A Scalable Approach to Thread-Level Speculation

J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{*steffan,colohan,zhaia,tcm*}@*cs.cmu.edu*

Abstract

While architects understand how to build cost-effective parallel machines across a wide spectrum of machine sizes (ranging from within a single chip to large-scale servers), the real challenge is how to easily create parallel software *to effectively exploit all of this raw performance potential. One promising technique for overcoming this problem is* Thread-Level Speculation (TLS)*, which enables the compiler to optimistically create parallel threads despite uncertainty as to whether those threads are actually independent. In this paper, we propose and evaluate a design for supporting TLS that seamlessly scales to any machine size because it is a straightforward extension of writeback invalidation-based cache coherence (which itself scales both up and down). Our experimental results demonstrate that our scheme performs well on both single-chip multiprocessors and on larger-scale machines where communication latencies are twenty times larger.*

1. Introduction

Machines which can simultaneously execute multiple parallel threads are becoming increasingly commonplace on a wide variety of scales. For example, techniques such as *simultaneous multithreading* [23] (e.g., the Alpha 21464) and *single-chip multiprocessing* [16] (e.g., the Sun MAJC [21] and the IBM Power4 [10]) suggest that thread-level parallelism may become increasingly important even within a single chip. Beyond chip boundaries, even personal computers are often sold these days in two or four-processor configurations. Finally, high-end machines (e.g., the SGI Origin [14]) have long exploited parallel processing.

Perhaps the greatest stumbling block to exploiting all of this raw performance potential is our ability to automatically convert single-threaded programs into parallel programs. Despite the significant progress which has been made in automatically parallelizing regular numeric applications, compilers have had little or no success in automatically parallelizing highly *irregular* numeric or especially *non-numeric* applications due to their complex control flow and memory access patterns. In particular, it is the fact that memory addresses are difficult (if not impossible) to statically predict—in part because they often depend on run-time inputs and behavior—that makes it extremely difficult for the compiler to statically prove whether or not potential threads are independent. One architectural technique which may help us overcome this problem is *thread-level speculation*.

1.1 Thread-Level Speculation

Thread-**L**evel **S**peculation (**TLS**) [9, 18, 20] allows the compiler to automatically parallelize portions of code in the presence of statically ambiguous data 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 reexecuting any iterations which do suffer dependence violations—could potentially speed up this loop significantly, as illustrated in Figure 1(b). Here 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. This example demonstrates the basic principles of TLS—it can also be applied to regions of code other than loops.

In this example we assume that the program is running on a shared-memory multiprocessor, and that 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. 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. Any *violation* of the data dependences imposed by this original program order is detected at run-time through our TLS mechanism. Finally, 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*. We provide this guarantee by passing a *homefree token* at the end of each epoch. Further examples of the use of thread-level speculation, and an exploration of the interface between TLS hardware and software, can be found in an earlier publication [19].

(a) *Example psuedo-code*

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

(b) *Execution using thread-level speculation*

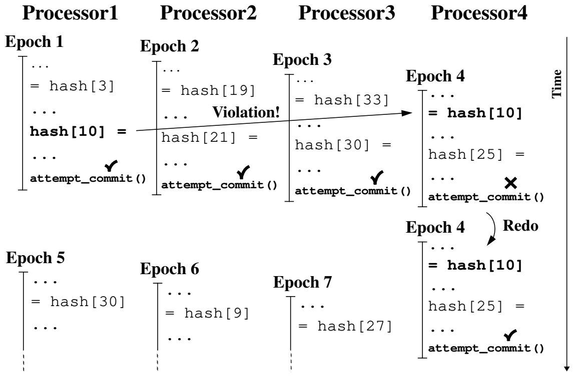


Figure 1. Example of thread-level speculation.

1.2 Related Work

Knight was the first to propose hardware support for a form of thread-level speculation [12]; his work was within the context of functional languages. The Multiscalar architecture [6, 18] was the first complete design and evaluation of an architecture for TLS. There have since been many other proposals which extend the basic idea of thread-level speculation [2, 7, 8, 9, 13, 15, 17, 20, 22, 25]. In nearly all of these cases, the target architecture has been a very tightly coupled machine—e.g., one where all of the threads are executed within the same chip. These proposals have often exploited this tight coupling to help them track and preserve dependences between threads. For example, the Stanford Hydra architecture [9] uses special write buffers to hold speculative modifications, combined with a write-through coherence scheme that involves snooping these write buffers upon every store. While such an approach may be perfectly reasonable within a single chip, it was not designed to scale to larger systems.

One exception (prior to this publication) is a proposal by Zhang *et al.* [25] for 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).

Concurrent with our study, Cintra *et al.* [5] have proposed using a hierarchy of MDTs (Memory Disambiguation Tables) 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, since we separate ordering from physical location through explicit software-managed epoch numbers and integrate the tracking of dependence violations directly into cache coherence (which may or may not be implemented hierarchically), our speculation occurs along a single flat *speculation level* (described later in Section 2.2), and does not impose any ordering or scheduling constraints on the threads.

1.3 Objectives of This Study

The goal of this study is to design and evaluate 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 [23], single-chip multiprocessors [16, 21], more traditional shared-memory multiprocessors of any size [14], and even multiprocessors built using software distributed shared-memory [11]. Our approach scales (both up and down) to all of these architectures because it is built upon writeback invalidation-based cache coherence, which itself scales to any of these machines. Our unified approach to supporting thread-level speculation offers the following advantages. First, we could build a large-scale parallel machine using either single-chip multiprocessors or simultaneously-multithreaded processors as the building blocks, and seamlessly perform thread-level speculation across the entire machine (or any subset of processors within the machine). Second, once we compile a program to exploit thread-level speculation, it can run directly on any of these machines without being recompiled. We demonstrate this in our our experimental results: the same executables (`buk` and `equake`) exploit our unified thread-level speculation mechanism to achieve good speedup not only on a single-chip multiprocessor, but also on multi-chip multiprocessors (where inter-chip communication latencies are 20 times larger).

The remainder of this paper is organized as follows. Section 2 describes how invalidation-based cache coherence can be extended to detect data dependence violations, and Section 3 gives a possible hardware implementation of this scheme. We describe our experimental framework in Section 4, evaluate the performance of our scheme in Section 5, and conclude in Section 6.

2. A Coherence Scheme For Scalable Thread-Level Speculation

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 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 not statically known.

Our solution is to leverage invalidation-based cache coherence. Recall that under invalidation-based cache coherence, a processor must first invalidate other cached copies of a line to get exclusive ownership before it can modify that line. The key insight in our scheme is that we can extend these existing invalidation messages to detect data dependence violations by noticing whenever an invalidation arrives from a *logically-earlier* epoch for a line that we have *speculatively* loaded in the past.

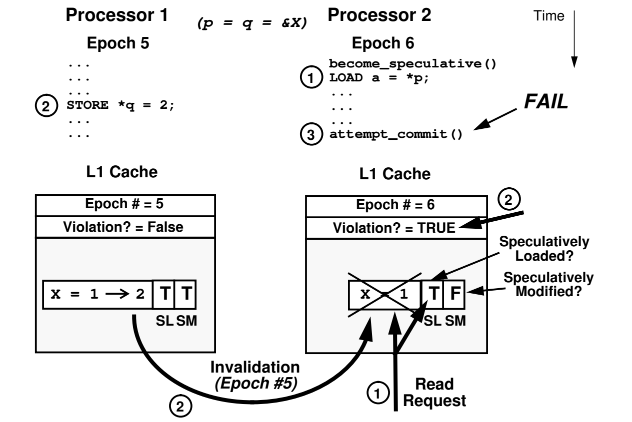


Figure 2. Using cache coherence to detect a RAW dependence violation.

2.1 An Example

To illustrate the basic idea behind our scheme, we show 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 2, we augment the state of each cache line to indicate whether the cache line has been speculatively loaded (SL) and/or speculatively modified (SM). For each cache, we also maintain a logical timestamp (called an *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; *epoch 6* is notified by setting the *violation flag*. As we will show, the full coherence scheme must handle many other cases, but the overall concept is analogous to this example.

In the sections that follow, we define the new speculative cache line states and the actual cache coherence scheme, including the actions which must occur when an epoch becomes *homefree* or is notified that a violation has occurred. We begin by describing the underlying architecture assumed by the coherence scheme.

2.2 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 single-chip multiprocessors, etc.).

For simplicity, we assume that the shared-memory architecture supports an invalidation-based cache coherence scheme where all hierarchies enforce the inclusion property. Figure 3(a) shows

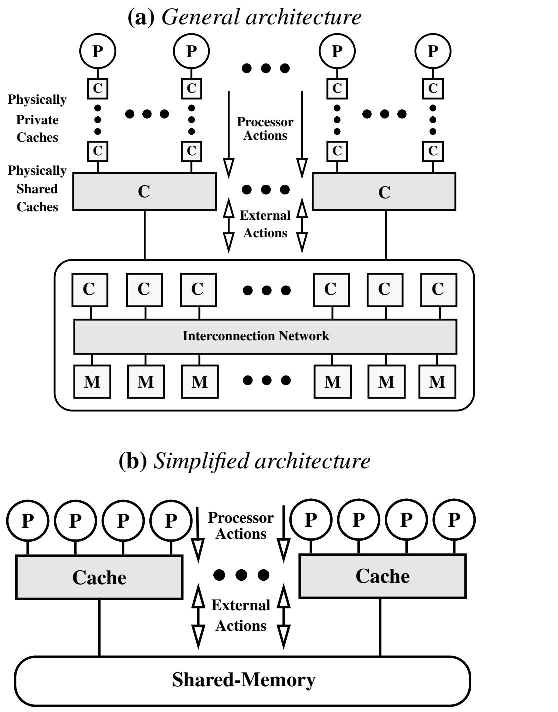


Figure 3. Base architecture for the TLS coherence scheme.

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 physically private 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(a) is not necessary for the purposes of describing our cache coherence scheme. Instead, Figure 3(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.

2.3 Overview of Our Scheme

The remainder of this section describes the important details 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

(a) *Cache line states*

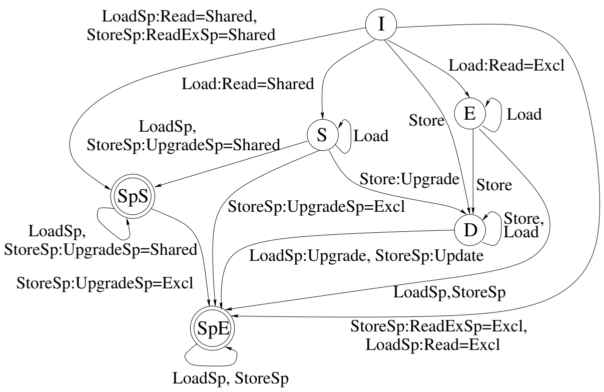
State	Description
I	Invalid
E	Exclusive
S	Shared
D	Dirty
SpE	Speculative (SM and/or SL) and exclusive
SpS	Speculative (SM and/or SL) and shared

(b) *Coherence messages*

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.
Flush	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.

(c) *Responses to processor events*



(d) *Responses to external coherence events*

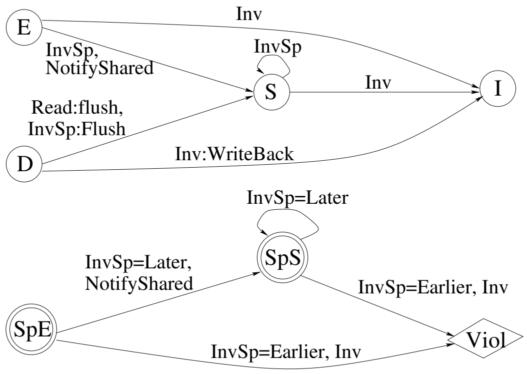


Figure 4. Our coherence scheme for supporting thread-level speculation.

(iii) an ordering of all speculative memory references (provided by epoch numbers and the *homefree token*). Following the description of our baseline scheme, we will discuss some additional support that can potentially improve its performance.

2.4 Cache Line States

A cache line in a basic 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 4(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. 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*).

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. Previous work has shown that a 16KB, 2-way set-associative cache along with a four-entry victim cache is sufficient to avoid nearly all failed speculation due to replacement [20].

2.5 Coherence Messages

To support thread-level speculation, we also add the three new speculative coherence messages shown in Figure 4(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).

2.6 Baseline Coherence Scheme

Our coherence scheme for supporting TLS is summarized by the two state transition diagrams shown in Figures 4(c) and 4(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, we now describe a few highlights of how we extend it to support TLS.

2.6.1 Some Highlights of Our 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 clearly the speculation fails. In contrast, if an *invalidation-speculative* arrives, then a violation only occurs if it is from a *logically-earlier* epoch.

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 a *flush* to ensure that the only up-to-date copy of the cache line is not corrupted with speculative modifications. For simplicity, we also generate a *flush* when a speculative load accesses a *dirty* cache line (we describe later in Section 2.7 how this case can be optimized).

A goal of this version 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 5.4.

2.6.2 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.

For each line in the *speculative-shared* (*SpS*) state that has been speculatively modified (i.e. the *SM* flag is set), 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.

2.6.3 When Speculation Fails

When speculation fails for a given epoch, any speculatively modified lines must be invalidated, and any speculatively loaded 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.

2.7 Performance Optimizations

We now present several methods for improving the performance of our baseline coherence scheme.

Forwarding Data Between Epochs: Often regions that we would like to parallelize contain predictable data dependences between epochs. We can avoid violations due to these dependences by inserting wait–signal synchronization. After producing the final value of a variable, an epoch signals the logically-next epoch that it is safe to consume that value. 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.

Dirty and Speculatively Loaded State: As described for the baseline scheme, when a speculative load or store accesses a *dirty* cache line we generate a *flush*, 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 that we evaluate later in Section 5.

Suspending Violations: Recall that if a speculatively accessed 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.) Alternatively, we could *suspend* the epoch until it becomes *homefree*, at which point we can safely allow the replacement to occur since the line is no longer speculative.

Support for Multiple Writers: If two epochs speculatively modify the same cache line, there are two ways to resolve the situation. One option is to simply squash the logically-later epoch, as is the case for our baseline scheme. 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 [3, 4].

To support multiple writers in our coherence scheme—thus allowing multiple speculatively modified copies of a single cache line to exist—we need the following two new features. First, an *invalidation-speculative* will only cause a violation if it is from a logically-earlier epoch and the line is speculatively loaded; this allows multiple speculatively modified copies of the same cache line to co-exist. Second, we must differentiate between normal invalidations (triggered by remote stores) and invalidations used only to enforce the inclusion property (triggered by replacements deeper in the cache hierarchy). A normal invalidation will not invalidate a speculative cache line that is only speculatively modified; hence the *homefree* epoch can commit a speculatively modified cache line to memory without invalidating logically-later epochs that have speculatively modified the same cache line.

3. Implementing Our Scheme

We now describe a potential implementation of our coherence scheme. We begin 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 support speculation in a shared cache.

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 logical ordering. (Signed differences preserve the relative ordering when the sequence numbers wrap around.)

The second issue is that we would like this comparison of epoch numbers to be performed quickly. At the same time, we would like to have the flexibility to have large epoch numbers (e.g., 32 or even 64 bits), since this simplifies TLS code generation when there is aggressive control speculation [19]. Rather than frequently computing the signed differences between large sequence numbers, we instead *precompute* the relative ordering between the current epoch and other currently-active epochs, and use the resulting *logically-later mask* to perform simple bit-level comparisons (as discussed later in Section 3.4).

The third issue is storage overhead. Rather than storing large epoch numbers in each cache line tag, we instead exploit the *logically-later mask* to store epoch numbers just once per chip.

3.2 Implementation of Speculative State

We encode the speculative cache line states given in Figure 4(a) using five bits as shown in Figure 5(a). Three bits are used to encode basic coherence state: *exclusive* (*Ex*), *dirty* (*Di*), and *valid* (*Va*). Two bits—*speculatively loaded* (*SL*) and *speculatively modified* (*SM*)—differentiate speculative from non-speculative states. Figure 5(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* & !*SM*).

Figure 5(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. 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 epoch number, an ownership required buffer (ORB), the addresses of the cancel and violation routines, and a violation flag which indicates whether a violation has occurred.[1]

3.3 Allowing Multiple Writers

As mentioned earlier in Section 2.7, it may be advantageous to allow multiple epochs to speculatively modify the same cache line. Supporting a multiple writer scheme requires the ability to merge partial modifications to a line with a previous copy of the line; this in turn requires the ability to identify any 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

[1] The cancel and violation routines are used to manage unwanted and violated epochs respectively. See [19] for more details.

(a) *Cache line state bits*

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

(b) *State encoding*

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

(c) *Hardware support*

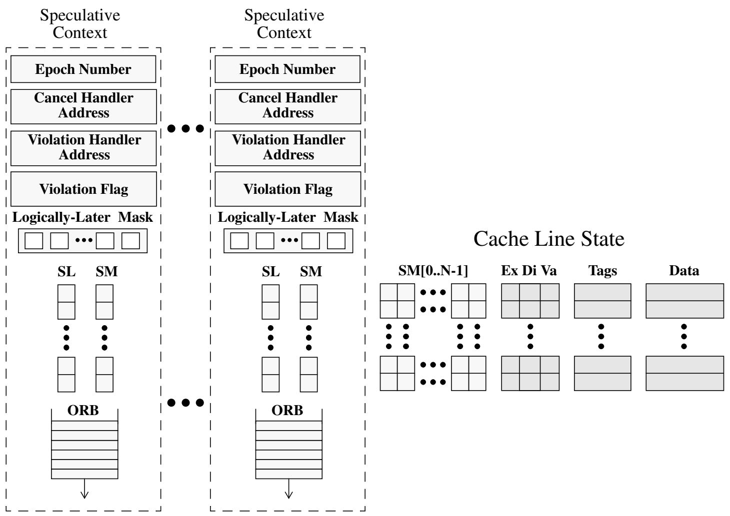


Figure 5. Encoding of cache line states.

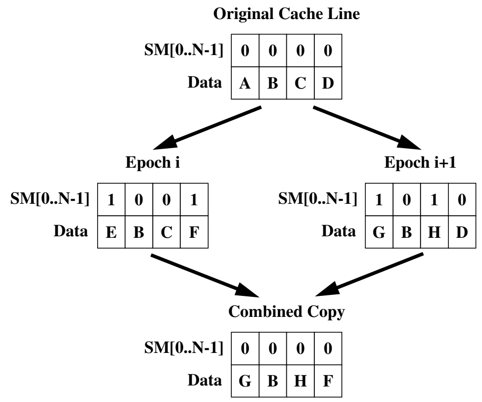


Figure 6. Support for combining cache lines.

a cache line, as shown in Figure 5(c). 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 lower granularity than the *SM* bits can resolve, we must conservatively set the *SL* bit for that cache line since we can no longer perform a combine operation on this cache line—setting the *SL* bit ensures that a violation is raised if a logically-earlier epoch writes the same cache line.

Figure 6 shows an example of how we combine a speculatively modified version 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* [1]). The *fine-grain SM* bits allow us to distinguish exposed loads, and therefore can help avoid false violations.

3.4 Support for Speculation in a Shared Cache

We would like to support multiple speculative contexts within a shared cache for three reasons. First, we want to maintain speculative state across OS-level context switches so that we can support TLS in a multiprogramming environment. Second, we can use multiple speculative contexts to allow a single processor to execute another epoch when the current one is suspended (i.e. during a suspending violation). Finally, multiple speculative contexts allow TLS to work with *simultaneous multithreading* (SMT) [23].

TLS in a shared cache allows epochs from the same program to access the same cache lines with two exceptions: (i) two epochs may not modify the same cache line, and (ii) an epoch may not read the modifications of a logically-later epoch. We can enforce these constraints either by suspending or violating the appropriate epochs, or else through cache line replication. With the latter approach, a speculatively modified line is replicated whenever another epoch attempts to speculatively modify that same line. This replicated copy is obtained from the external memory system, and both copies are kept in the same associative set of the shared cache. If we run out of associative entries, then replication fails and we must instead suspend or violate the logically-latest epoch owning a cache line in the associative set. Suspending an epoch in this case must be implemented carefully to avoid deadlock.

Figure 5(c) shows hardware support for shared-cache speculation where we implement several speculative contexts. The *Ex*, *Di*, and *Va* bits for each cache line are shared between all speculative contexts, but each speculative context has its own *SL* and *SM* bits. If *fine-grain SM* bits are implemented, then only one group of them is necessary per cache line (shared by all speculative contexts), since only one epoch may modify a given cache line. The single *SM* bit per speculative context indicates which speculative context owns the cache line, and is simply computed as the `OR` of all of the *fine-grain SM* bits.

To determine whether a speculative access requires replication, we must compare the epoch number and speculative state bits with other speculative contexts. Since epoch number comparisons may be slow, we want to use a bit mask which can compare against all speculative contexts in one quick operation. We maintain a *logically-later mask* for each speculative context (shown in Figure 5(c)) that indicates which speculative contexts contain epochs that are logically-later, thus allowing us to quickly make the comparisons using simple bit operations [19].

3.5 Preserving Correctness

In addition to data dependences, there are a few other issues related to preserving correctness under TLS. First, speculation must fail whenever any speculative state is lost (e.g., the replacement of a speculatively-accessed cache line, the overflow of the ORB, etc.). 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. We will explore this issue more aggressively in future work; for now, we simply stall a speculative thread if it attempts to perform a system call until it is *homefree*.

4. Experimental Framework

We evaluate our coherence protocol through detailed simulation. Our simulator models 4-way issue, out-of-order, superscalar processors similar to the MIPS R10000 [24]. 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 1. We simulate all applications to completion.

Our baseline architecture has four tightly-coupled, single-threaded processors, each with their own primary data and instruction caches. These are connected by a crossbar to a 4-bank, unified secondary cache. Our simulator implements the coherence scheme defined in Section 2 using the hardware support described in Section 3. To faithfully simulate the coherence traffic of our scheme, we model 8 bytes of overhead for coherence messages that contain epoch numbers. Because epoch numbers are compared lazily (and in parallel with cache accesses), they have no impact on memory access latency.

The simulated execution model makes several assumptions with respect to the management of epochs and speculative threads. Epochs are assigned to processors in a round-robin fashion, and each epoch must spawn the next epoch through the use of a lightweight fork instruction. For our baseline architecture, we assume that a fork takes 10 cycles, and this same delay applies to synchronizing two epochs when forwarding occurs. Violations are detected through polling, so an epoch runs to completion before checking if a violation has occurred. When an epoch suffers a violation, we also squash all logically-later epochs.

We are simulating real MIPS binaries which contain TLS instructions. Unused coprocessor instruction encodings are used for TLS primitives, and are added to the applications using `gcc ASM` statements. To produce this code, we are using a set of tools based on the SUIF compiler system. These tools, which are not yet complete, help analyze the dependence patterns in the code, insert TLS primitives into loops, perform loop unrolling, and insert synchronization code. The choice of loops to parallelize and other optimizations (described below) were made by hand, although we plan to have a fully-automatic compiler soon. We only parallelize regions of code that are not provably parallel (by a compiler).

Table 1. Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder Buffer Size	32
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 (16K, 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	8 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
Intra-Chip Communication Latency	10 cycles
Inter-Chip Communication Latency	200 cycles

Table 2 shows the applications used in this study: `buk` is an implementation of the bucket sort algorithm; `compress95` performs data compression and decompression; `equake` uses sparse matrix computation to simulate an earthquake; and `ijpeg` performs various algorithms on images. The `buk` application has been reduced to its kernel, removing the data set generation and verification code—the other applications are run in their entirety. For `compress95`, certain loop-carried dependences occur frequently enough that we either hoist them outside of the loop or else explicitly forward them using wait-signal synchronization.

5. Experimental Results

We now present the results of our simulation studies. To quantify the effectiveness of our support for TLS, we explore the impact of various aspects of our design on the performance of the four applications. Our initial sets of experiments are for a single-chip multiprocessor, and later (in Section 5.5) we evaluate larger-scale machines that cross chip boundaries.

5.1 Performance of the Baseline Scheme

Table 3 summarizes the performance of each application on our baseline architecture, which is a four-processor single-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

Table 2. Applications and their speculatively parallelized regions.

Suite	Application	Input Data Set	Speculative Region (*src_file:line, loop type*)	Unrolling Factor	Avg. Insts. per Epoch	Parallel Coverage
NAS-Parallel	`buk`	4MB	buk.f:111, do loop	8	81.0	22.8%
			buk.f:123, do loop	8	135.0	33.8%
Spec95	`compress95`	test [test.in]	compress.c:480, while loop	1	196.7	24.6%
			compress.c:706, while loop	1	240.4	22.7%
	`ijpeg`	test	jccolor.c:138, for loop	32	1467.9	8.2%
		[specmun.ppm]	jcdctmgr.c:214, for loop	1	80.8	2.2%
		quality 10	jidctint.c:171, for loop	1	84.0	5.0%
		smoothing_factor 10	jidctint.c:276, for loop	1	100.3	6.7%
Spec2000	`equake`	test [inp.in]	quake.c:1195, for loop	1	2925.5	39.3%

Table 3. Performance impact of TLS on our baseline architecture (a four-processor single-chip multiprocessor).

Application	Overall Region Speedup	Parallel Coverage	Program Speedup
`buk`	2.26	56.6%	1.46
`compress95`	1.27	47.3%	1.12
`equake`	1.77	39.3%	1.21
`ijpeg`	1.94	22.1%	1.08

self-relative speedups. As we see in Table 3, we achieve speedups on the regions of code that we parallelized ranging from 27% to 126%. The overall program speedups are limited by the *coverage* (i.e. the fraction of the original execution time that was parallelized), and they range from 8% to 46%. To simplify our discussion, we will focus only on the speculatively parallelized regions of code throughout the remainder of this section.

Figure 7 shows how performance varies across a number of different processors from two different perspectives. Figure 7(a) shows *execution time* normalized to the original (i.e. non-TLS) sequential execution. (Note that the *1 processor* bars in Figure 7 are this original executable, rather than the TLS executable running on a single processor.) Figure 7(b) shows *aggregate cycles*, which is simply the normalized number of cycles multiplied by the number of processors. Ideally, the *aggregate cycles* would remain at 100% if we achieved linear speedup; in reality, it increases as the processors become less efficient.

The bars in Figure 7 are broken down into seven segments explaining what happened during all potential graduation slots.[2] The top 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 location (*sync*); and waiting to become homefree (*homefree*). The remaining segments represent regular execution: the *busy* segment is the number of slots where instructions graduate; the *dcache_miss* 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. Finally, the *idle* segment represents slots where a processor has nothing to execute. It is somewhat easier to directly compare these categories in Figure 7(b), where an increase in the size of a segment means that a problem is getting worse. Also note that time wasted on failed speculation can contribute to any one of these segments.

[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.

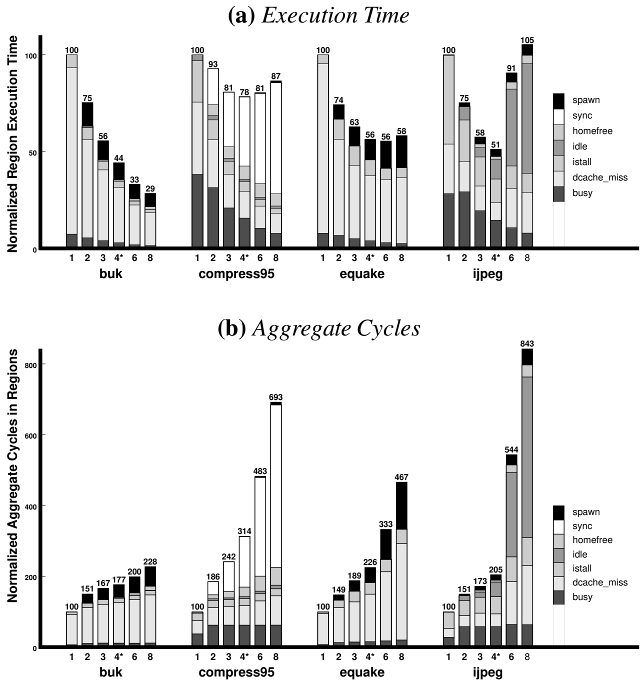


Figure 7. Performance of our TLS scheme on a single-chip multiprocessor. Part (a) shows normalized execution time, and part (b) is scaled to the number of processors multiplied by the number of cycles. The number of processors in our baseline architecture is four, as indicated by the *.

Looking at the single-processor results, we see that `buk` and `equake` are limited by memory performance (since they have large *dcache_miss* segments), while the other applications are more computationally intensive and have relatively large *busy* and *istall* segments. As we increase the number of processors and begin to speculatively execute in parallel, we achieve speedup in all cases. All applications with the exception of `compress95` experience an increase in time spent waiting for the lightweight fork (*spawn*), since we fork epochs in sequential order. For `compress95`, this overhead is hidden by synchronization (*synch*) for forwarded values, which increases quickly with the number of processors.

As we see in Figure 7(a), `buk` continues to enjoy speedups up through eight processors. For the other three cases, however, performance levels off and starts to degrade prior to eight processors.

Table 4. TLS overhead statistics for our baseline architecture (a four-processor single-chip multiprocessor).

Application	Dynamic Instr. Overhead	Misses to Other Caches	ORB Statistics: Avg. Flush Latency (cycles)	ORB Statistics: Size (entries) Avg.	ORB Statistics: Size (entries) Max.
`buk`	5.3%	34.47%	13.95	2.38	9
`compress95`	30.6%	3.02%	0.04	0.01	8
`equake`	3.7%	1.67%	0.13	0.04	12
`ijpeg`	7.0%	65.00%	1.06	0.17	5

The most dramatic case is `ijpeg`, where performance degrades sharply beyond four processors for the following reasons: (i) some of the speculative regions in `ijpeg` contain only four epochs (after loop unrolling); and (ii) an unfortunate mapping conflict in the cache causes many violations due to replacements. In general, the more epochs one attempts to execute in parallel, the greater the likelihood of dependence violations. Fortunately, there are applications which scale well to eight processors and beyond using TLS, as we will see later in Section 5.5.

5.1.1 Overheads of Thread-Level Speculation

We now investigate the overheads of our baseline scheme in greater detail using the statistics in Table 4. The first column in this table shows the TLS instruction overhead as a percentage of the original dynamic instructions. This instruction overhead is significant for `compress95` (over 30%) due to the large amount of data forwarding and the relatively small size of each epoch. The instruction overheads are much smaller (7% or less) for the remaining applications.

A second potential source of overhead with TLS is decreased cache locality due to data being distributed across multiple processors. The second column in Table 4 shows the percentage of cache misses with TLS on four processors where the data was found in another processor's cache. This rough indication of cache locality suggests that in two cases (`buk` and `ijpeg`), there may be significant room for improvement through more intelligent data placement and thread scheduling.

The ORB presents a third potential source of overhead. 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 hope that the average number of ORB entries per epoch remains small. As we see in Table 4, the average number of ORB entries is in fact small: less than 2.5 for `buk`, and less than 0.2 for the other three cases. This translates into an average ORB flush latency of roughly fourteen cycles for `buk`, and roughly one cycle or less for the other cases. Despite `buk`'s fourteen cycle ORB flush latency, it still speeds up quite well. To further mitigate the impact of this latency on performance, we could design the hardware to begin flushing the ORB as soon as the *homefree token* arrives (in our experiments, we take the less aggressive approach of also waiting until the epoch finishes before flushing the ORB). The rightmost column in Table 4 shows that a twelve-entry ORB is sufficient to eliminate the possibility of ORB overflow for these applications.

Finally, Figure 8 shows a breakdown of the causes of violations, which vary across the applications. Below each bar, we show the ratio of the number of violations to the number of epochs committed. (Note that this ratio can be greater than one, since an epoch can suffer multiple violations prior to committing.) The violations are broken down into the following three categories: (i) those due to the replacement of speculatively-accessed lines from the cache; (ii) those due to normal invalidations, which correspond to *logically-earlier* epochs flushing the given address from their ORB at commit time; and (iii) those due to *speculative* invalidations, which correspond to another epoch speculatively modifying a line that the given epoch had speculatively loaded earlier. As we see in Figure 8, `compress95` does not suffer any violations (in part due to the use of explicit data forwarding), and the few violations that occur in `ijpeg` are due to cache replacements. Violations occur far more frequently in `buk` and `equake`, where they are caused primarily by either normal or speculative invalidations. Given the choice, speculative invalidations are preferable over normal ones because they help reduce then size of the ORB and give earlier notification of violations.

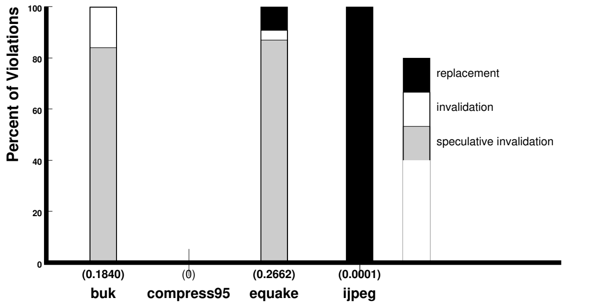


Figure 8. Breakdown of causes of violations on our four-processor baseline architecture. The ratio of violations to epochs committed is shown below each bar.

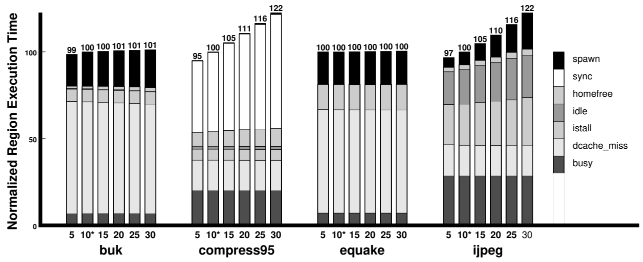


Figure 9. Impact of varying communication latency (in cycles). The baseline architecture has a communication latency of 10 cycles, as indicated by the *.

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

5.2 Impact of Communication Latency

Figure 9 shows the impact of varying the communication latency within the single-chip multiprocessor from five to thirty cycles (in the baseline architecture, it is ten cycles). As we see in Figure 9, `buk` and `equake` are insensitive to communication la-

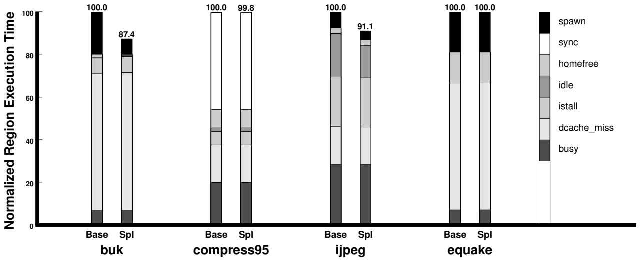


Figure 10. Benefit of allowing speculative invalidations to invalidate non-speculative cache lines (*SpI*) vs our baseline coherence scheme (*Base*).

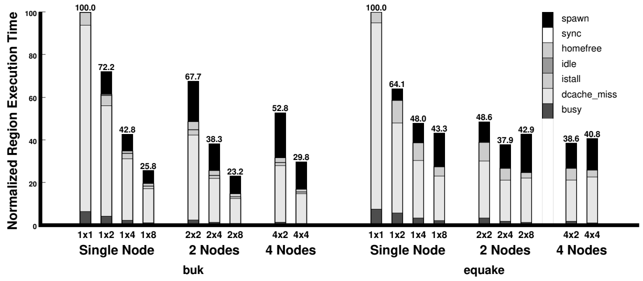


Figure 11. Region performance of `buk` and `equake` on a variety of multiprocessor architectures (*NxM* means N nodes of M processors).

tencies within this range because their performance is mostly limited by data cache capacity misses rather than inter-epoch communication. `Compress95` and `ijpeg` are more latency-sensitive: they suffer from increased synchronization and thread spawning times, respectively. Given the region and program speedups in Table 3, we observe that all of these applications would still enjoy speedups with higher communication latencies than the ten cycles assumed in our baseline architecture.

5.3 Support for Multiple Writers

As discussed earlier in Sections 2.7 and 3.3, a potential enhancement of our baseline coherence scheme is to allow for multiple writers to the same cache line. We simulated such a multiple-writer scheme, and found that it offered no performance benefit for any of our four applications. While this is hardly sufficient evidence to claim a negative result, we can offer the following insights into why our applications did not require multiple writer support (i.e. write-after-write (WAW) dependence violations rarely occurred). In `buk` and `equake`, we see fairly random access patterns for many of the stores; these cases are not a problem since the likelihood of successive epochs storing to the same cache line is low. The case that we did see that can be pathologically bad (e.g., in `ijpeg`) is when each loop iteration stores the next sequential element in an array. Fortunately this case is easy to identify and fix: we simply unroll (or strip-mine) the loop body such that each epoch gets a block of iterations that perform all of the sequential stores to a given cache line. In other words, we use *block-cyclic* rather than *cyclic* (aka "round-robin") scheduling, which is a common technique for avoiding the analogous problem of *false sharing* in traditional shared-memory multiprocessors. Since loop unrolling (or strip-mining) is also attractive in TLS for the sake of creating larger epochs to help reduce the relative communication overhead, this may be a valuable technique for machines that support TLS but not multiple writers.

5.4 Speculative Invalidation of Non-Speculative Cache Lines

As discussed earlier in Section 2.6.1, one design choice is whether a speculative invalidation can invalidate a cache line in a non-speculative state. Recall that our baseline scheme did not allow this, with the goal of not impeding the progress of a *homefree* epoch. However, as we see in Figure 10, both `buk` and `ijpeg` achieve significantly better performance if we *do* allow speculative invalidations of non-speculative lines since this reduces the average number of ORB entries, and hence the latency of flushing the ORB and passing the *homefree token*. These additional speedups of roughly 10% within the parallelized regions of `buk` and `ijpeg` translate into overall program speedups of 53% (vs. 46%) and 10% (vs. 8%), respectively. Hence allowing speculative invalidations to invalidate non-speculative lines is clearly a worthwhile enhancement to our baseline scheme.

5.5 Scaling Beyond Chip Boundaries

Having demonstrated the effectiveness of our TLS scheme on single-chip multiprocessors, we now evaluate how well it scales to larger-scale, *multi-chip* multiprocessors, where each node in the system is itself a single-chip multiprocessor. Figure 11 shows the performance of `buk` and `equake` across a range of these multi-node architectures. Starting with single-node performance, notice that both of these applications speed up well within a single chip, although `equake` shows diminishing returns with eight processors. (Note that these results differ slightly from those given earlier in Section 5.1, since we are now simulating an extra level of interconnection in the memory hierarchy.)

Now consider the multi-node architectures, where the communication latency between nodes is twenty times larger than that within a node (i.e. 200 vs. 10 cycles). As we see in Figure 11, both `buk` and `equake` speed up well on many of these 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); this is the reason why both the *2x4* and *4x2* configurations are faster than the *1x8* configuration for `equake`. On the other hand, an obvious disadvantage is that having more nodes increases the average cost of inter-processor communication; for this reason, the *2x4* and *4x2* configurations are both *slower* than the *1x8* configuration for `buk`.

Overall, we observe that the best performance for each application was achieved on a multi-node architecture: *2x8* for `buk`, and *2x4* for `equake`. These region speedups of 331% and 164% for `buk` and `equake`, respectively, translate into program speedups of 75% and 39%. These results demonstrate that our mechanisms for flushing the ORB and passing the homefree token are scalable, and do not limit the scope of our TLS scheme.

6. Conclusions

We have presented a cache coherence scheme that supports thread-level speculation on a wide range of different parallel architectures, from single-chip multiprocessors or simultaneously multithreaded processors up to large-scale machines which might use single-chip multiprocessors as their building blocks. Our experimental results demonstrate that our baseline TLS scheme offers absolute program speedups ranging from 8% to 46% on a four-processor single-chip multiprocessor, and that two of the applications we studied achieve even larger speedups (up to 75%) on multi-chip architectures. We observe that the overheads of our scheme are reasonably small—in particular, the ORB mechanism used to commit speculative modifications at the end of an epoch is not a performance bottleneck, and only a relatively small ORB (e.g., twelve entries) is necessary.

We make two observations regarding the applications we studied. First, we notice that some applications are sensitive to communication latency and are likely to perform well only in a tightly-coupled environment (e.g., `compress` and `ijpeg`), while others are also suitable for larger-scale multiprocessors with longer communication latencies (e.g., `buk` and `equake`). Second, we observe that the applications benefit from TLS *without* special support for multiple speculative writers, in part because we can use loop unrolling to avoid problems with false dependence violations.

Our scheme does not require a large amount of new hardware; in fact, we are currently implementing a purely software-based version of our scheme within a software DSM system. As parallel architectures become increasingly commonplace in the future on a wide variety of scales, we expect that thread-level speculation will become an increasingly important technique for helping compilers automatically create parallel programs to exploit all of this processing potential.

7. Acknowledgments

This research is supported by a grant from NASA. Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship.

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

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

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

[11] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter Usenix Conference*, January 1994.

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

[13] 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.

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

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

[16] 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.

[17] 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.

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

[19] J. G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[20] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallellization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

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

[22] 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.

[23] 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.

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

[25] 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.