obvious data dependences and scheduled any critical forwarding paths, thereby removing the “easy” bottlenecks to achieving good performance. This leads us to very different conclusions than previous studies on exploiting value prediction within TLS [51, 56]. We demonstrate the importance of throttling back value prediction to avoid the high cost of misprediction, and propose and evaluate techniques for focusing prediction on the dependences that matter most. We also present the first exploration of how *silent stores* can be exploited within TLS. Finally, we evaluate two novel hardware techniques for enhancing the performance of synchronized dependences across speculative threads, but find that the compiler is better suited to optimizing the communication of forwarded values than hardware.

6.2 Future Work

Thread-level speculation remains a relatively new mode of execution for both compiler and architectural support, and there is plenty of research to be done beyond this dissertation. These topics are presented in the sections that follow, including future compiler research, investigation of hardware support for improving cache locality on private-cache CMPs, and exploration of techniques for online feedback and dynamic adaptation.

6.2.1 Compiler Support

Compiler support for TLS is still in its infancy. While our compiler infrastructure is sufficiently powerful and realistic to allow the detailed evaluation of hardware support presented in this dissertation, there are still many aspects which can be improved—this is the most abundant area for future work.

Speculative parallelization of regions of code other than loops warrants investigation. Other possibilities include procedure continuations (executing the code beyond a procedure call in parallel with the procedure call itself), loop continuations (executing the code beyond the end of a loop in parallel with the loop itself), simultaneously executing multiple loops in parallel, recursion, selection structures, and more. While other work has investigated speculation on structures other than loops [52, 56], the evaluations are of limited detail.

In Section 2.4 we discuss the issue of selecting which loops in a program to speculatively parallelize, and the impact of unrolling the selected loops. For the *select* benchmarks evaluated in this dissertation, we use a set of loops that maximize performance on our baseline hardware support, assuming that the compiler has access to complete profile information on the relative performance of each potentially-speculative loop with several different unrollings. This abundance of information would not likely be available to a real compiler, hence there is a need for the compiler to be more independent of profile information. A more comprehensive compiler infrastructure would combine state-of-the-art techniques for traditional automatic parallelization and pointer analyses, and also benefit from new kinds of analyses to statically estimate dynamic behavior.

As reported in Section 2.5.1, a major source of overhead in our transformed benchmarks is due to the insertion of `gcc` "`asm`" statements which represent new TLS instructions: the compiler is forced to be conservative, resulting in a dramatic increase in the number of spills, and optimization is hindered in other ways as well. A back-end which understands TLS instructions (which for the most part should not hamper optimization) would be able to produce code that is significantly more efficient.

For many of the hardware techniques discussed in this dissertation, there exist potential software counterparts. For example, in Section 5.4.2, we compare the optimization of the critical forwarding path through both hardware prioritization and compiler scheduling; in this case, we find that compiler scheduling is much more effective. One could also envision software versions of value prediction and dynamic synchronization, which could eliminate the need for additional hardware tables. Furthermore, compiler-inserted prefetching could be beneficial, especially for speculatively-parallelized loops executing on a private-cache CMP.

Another aspect of compilation for TLS with a potentially large impact is the layout of data in memory. In Section 4.3.4 we demonstrate that the *layout* of pages in the DSM system for TLS on multi-node architectures can have a large impact on performance, and could possibly be dealt with by the compiler. Furthermore, the compiler could also reduce false dependences through manipulation of the alignment of data structures in memory. For example, in an array of structures, each structure could be cache-line aligned, eliminating any false dependences between epochs that access adjacent structures.

6.2.2 Hardware Support for Improving Cache Locality

While it is desirable for the speculative buffering for TLS hardware support to be distributed (e.g., the private data caches in a chip-multiprocessor), one consequence is reduced data locality compared to the original sequential execution. As demonstrated in Section 3.4.2, decreased cache locality is one of most significant overheads of TLS execution after failed speculation and synchronization (which were addressed in Chapter 5). Techniques exist, such as prefetching, which can potentially be implemented in hardware to improve cache locality and hence performance for TLS execution on distributed-cache hardware support.

6.2.3 Online Feedback and Dynamic Adaptation

Another potentially important area for future TLS research is in improving the cooperation between software in hardware at run-time. As a simple example, consider a speculatively-parallelized region of code for which speculation constantly fails. In such a case, we would like the hardware to suggest that software cease speculation and instead continue sequentially. A trivial implementation of this feedback would use performance counters that track the fraction of epochs that are violated. However, failed speculation is not the only overhead that can cause a speculative execution to slow down with respect to the sequential execution—other overheads include synchronization and cache locality. A more sophisticated approach would allow the estimation of true performance for both the sequential and speculative executions. Such information could allow more involved dynamic adaptation than simply discontinuing unsuccessful speculation. First, this information could aid in the decision of where to speculate in the first place. Second, as observed in Sections 3.4 and 4.3 for some applications, increasing the number of processors beyond a certain point can yield diminishing returns or even degrade performance; this suggests that for every speculative region there exists an optimal number of processors, which could potentially be obtained dynamically through sampled performance feedback.

Appendix A

Full Coherence Scheme

In this appendix, we present the full details of the speculative coherence scheme, including support for TLS within a chip-multiprocessor with a private- and/or shared-cache architecture, as well as beyond a chip for multiprocessor systems that use chip-multiprocessors as building blocks.

A.1 Line State in the Cache

The invalidation-based cache coherence scheme that we extend can be in one of the following states: invalid (*I*), exclusive (*E*), shared (*S*), or dirty (*D*). The invalid state indicates that the cache line is no longer valid and should not be used. The shared state denotes that the cache line is potentially resident in some other cache, while the exclusive state indicates that this is the only cached copy. The dirty state denotes that the cache line has been modified and must be written back to external memory. When a processor attempts to write to a cache line, exclusive access must first be obtained—if the line is not already in the exclusive state, upgrade requests must be sent to all other caches which contain a copy of the line, thereby invalidating those copies.

To detect data dependences and to buffer speculative memory modifications, we extend the standard set of cache line states to include the seven new states as described in Table A.1. The new states denote four orthogonal properties of a cache line: whether it is dirty; whether it has been speculatively-loaded (*SpL*); whether it has been speculatively-modified (*SpM*); and whether it is exclusive (*E*) versus shared (*S*)[1].

[1] Note that in Section 3.2.2

Table A.1. Shared cache line states.

State	Description
I	invalid
E	exclusive
S	shared
D	dirty (implies exclusive)
DSpL	dirty and speculatively-loaded (implies exclusive)
SpLE	speculatively-loaded exclusive
SpLS	speculatively-loaded shared
SpME	speculatively-modified exclusive
SpMS	speculatively-modified shared
SpLME	speculatively-loaded and modified exclusive
SpLMS	speculatively-loaded and modified shared

Although these properties are orthogonal, some combinations are not allowable, e.g., dirty and speculatively-modified. When a cache line is dirty, the cache owns the only up-to-date copy of that cache line, and must preserve it from speculative modifications so that the line can eventually be supplied to the rest of the memory system. Conversely, when a cache line is in the speculatively-modified state, we may need to discard any speculative modifications to the line if speculation ultimately fails. Since it would be difficult to isolate both dirty and speculatively-modified portions of the same line in a traditional hardware cache (especially if these portions can overlap), it is difficult to allow both of these states to co-exist.

Maintaining the notion of exclusiveness is important since a speculatively-modified cache line that is exclusive (*SpME* or *SpLME*) does not require upgrade requests to be sent out when side-effects are committed to memory. It is also interesting to note that the states *SpMS* and *SpLMS* imply that the cache line is both speculatively-modified and shared. This means that it is possible for more than one modified copy of a cache line to exist as long as no more than one copy is non-speculative and the rest of the copies are speculative.

The dirty and speculatively-loaded state (*DSpL*) indicates that a cache line is dirty and that the cache line is the only up-to-date copy. Since a speculative load cannot corrupt the cache line, it is safe to delay writing the line back until a speculative store occurs.

For speculation to succeed, any cache line with a speculative state must remain in the cache (or be preserved in a victim cache) until the corresponding epoch becomes *homefree*; speculative modifications may not be propagated to the rest of the memory hierarchy, and cache lines that have been speculatively-loaded must be tracked in order to detect whether data dependence violations have occurred. If a speculative cache line must be replaced then this is

Table A.2. Processor-initiated actions.

Action	Description
PRM	processor read miss
PRH	processor read hit
PWM	processor write miss
PWH	processor write hit
PRMSp	processor read miss speculative
PRCMSp	processor read conflict-miss speculative (a logically-later epoch has already modified the same cache line)
PRHSp	processor read hit speculative
PWMSp	processor write miss speculative
PWCMSp	processor write conflict-miss speculative (another epoch has already speculatively-modified the same cache line)
PWHSp	processor write hit speculative

treated as a violation, causing speculation to fail and the epoch to be re-executed.

A.2 Processor Actions

We now describe an implementation of an invalidation-based cache coherence scheme extended to detect data dependence violations. First, we list all possible actions that are originated by the processors, the cache controller, or the external memory system. Table A.2 lists the possible actions which are originated by the processor. Processor-initiated actions are divided into reads and writes, hits and misses, and speculative and non-speculative accesses.

Misses are further divided into regular misses and conflict misses. For states other than invalid (*I*), a regular miss indicates that the current cache line must be replaced. A conflict-miss indicates that two different epochs—executing on processors that physically share the cache or on one multithreaded processor—have accessed the same cache line in an unacceptable manner. The following scenarios differentiate acceptable access patterns from those that are unacceptable in a shared cache.

- Two different epochs may both speculatively read the same cache line.

- If an epoch speculatively-modifies a cache line, only a logically-later epoch may read that cache line afterwards. This *implicit forwarding* effectively allows us to forward speculative modifications between two properly ordered epochs. If a logically-earlier epoch attempts to read the cache line, a read conflict-miss will result.

- Only one epoch may speculatively-modify a given cache line. If an epoch attempts to speculatively-modify a

Table A.3. Actions generated by the shared cache controller.

Sent To	Action	Description
External	G.ER	Generate external read.
	G.EREx	Generate external read exclusive.
	G.EWb	Generate external writeback (the line is no longer cached).
	G.EU	Generate external update-line (like a write, except the line remains cached).
	G.EUp	Generate external upgrade request (request for ownership, copy of line not required).
	G.EUpSp	Generate external upgrade request speculative (request for ownership which may not be granted, copy of line not required).
	G.ERExSp	Generate external read exclusive speculative.
Shared Cache Controller	G.Viol	Generate violation (a definite violation).
	G.Suspend	Generate a suspend (a violation which may be avoided by suspending).
	G.Combine	Combine this cache line with the external copy (if combining is not supported then ignore this action).
	G.ORB	Add current tag to ORB (ownership required buffer).
	G.FlushORB	For each tag in the ORB (ownership required buffer) generate an EUpSp. If cache line combining is supported, otherwise an EUp. If any actions follow, they must wait until G.FlushORB completes.
	G.Progress	If epoch E, which has speculatively-modified the cache line, is logically-later than the current epoch then violate epoch E, otherwise G.Suspend for the current epoch (ensuring forward progress).
Conditions	Ack=Excl	Acknowledgement from the previous action indicates that the cache line is exclusive.
	Exposed	False if the epoch stored to the memory location before the load occurred, true otherwise.
	Older	True if the epoch which generated the action is older (logically-earlier) than the current epoch.
	Replicate	True if the cache line may be replicated in the local cache (the actions that follow apply to the replicated cache line).

cache line that has already been speculatively-modified by a different epoch, a write conflict miss (*PWCMSp*) results.

A.3 Cache Actions

Table A.3 describes all actions generated by the cache controller. The first group are actions which are sent to the external memory system. There are two possible write actions: (i) a writeback (*G.EWb*) which sends a copy of the cache line and indicates that the line is no longer cached; and (ii) an update (*G.EU*) which also sends a copy of the cache line but implies that the line is still cached. If a line is not owned exclusively and a processor attempts to write to the line, exclusive ownership must first be obtained using the upgrade request (*G.EUp*) action. An upgrade request thus differs from a read exclusive request by not requiring that a copy of the line to be sent with the acknowledgement.

There are also two speculative actions which may be sent to the external memory system. There is a speculative version of the upgrade request action which piggybacks the epoch number of the requester (*G.EUpSp*). This action is unlike any regular coherence action because it is a hint and not a definite command—i.e. a speculative upgrade request is not guaranteed to provide exclusiveness. This is useful when an epoch tries to speculatively-modify a cache line that a logically-earlier epoch has already speculatively-modified. We do not want the logically-earlier epoch to give up its copy (and thus fail), so the upgrade request may not give the requesting epoch exclusive ownership—success of the upgrade request is indicated in an acknowledgement. There is also a speculative version of read exclusive (*G.ERExSp*) which has the same properties as a speculative upgrade request, except that a copy of the cache line is provided with the acknowledgement.

The next group of actions are performed at the cache controller. The violation action (*G.Viol*) indicates that speculation has failed and must recover. The (*G.Suspend*) action indicates that a violation is about to occur, and the only way to avoid it is to suspend the thread. For example, if a speculative cache line is about to be replaced, the thread which caused the replacement could be suspended until the epoch which owns the speculative cache line becomes homefree. If a suspension is not supported, the *G.Suspend* action may be conservatively replaced with a real violation (*G.Viol*).

If two epochs speculatively-modify the same cache line, there are two ways to resolve the situation. One option is to simply violate the logically-later epoch. Alternatively, we could allow both epochs to modify their own copies of the cache line and combine them with the real copy of the cache line as they commit, as is done in a multiple-writer coherence protocol [5, 10]. The action *G.Combine* indicates that the current cache line should be combined with the copy stored at the next level of caching in the external memory system. If combining is not supported, the *G.Combine* action is simply ignored.

When an epoch becomes homefree, it may allow its speculative modifications to become visible to the rest of the system. However, the epoch must first acquire ownership of all cache lines that are speculatively-modified but not in an exclusive state. Since a search over the entire cache for such cache lines would take far too long and delay passing the homefree token, we instead store the addresses of cache lines requiring ownership in an *ownership required buffer* (ORB). The *G.ORB* action adds the current cache line address to the ORB.

When an epoch becomes homefree, it generates an upgrade request for each entry in the ORB, as described by the *G.FlushORB* action. If cache line combining is supported, *G.FlushORB* may instead generate speculative upgrade

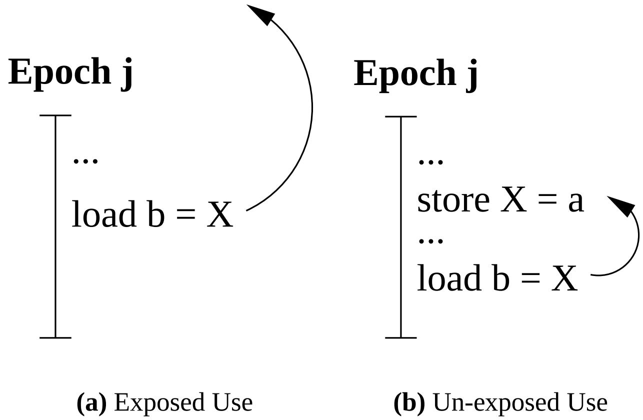


Figure A.1. Exposed and un-exposed uses.

requests for each line address in the ORB. Since write-after-write (WAW) dependences are not true dependences they may be eliminated through renaming, and therefore the logically-later of two epochs which both speculatively-modified the same cache line does not need to be violated: the speculative modifications may be combined later. A speculatively-modified cache line may not change to the dirty state until *G.FlushORB* completes.

The two conflict-misses described previously in Table A.2 do not necessarily have to result in failed speculation. If cache line replication is supported, certain violations can be avoided. However, when replication is not possible we must be careful which epoch is suspended or violated, since the wrong choice could result in deadlock or even livelock. The action *G.Progress* only suspends or violates logically-later epochs, ensuring that speculation makes forward progress. If epoch E which speculatively-modified the cache line is logically-later than the current epoch, then *G.Progress* violates epoch E; otherwise, *G.Progress* performs *G.Suspend* for the current epoch.

The last group of actions are actually conditions evaluated by the coherence mechanism. Since we want to maintain the most exact exclusiveness information possible, external upgrade requests will indicate in the acknowledgement whether exclusiveness is obtained (*Ack=Excl*).

If *fine-grain* speculatively-modified (SM) bits exist (as described in Section 3.3.4), we can detect whether a load is an *upwards exposed use*—i.e. the use of a location without a prior definition of that location by the same epoch [3]. In Figure A.1(a) we see that the load is exposed since the location p has not already been defined by the epoch. Conversely, in Figure A.1(b) we see that the load is not exposed since the location p has been defined by the a previous store in the same epoch. We only have to consider a location to be speculatively-loaded if it is an upwards exposed use, otherwise the load cannot cause a data dependence violation. To differentiate these cases, we will check

Table A.4. Other actions.

Action	Description
ER	External read.
EREx	External read exclusive (copy of line is supplied with ack).
EI	External invalidate.
EUp	External upgrade request (copy of line is not supplied with ack).
ERExSp	External read exclusive speculative (copy of line is supplied with ack).
EUpSp	External upgrade request speculative (copy of line is not supplied with ack).
HFree	Epoch has become homefree.
Viol	Epoch has committed a violation or been cancelled.
→X	Transition to new state X.
(A)?(B):(C)	If A then B else C.

the condition *Exposed*.

A key function of our speculative coherence scheme is the ability to decide whether a speculative coherence action originated from a logically-earlier epoch or a logically-later epoch. The *Older* condition is true if the epoch which generated the action is logically-earlier than the current epoch and false otherwise.

Finally, the condition *Replicate* is true if a cache line is successfully replicated, where successful replication means that another copy of the same cache line (with the same cache tag) may be created within the same associative set. If cache line replication is not supported, then *Replicate* is always false.

A.4 Other Actions

Table A.4 describes several miscellaneous actions. The first group are actions which are received from the external memory system. The action *HFree* indicates that an epoch has received the homefree token and has processed any pending incoming coherence actions; hence, at this point memory is consistent with the rest of the system, and the epoch is guaranteed not to have violated any data dependences with logically-earlier epochs and can therefore commit all of its speculative modifications by changing their cache states to dirty (*D*).

The action *Viol* indicates that the current epoch has either committed a violation or has been cancelled. All cache lines which have been speculatively-modified must be invalidated (changed to the invalid state), and all other speculative cache lines may be changed back to an appropriate non-speculative state.

A.5 State Transition Diagram

We describe the coherence scheme for supporting TLS using a state transition diagram, given in Table A.5. For each current shared cache line state and each possible action we give the appropriate response actions and the transition to the new state.

We now investigate several "*action* × *state*" pairs of interest.

- Some actions cannot occur in a given state. For example, *PRH* × *I* cannot occur since an invalid cache line cannot yield a hit. A conflict-miss like *PWCMSp* can only occur if the cache line has been speculatively-modified. By definition, a conflict miss occurs when another epoch, sharing the same cache, has already speculatively-modified the cache line in question—i.e. it is in one of the states *SpME*, *SpMS*, *SpLME*, or *SpLMS*.

- An example of the basic detection of a read-after-write dependence violation is illustrated by *EUpSp* × *SpLS*. If the epoch which generated the speculative upgrade request is older than the current epoch, then a dependence violation has occurred and the processor is notified (*G.Viol*).

- This version of the coherence scheme is implemented with the objective of slowing down a non-speculative thread as little as possible. For this reason, a cache line in a non-speculative state is not invalidated when a speculative upgrade request occurs, as shown by *EUpSp* × *E*. Alternatively, the cache line could be relinquished in order to give exclusiveness to the speculative thread, possibly eliminating the need for that thread to obtain ownership when it becomes homefree. These two options are analyzed experimentally in Section 3.5.3.

- *PWHSp* × *D* generates an update (*G.EU*), ensuring that the only up-to-date copy of a cache line is not corrupted with speculative modifications. Conversely, *PRHSp* × *D* simply changes to the dirty and speculatively-loaded state (*DSpL*), since the cache line will not be corrupted by a speculative load.

- *PRMSp* × *SpME* results in a *G.Suspend*: the cache line which has been speculatively-modified must be replaced to continue, and this is not allowable. We may either violate the epoch, or suspend until the epoch becomes homefree at which point we may allow the speculative modifications to be written-back to the external memory system.

Table A.5. Cache state transition diagram (Continued on next page). →X represents the transition to new state X, and (A)?(B):(C) denotes `if A then B else C`.

Action	Cache Line State	
	I	E
PRM	G.ER; (Ack=Excl)?(→E):(→S);	G.ER; (Ack=Excl)?(→E):(→S);
PRH	-	→E;
PWM	G.EREx; →D;	G.EREx; →D;
PWH	-	→D;
PRMSp	G.ER; (Ack=Excl)?(→SpLE):(→SpLS);	G.ER; (Ack=Excl)?(→SpLE):(→SpLS);
PRCMSp	-	-
PRHSp	-	→SpLE;

PWMSp	G.ERExSp;
	(Ack=Excl)?(→SpME):(G.ORB; →SpMS);
		G.ERExSp;
		(Ack=Excl)?(→SpME):(G.ORB; →SpMS);

PWCMSp	-	-
PWHSp	-	→SpME;
ER	-	→S;
EREx	-	→I;
EI	-	→I;
EUp	-	→I;
ERExSp	-	→S;
EUpSp	-	→S;
HFree	→I; G.FlushORB;	→E; G.FlushORB;
Viol	→I;	→E;

Action	Cache Line State	
	S	D
PRM	G.ER; (Ack=Excl)?(→E):(→S);	G.EWb; G.ER; (Ack=Excl)?(→E):(→S);
PRH	→S;	→D;
PWM	G.EREx; →D;	G.EWb; G.EREx; →D;
PWH	G.EUp; →D;	→D;

PRMSp	G.ER; (Ack=Excl)?(→SpLE):(→SpLS);	G.EWb;
		G.ER; (Ack=Excl)?(→SpLE):(→SpLS);

PRCMSp	-	-
PRHSp	→SpLS;	→DSpL;

PWMSp	G.ERExSp;
	(Ack=Excl)?(→SpME):(G.ORB; →SpMS);
		G.EWb; G.ERExSp;
		(Ack=Excl)?(→SpME):(G.ORB; →SpMS);

PWCMSp	-	-

PWHSp	G.EUpSp;
	(Ack=Excl)?(→SpME):(G.ORB; →SpMS);
		G.EU; →SpME;

ER	→S;	G.EU; →S;
EREx	→I;	G.EWb; →I;
EI	→I;	G.EWb; →I;
EUp	→I;	G.EWb; →I;
ERExSp	→S;	G.EU; →S;
EUpSp	→S;	→D;
HFree	→S; G.FlushORB;	→D; G.FlushORB;
Viol	→S;	→D;

Table A.5. Cache state transition diagram (Continued on next page).

Action	Cache Line State		
	DSpL	SpLE	SpLS
PRM	G.Suspend;	G.Suspend;	G.Suspend;
PRH	→DSpL;	→SpLE;	→SpLS;
PWM	G.Suspend;	G.Suspend;	G.Suspend;
PWH	G.Viol;	G.Viol;	G.Viol;
PRMSp	G.Suspend;	G.Suspend;	G.Suspend;
PRCMSp	-	-	-
PRHSp	→DSpL;	→SpLE;	→SpLS;
PWMSp	G.Suspend;	G.Suspend;	G.Suspend;
PWCMSp	-	-	-
PWHSp	G.EU; →SpLME;	→SpLME;	G.EUpSp; (Ack=Excl)? (→SpLME):(G.ORB; →SpLMS);
ER	G.EU; →SpLS;	→SpLS;	→SpLS;
EREx	G.EWb; G.Viol;	G.Viol;	G.Viol;
EI	G.EWb; G.Viol;	G.Viol;	G.Viol;
EUp	G.EWb; G.Viol;	G.Viol;	G.Viol;
ERExSp	G.EU; →SpLS;	(Older)?(G.Viol):(→SpLS);	(Older)?(G.Viol):(→SpLS);
EUpSp	G.EU; →SpLS;	(Older)?(G.Viol):(→SpLS);	(Older)?(G.Viol):(→SpLS);
HFree	→D; G.FlushORB;	→E; G.FlushORB;	→S; G.FlushORB;
Viol	→D;	→E;	→S;

Action	Cache Line State	
	SpME	SpMS
PRM	G.Suspend;	G.Suspend;
PRH	G.Viol;	G.Viol;
PWM	G.Suspend;	G.Suspend;
PWH	G.Viol;	G.Viol;
PRMSp	G.Suspend;	G.Suspend;
PRCMSp	(Replicate)?(→SpLS):(G.Progress);	(Replicate)?(→SpLS):(G.Progress);
PRHSp	(Exposed)?(→SpLME):(→SpME);	(Exposed)?(→SpLMS):(→SpMS);
PWMSp	G.Suspend;	G.Suspend;
PWCMSp	(Replicate)?(→SpMS):(G.Progress);	(Replicate)?(→SpMS):(G.Progress);
PWHSp	→SpME;	→SpMS;
ER	G.ORB; →SpMS;	→SpMS;
EREx	G.Viol;	G.Viol;
EI	G.Viol;	G.Viol;
EUp	G.Viol;	G.Viol;
ERExSp	G.ORB; →SpMS;	→SpMS;
EUpSp	G.ORB; →SpMS;	→SpMS;
HFree	G.FlushORB; →D;	G.FlushORB; G.Combine; →D;
Viol	→I;	→I;

- *ER* × *SpLME* demonstrates the case when exclusive ownership of a cache line which has been speculatively-modified is lost. The tag for this cache line is added to the ORB by the action *G.ORB* so that ownership may be obtained quickly when the epoch becomes homefree.

Table A.5. Cache state transition diagram (Continued from previous page).

	Cache Line State	
Action	**SpLME**	**SpLMS**
PRM	G.Suspend;	G.Suspend;
PRH	G.Viol;	G.Viol;
PWM	G.Suspend;	G.Suspend;
PWH	G.Viol;	G.Viol;
PRMSp	G.Suspend;	G.Suspend;
PRCMSp	(Replicate)?(→SpLMS):(G.Progress);	(Replicate)?(→SpLMS):(G.Progress);
PRHSp	→SpLME;	→SpLMS;
PWMSp	G.Suspend;	G.Suspend;
PWCMSp	(Replicate)?(→SpLMS):(G.Progress);	(Replicate)?(→SpLMS):(G.Progress);
PWHSp	→SpLME;	→SpLMS;
ER	G.ORB; →SpLMS;	→SpLMS;
EREx	G.Viol;	G.Viol;
EI	G.Viol;	G.Viol;
EUp	G.Viol;	G.Viol;
ERExSp	(Older)?(G.Viol):(G.ORB; →SpLMS);	(Older)?(G.Viol):(→SpLMS);
EUpSp	(Older)?(G.Viol):(G.ORB; →SpLMS);	(Older)?(G.Viol):(→SpLMS);
HFree	G.FlushORB; →D;	G.FlushORB; G.Combine; →D;
Viol	→I;	→I;

- *HFree* × *SpLME* demonstrates waiting for *G.FlushORB* to complete, which guarantees ownership of all speculatively-modified cache lines, before changing to the dirty (*D*) state. In *HFree* × *SpLMS*, after waiting for *G.FlushORB* to complete, the speculatively-modified cache line is combined with the current external copy before changing to the dirty state.

A.6 Coherence in the External Memory System

Coherence with support for speculation in the external memory system is quite similar to regular coherence. As listed in Table A.4, we require the following standard coherence actions: read (*ER*), read-exclusive (*EREx*), invalidate (*EI*), and upgrade requests (*EUp*). A read is a request for a copy of the cache line, and a read-exclusive is a request for a copy of the cache line as well as ownership; an upgrade request does not require a copy of the cache line. An invalidation, which is used to maintain inclusion, causes the cache to relinquish the appropriate cache line.

Two new speculative coherence actions are supported by the external memory system: read-exclusive speculative (*ERExSp*) and upgrade request speculative (*EUpSp*). Both of these actions behave similarly to their non-speculative counterparts with the exception of two important distinctions. First, the epoch number of the requester is piggybacked along with the request in both cases, so the receiver can make decisions based on the relative ordering of the requesting epoch. Second, both actions are only hints and do not compel the cache to relinquish ownership. As long as these signals are propagated, this layer of the coherence scheme may be applied recursively to deeper levels of the external memory system.

It is possible to re-design the speculative coherence scheme so that speculative messages are not used—this has the advantage that the underlying coherence mechanisms are not modified in any way, and that only the cache state and cache controllers must be extended to support TLS. The performance impact of this design option is evaluated in Section 3.5.2.

Bibliography

[1] W. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of ISCA 27*, June 2000.

[2] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[4] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *MICRO-31*, December 1998.

[5] C. Amza, S. Dwarkadas A.L. Cox, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proceedings of the Third High Performance Computer Architecture Conference*, pages 261–271, February 1997.

[6] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[7] Scott E. Breach, T. N. Vijaykumar, Sridhar Gopal, James E. Smith, and Gurindar S. Sohi. Data memory alternatives for multiscalar processors. Technical Report CS-TR-1997-1344, Computer Sciences Department, University of Wisconsin-Madison, 1996.

[8] Broadcom Corporation. The Sibyte SB-1250 Processor. http://www.sibyte.com/mercurian.

[9] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *International Symposium on Computer Architecture*, 1999.

[10] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[11] G. Chrysos and J. Emer. Memory dependency prediction using store sets. In *Proceedings of the 25th ISCA*, June 1998.

[12] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of ISCA 27*, June 2000.

[13] M. Cintra and J. Torrellas. Learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th HPCA*, 2002.

[14] C. B. Colohan, A. Zhaia, J. G. Steffan, and T. C. Mowry. Compiling sequential programs for a thread level speculative architecture. In *Proceedings of Some Conference*, Month 2003.

[15] Compaq Computer Corporation. Alpha 21264/ev67 microprocessor hardware reference manual.

[16] Standard Performance Evaluation Corporation. The SPEC Benchmark Suite. Technical report. http://www.spechbench.org.

[17] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, 1986.

[18] Sandhya Dwarkadas, Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of ISCA 20*, pages 144–155, May 1993.

[19] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors.

[20] J. Emer. Ev8: The post-ultimate alpha.(keynote address). In *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[21] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *IEEE Computer*, February 2002.

[22] M. Farrens, G. Tyson, , and A.R. Pleszkun. A study of single-chip processor/ cache organizations for large number of transistors. In *Proceedings of ISCA 21*, pages pp. 338–347, 1994.

[23] Brian A. Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *ISCA 2001*, 2001.

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

[25] M. Frank, C. Moritz, B. Greenwald, S. Amarasinghe, and A. Agarwal. Suds: Primitive mechanisms for memory dependence speculation. Technical Report MIT/LCS Technical Memo LCS-TM-591, January 1999.

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

[27] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[28] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report EE Department TR #1080, Technion–Israel Institute of Technology, 1996.

[29] Maria Jesus Garzaran, Milos Prvulovic, Jose Maria Llaberia, Victor Vinals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *In Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, February 2003.

[30] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[31] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[32] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *In Proceedings of Supercomputing 1998*, November 1998.

[33] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing '98*, November 1998.

[34] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of ASPLOS-VIII*, October 1998.

[35] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of ISCA 20*, 1993.

[36] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. D. Loreto, P. Hontalas, P. LaRouche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the time warp operating system. In *In Proceedings of the 11th Annual ACM Symposium on Operating System Principles*, pages 77–93, November 1987.

[37] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of ISCA 17*, pages 364–373, May 1990.

[38] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum '99*, October 1999.

[39] T. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[40] Jens Knoop and Oliver Ruthing. Lazy code motion. In *Proc. ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, 92.

[41] V. Krishnan and J. Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[42] Venkata Krishnan and Josep Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[43] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions Database Systems*, 6(2):213–226, June 1981.

[44] 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 1992.

[45] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[46] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[47] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th ISCA*, pages 241–251, June 1997.

[48] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of ISCA 27*, June 2000.

[49] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, 1996.

[50] P. Marcuello and A. Gonzlez. Clustered Speculative Multithreaded Processors. In *Proc. of the ACM Int. Conf. on Supercomputing*, June 1999.

[51] P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. In *International Symposium on Microarchitecture*, November 1999.

[52] Pedro Marcuello and Antonio González. Thread-Spawning Scheme for Speculative Multithreading. In *Proceedings of the 8th HPCA*, February 2002.

[53] G. Morrisett and M. Herlihy. Optimistic Parallelization. Technical Report CMU-CS-93-171, School of Computer Science, Carnegie Mellon University, October 1993.

[54] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. June 1997.

[55] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of ASPLOS-VII*, October 1996.

[56] J. Oplinger, D. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[57] D. Padua, D. Kuck, and D. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computing*, September 1980.

[58] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-1996-1328, University of Wisconsin-Madison, 1996.

[59] M. Prvulovic, M. Garzaran, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelizatoin. In *proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[60] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops With Privatization and Reduction Parallelization. In *Proceedings of PLDI '95*, pages 218–232, June 1995.

[61] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[62] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of Micro 30*, 1997.

[63] P. Rundberg and P. Stenstrom. Low-cost thread-level data dependence speculation on multiprocessors. In *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*, December 2000.

[64] Y. Sazeides and J. E. Smith. The Predictability of Data Values. *Proceedings of Micro 13*, pages 248–258, December 1997.

[65] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of ISCA 22*, pages 414–425, June 1995.

[66] J. G. Steffan, C. B. Colohan, A. Zhaia, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of ISCA 27*, June 2000.

[67] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. *Languages and Compilers for Parallel Computing*, pages 137–151. Springer-Verlag, Berlin, Germany, 1992.

[68] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. *HotChips '99*, August 1999.

[69] J.-Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, 48(9), September 1999.

[70] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA 22*, pages 392–403, June 1995.

[71] Dean Tullsen, Susan Eggers, Joel Emer, Henry Levy, Jack Lo, and Rebecca Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA 1996*, May.

[72] J. Veenstra. `MINT+` mips emulator. Personal communication.

[73] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Os support for improving data locality on cc-numa compute servers. In *In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[74] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *International Symposium on Microarchitecture*, 1997.

[75] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.

[76] A. Zhaia, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of ASPLOS-X*, October 2002.

[77] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[78] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Fifth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 135–141, January 1999.

[79] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–200, June 1995.