Hardware Support for Thread-Level Speculation

Thesis Summary

J. Gregory Steffan

Thursday, Feb 27 2003, 2pm, Weh4623

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

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.

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,

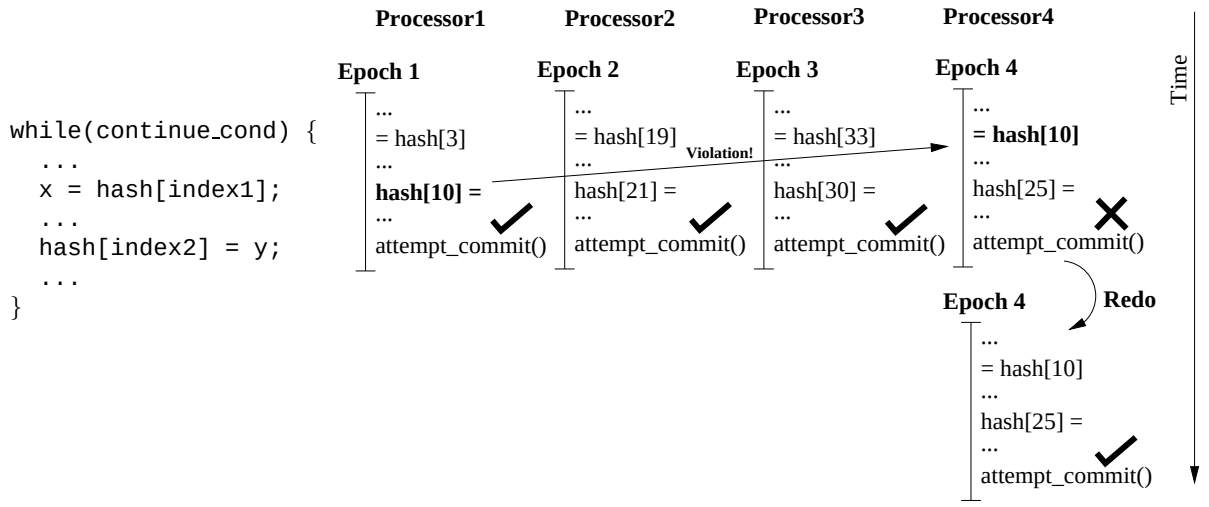


Figure 1. Example of thread-level speculation (TLS).

making large and highly interconnected designs infeasible [1, 17]. 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 [7]. 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 [21], IBM Power4 [9], and the Sibyte SB-1250 [3] are all *chip-multiprocessors* (CMPs), in that they incorporate multiple processors on a single die. Alternatively, the Alpha 21464 [6] supports *simultaneous multithreading* (SMT) [22], 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 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. Instead, 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 run-time. To illustrate how TLS works, consider the simple `while` loop in Figure 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(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 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 scales 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.

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 scale 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.

1.3 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.

1.3.1 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.

1.3.2 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.

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. Glossary of terms.

1.3.3 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 [14, 15]. 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.

2 TLS Execution model

This section describes the execution model for TLS that is targeted by the compiler and implemented in hardware. 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`. 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 centralized work queue—however, this second approach is less scalable than the distributed approach.

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. 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.

3 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 [20] 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.

3.1 Deciding Where to Speculate

One of the most important tasks in a thread-speculative system is deciding which portions of code to speculatively parallelize [4]. For the evaluations in this paper, 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 paper. 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 paper.

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 on our baseline hardware support for TLS, and select loops to maximize performance where we select the loops that contribute the greatest performance gain. The best performing unrolling factor is used for each loop, and chosen independently for the sequential and speculative versions of each application.

3.2 Transforming to Exploit TLS

Once speculative regions are chosen, the compiler inserts the new TLS instructions that interact with hardware to create and manage the speculative threads and forward values.

3.3 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 [10] 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 [24].

3.4 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.

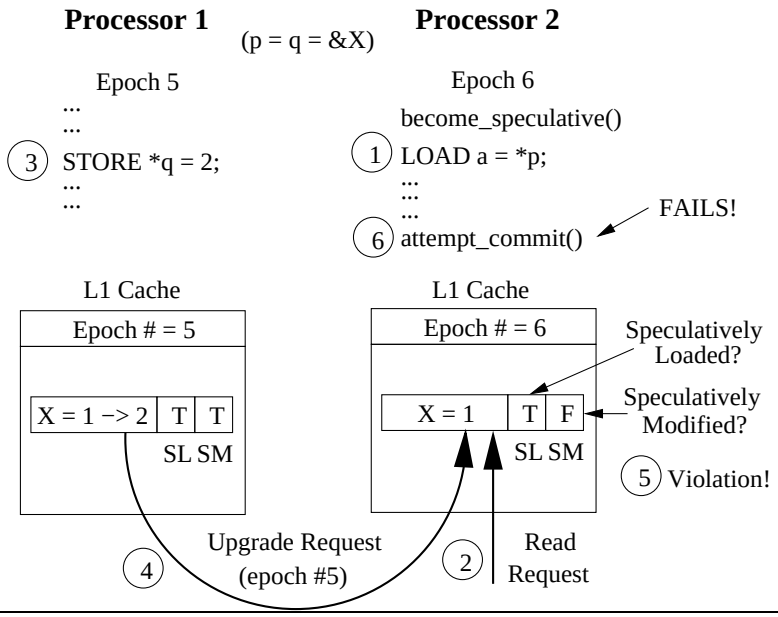


Figure 3. Using cache coherence to detect a RAW dependence violation.

4 Support for TLS in a Chip Multiprocessor

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.

Oneapproach 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.

4.1 The Basic Idea

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, 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 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.

Table 1. 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%

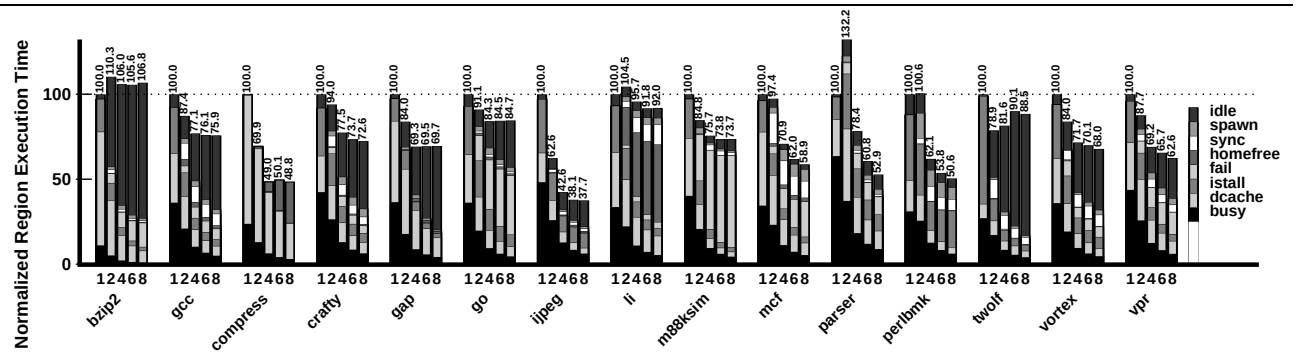


Figure 4. Varying the number of processors from one through eight. The baseline architecture has four processors.

4.2 Evaluation

Table 1 summarizes the performance of each application on our baseline architecture, which is a four-processor chip-multiprocessor that implements our baseline coherence scheme. Throughout this paper, 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 1, 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 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.

Figure 4 shows execution time normalized to that of the original sequential version; each of the bars are broken down into eight segments explaining what happened during all potential graduation slots.[1] 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.

[1]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.

Figure 4 shows how performance varies across a number of different processors. 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 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.

4.3 Summary

Our approach to TLS hardware support 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 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 section, 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.

5 Support for Scalable Thread-Level Speculation

The previous section 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 section 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 [11]) 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.

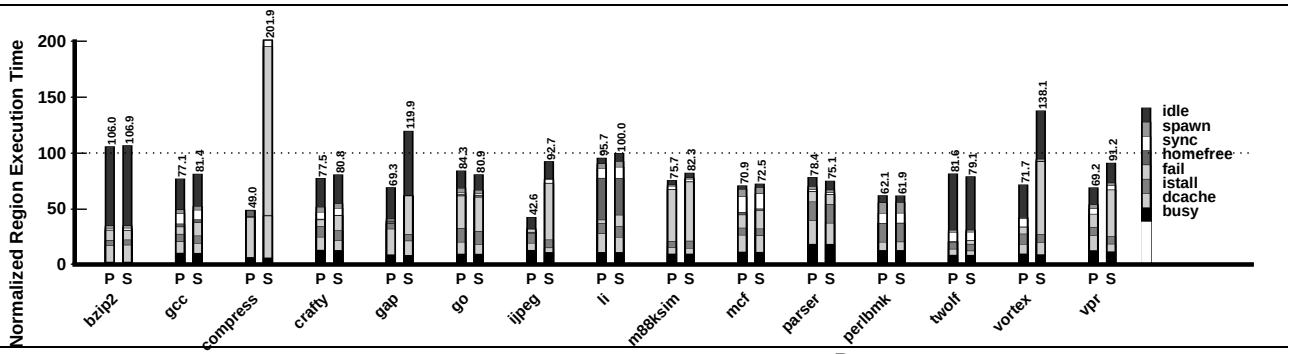


Figure 5. Region performance on both private-cache and shared-cache architectures. P is speculatively executed on a 4-processor CMP with private caches, and S is speculatively executed on a 4-processor CMP with a shared cache.

5.1 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) [22] 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.[2]

5.1.1 Performance of Shared Data Cache Support for TLS

We compare the performance of both private-cache and shared-cache support for TLS in Figure 5. *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.

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. In the dissertation I evaluate several mechansism to reduce these conflicts and hence improve performance.

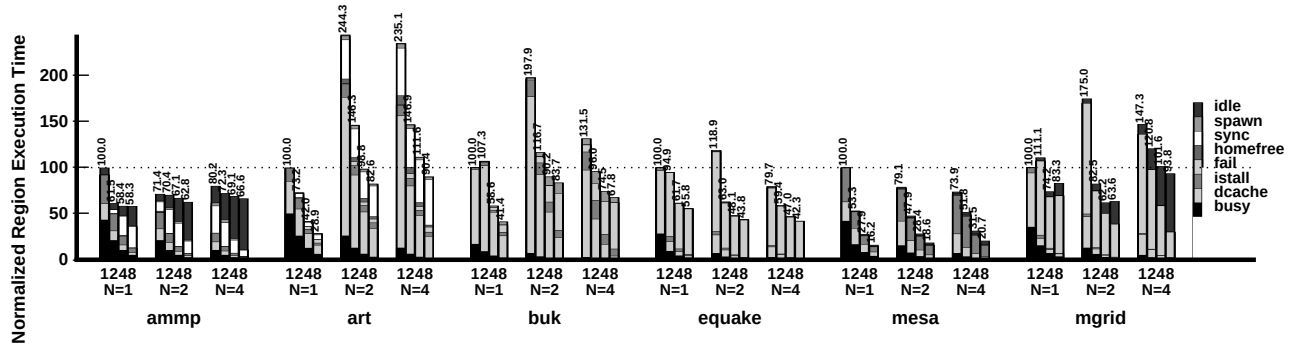
5.2 Scaling Beyond Chip Boundaries

Our scheme scales naturally 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.

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 6 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.

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.

[2]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 paper.



(a) Full detail.

Figure 6. 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.

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 [9] 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 ; 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 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.

5.3 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 section 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.

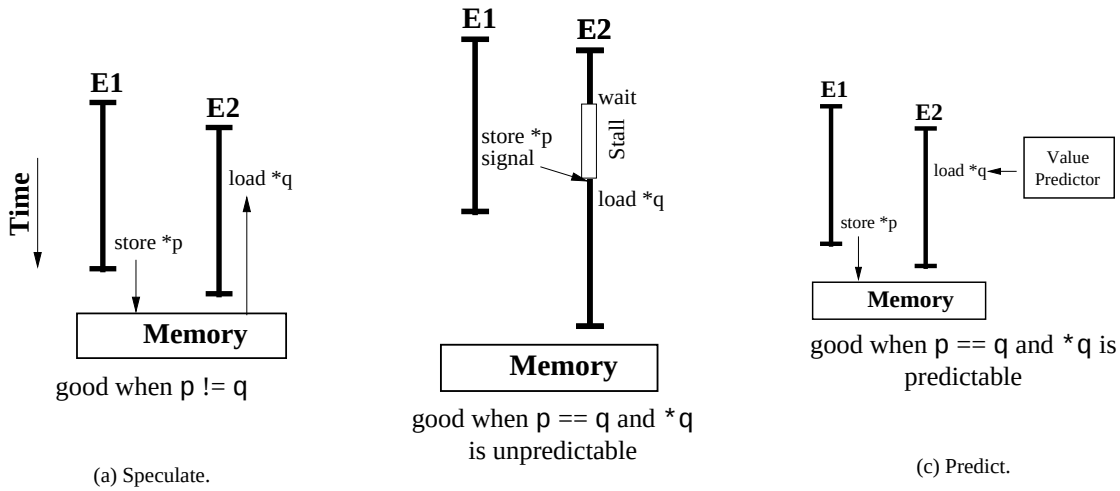


Figure 7. A memory value may be communicated between two epochs (E1 and E2) through (a) speculation, (b) synchronization, or (c) prediction.

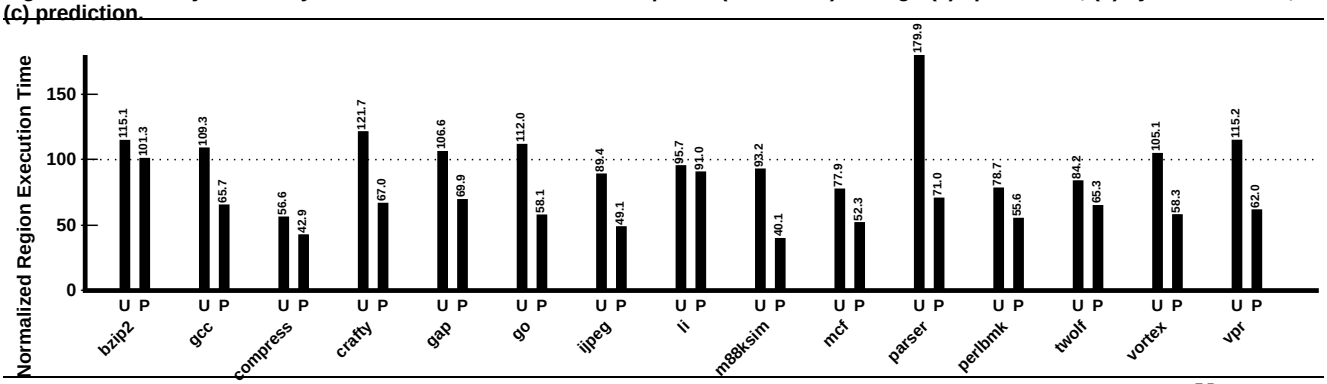


Figure 8. Potential impact of optimizing value communication. Relative to the normalized, original sequential version, U shows the unoptized speculative version and P shows perfect prediction of all inter-thread data dependences.

6 Improving Value Communication

6.1 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 section we show how to apply value prediction, dynamic synchronization, and hardware instruction prioritization to improve value communication and hence performance for TLS.

6.1.1 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 7(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[3] between epochs (Figure 7(b)). Since a parallelized region of code contains many variables, the compiler employs a combination of speculation and synchronization as appropriate.

[3]This is also known as *DOACROSS* [5, 16] parallelization.

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* [2, 8, 13, 14, 18, 19, 23], as illustrated in Figure 7(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 8 shows the impact of perfect prediction on the performance of both the *select* and *max-coverage* benchmarks, evaluated on a 4-processor CMP that implements our TLS scheme. 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 8, 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.

6.1.2 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 8? Figure 7 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) [24], and speculates on everything else. How can we use hardware to improve on this approach? Our focus in this section 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 7(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 *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* [12], 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 7(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 7(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 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 9. We implement both compiler and hardware methods for reducing the critical forwarding path.

7 Summary of Results for Improving Value Communication

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

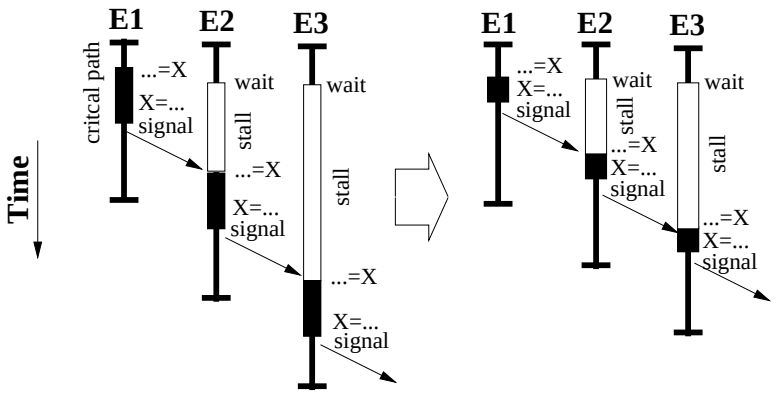


Figure 9. Reducing the critical forwarding path.

Table 2. Summary of techniques for improving value communication.

Technique	Improves/Applies To	Requires Throttling	Complexity	Average Improvement		Maximum Improvement	
				select	*max-coverage*	*select*	*max-coverage*
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%

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 2 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.

References

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

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

[4] 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.

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

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

[7] 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.

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

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

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

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

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

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

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

[15] 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.

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

[17] 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.

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

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

[20] 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.

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

[22] 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.

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

[24] 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.