Extending Cache Coherence to Support Thread-Level Data Speculation on a Single Chip and Beyond

J. Gregory Steffan Christopher B. Colohan
Todd C. Mowry

December 1998
CMU-CS-98-171

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

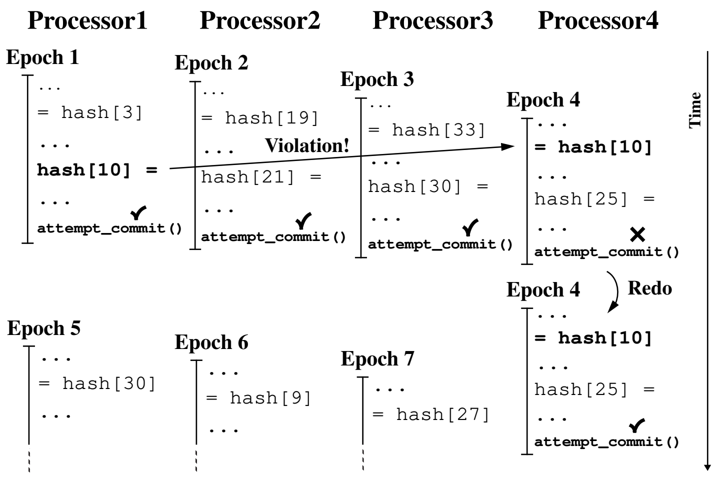
Thread-Level Data Speculation (TLDS) is a technique which enables the optimistic parallelization of applications despite ambiguous data dependences between the resulting threads. Although TLDS is mostly managed by software, hardware provides two key pieces of functionality: (i) detecting dependence violations, and (ii) buffering speculative side-effects until they can be safely committed to memory. To provide this functionality we present an extension to invalidation-based cache coherence which is both scalable and has a minimal impact on hardware complexity. We explore the design space in depth and find that our baseline architecture is sufficient to exploit speculative parallelism.

This research is supported by a grant from NASA. Todd C. Mowry is partially supported by a Faculty Development Award from IBM.

Keywords: C.1.2 Multiple Data Stream Architectures (Parallel Processors), B.3.2 Cache Memories, C.4 Performance of Systems

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

(a) Pseudo-Code



(b) TLDS Execution

Figure 1: Example of TLDS execution.

1 Introduction

As the number of transistors on a state-of-the-art microprocessor chip continues to increase, architects are exploring new ways to translate these additional resources into improved performance beyond today's superscalar paradigm. One option is to integrate *multiple* processors onto a single chip. While single-chip multiprocessing is attractive from an implementation point of view [9], it is only useful for accelerating a given application if that application contains parallel threads. To avoid placing an additional burden on programmers, the most desirable method of parallelizing an application would be for the compiler to create the parallel threads automatically. Unfortunately, despite the significant progress which has been made in automatically parallelizing numeric applications, compilers have had little or no success in automatically parallelizing the ubiquitous *non-numeric* applications due to their complex control flow and memory access patterns.

1.1 Thread-Level Data Speculation

Thread-**L**evel **D**ata **S**peculation (**TLDS**) [15] and other similar techniques [10, 13] allow 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 TLDS 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 rewinding and re-executing 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*, so *epoch 4* is rewound and restarted to produce the correct result. This example demonstrates the basic principles of TLDS—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 is a single loop iteration here. *Epoch numbers* are used to specify and maintain the ordering of memory references, and their order corresponds to the original sequential execution of the program. Any *violation* of the original program order is detected with the assistance of the TLDS mechanism. To help the reader, a glossary is provided in Figure 2 to summarize the terminology we use.

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 can also indicate that certain parallel threads are unordered.

Homefree Token: A token which indicates that an epoch is the least speculative, and therefore cannot suffer a violation. If the epoch has not already suffered a violation when it receives the homefree token, it may commit its speculative modifications to memory.

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

Speculation Level: The level in the cache hierarchy below which speculative data references are handled by coherence.

Speculative Context: The state information associated with the execution of an epoch. This includes the register contents and cache state.

Speculative Region: Part of a program where TLDS is performed.

Speculative Thread: A light-weight thread that is used to exploit 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 sequentially earlier epoch.

Figure 2: Glossary of terms for TLDS.

This paper focuses on the design of a cache coherence protocol. Further examples of the use of TLDS, and an exploration of the interface between TLDS hardware and software can be found in a previous technical report [14].

1.2 Related Work

The Multiscalar architecture [5, 13] was the first to provide hardware support for thread-level speculation. While the Multiscalar approach is effective at exploiting speculative parallelism, it originally had two important disadvantages: the ring architecture is over-specialized; and memory disambiguation is performed by a large, centralized structure (the ARB). SVC [6] is a cache coherence scheme which overcomes the centralization of the ARB, but it uses a snoopy bus based protocol which does not scale beyond a tightly coupled system.

There have since been several other proposals [6, 7, 15, 18] describing distributed approaches, the most relevant being the Stanford Hydra [7]. Similar to the TLDS approach, Hydra adds memory disambiguation support to a general-purpose, chip multiprocessor (CMP), and uses software to manage threads and speculation. However, there are two important distinctions between Hydra and TLDS. First, each processor in a Hydra CMP has a special write-buffer for speculative modifications, while our implementation of TLDS uses the first-level caches to buffer speculative state. Second, to ensure that data dependences are preserved, Hydra employs write-through coherence and snoops the write buffers on all stores, while we use write-back invalidation-based coherence. The previous schemes do not scale beyond a tightly coupled system such as single-chip or snoopy bus based multiprocessor. We believe that irregular scientific applications will be fertile ground where TLDS can offer benefits, since large data sets will make true dependences between threads sparse. Demonstrating the power of TLDS with such applications will be future work, and is beyond the scope of this paper.

1.3 Contributions

The protocol we propose is the first truly scalable solution for TLDS. Since this coherence scheme is an extension of invalidation-based coherence, it will work wherever invalidation-based coherence can be used. This means that it can be used in a wide variety of multiprocessor architectures including multithreaded processors, chip-multiprocessors, more traditional multiprocessors, and even software distributed shared memory multiprocessors. Implementing our scheme has a minimal impact on the area and complexity of the augmented multiprocessor. This paper presents a detailed description of how our coherence scheme can be realized, and is the first to evaluate this type of scheme using a collection of R10000 class processors.

The remainder of this paper is organized as follows: Section 2 describes how an invalidation-based cache coherence scheme can be extended to detect data dependence violations, and Section 3 gives a possible

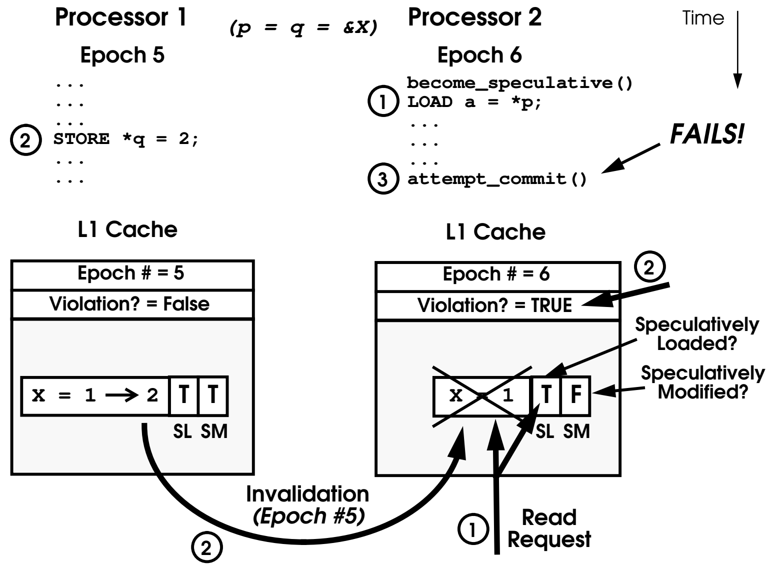


Figure 3: Using cache coherence to detect a RAW dependence violation.

hardware implementation of that scheme. We evaluate the performance of our implementation in Section 4, and conclude in Section 5.

2 Coherence Scheme

To support TLDS, we must perform the difficult task of detecting data dependence violations at run-time. Since we perform memory operations in parallel which may not be independent, the addresses of all memory references generated by separate speculative threads must be compared to ensure that data dependences are preserved. A key element of data speculation is detecting when a data dependence has been violated. This detection is straightforward for instruction-level data speculation, since there are few load and store addresses to compare. However, for thread-level data speculation 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.

One solution is to use an invalidation-based cache coherence scheme. In invalidation-based coherence, a cache line must be owned exclusively before it may be modified. Exclusive access is achieved by sending out invalidations to all other caches owning a copy of the line. To extend this mechanism to detect data dependence violations, we simply need to track which cache lines have been *speculatively* loaded. Whenever a less speculative epoch modifies a cache line that we have speculatively loaded (as indicated by an arriving invalidation), we know that a violation has occurred.

2.1 Example

To demonstrate how invalidation-based cache coherence can be extended to track data dependences, we give an example of the detection of 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, we augment the state of each cache line to indicate whether the cache line has been speculatively loaded and/or speculatively modified. For each cache, we also maintain a number which indicates the sequential ordering of that epoch with respect to all other epochs (the *epoch number*), 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

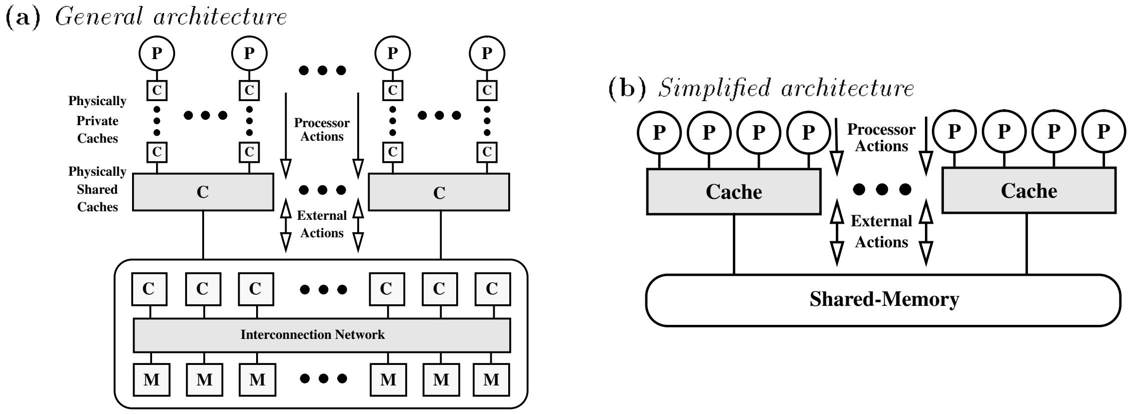


Figure 4: Base architecture for the TLDS coherence scheme.

violation. First, the same cache line that the invalidation is for must be present in the cache. Second, it must be marked as speculatively loaded. Third, comparison of the current epoch number with the epoch number of the invalidation must tell us that the invalidation came from a less speculative epoch. Since all three conditions are true, 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. We want the coherence mechanism to be applicable to any combination of single-threaded or multithreaded processors and shared-memory architectures, not necessarily restricted to multiprocessors on a single chip.

For simplicity, we assume some shared-memory architecture that supports an invalidation-based cache coherence scheme where all hierarchies enforce inclusion. Figure 4(a) shows a generalization of the underlying architecture. There may be a number of processors or perhaps only a single multithreaded processor, followed by an arbitrary number of levels of physically private caching. The level of interest is the first level where invalidation-based cache coherence begins, which we will call the *speculation level*.

We generalize levels of the system below the speculation level, as shown in Figure 4(a), to be an interconnection network providing access to main memory with some arbitrary number of levels of caching. All memory references originating from a processor that reach the speculation level will be referred to as *processor actions*, and all coherence events that are received from lower levels of the system (i.e. further away from the processor) will be referred to as *external actions*.

The amount of detail shown in Figure 4(a) is not necessary for the purposes of describing our cache coherence scheme. Instead, Figure 4(b) shows a simplified model of the underlying architecture. The speculation level described above happens to be a physically shared cache and is simply referred to from now on as "the cache". Above the caches, we have some number of processors, and below the caches we have an implementation of cache-coherent shared memory.

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

Table 1: Shared cache line states

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

Table 2: Processor-initiated actions

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

2.3 Line State in the Cache

A standard invalidation-based cache 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 external memory. When a processor attempts to write to a cache line, exclusive access must first be obtained—if the line is not already in the exclusive state, 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 to include the seven new states as described in Table 1. The new states denote four orthogonal properties of a cache line: whether it is dirty; whether it has been speculatively loaded (SpL); whether it has been speculatively modified (SpM); and whether it is exclusive (E) versus shared (S).

Although these properties are orthogonal, some combinations are not allowable such as dirty and speculatively modified. When a cache line is dirty, the cache owns the only up-to-date copy of the cache line, and must preserve it without speculative modifications so that the line can eventually be supplied to the rest of the memory system. Conversely, when a cache line is in the speculatively modified state, we may need to discard any speculative modifications to the line if the speculation ultimately fails. Since it would be difficult to isolate both dirty and speculatively modified portions of the same line in a traditional hardware cache (especially if these portions can overlap), it is difficult to allow both of these states to co-exist.

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

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

For speculation to succeed, any cache line with a speculative state must remain in the cache 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 not correctness.

2.4 Processor Actions

We now describe an implementation of an invalidation-based cache coherence scheme extended to detect data dependence violations. First, we list all possible actions that are originated by the processors, the shared

Table 3: Coherence Actions and Conditions

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

cache controller, or the external memory system. Table 2 lists the possible actions which are originated by the processor. Processor-initiated actions are divided into reads and writes, hits and misses, and speculative and non-speculative accesses.

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

- Two different epochs may both speculatively read the same cache line.
- If an epoch speculatively modifies a cache line, only a more speculative epoch may read that cache line afterwards. This effectively allows us to forward speculative modifications between two properly ordered epochs, and is only guaranteed to be correct for epochs executing on processors that share a cache. If a less speculative epoch attempts to read the cache line, a read conflict-miss will result.
- Only one epoch may speculatively modify a given cache line. If an epoch attempts to speculatively modify a cache line that has already been speculatively modified by a different epoch, a write conflict miss (*PWCMSp*) results.

2.5 Cache Actions

Table 3 describes all coherence actions and conditions. The first five actions (*G.ER*, *G.EREx*, *G.EWb*, *G.EU*, and *G.EU*) all behave as in a standard coherence scheme, and the next two are new speculative actions to support TLDS. These speculative actions both piggyback the epoch number of the requester. *G.EUpSp* sends speculative invalidations to all appropriate processors, and *G.ERExSp* requests a copy of a cache line. For both speculative actions, exclusiveness may not be granted if the requesting epoch is more speculative than the receiving epoch, as determined by comparison of the epoch numbers.

If two epochs speculatively modify the same cache line, there are two ways to resolve the situation. One option is to simply squash the more speculative epoch. Alternatively, we could allow both epochs to modify their own copies of the cache line and combine them with the real copy of the cache line as they commit, as is done in a multiple-writer coherence protocol [2, 3]. The action *G.Combine* indicates that the current cache line should be combined with the copy stored at the next level of caching in the external memory system. If combining is not supported, the *G.Combine* action is simply ignored.

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

When an epoch becomes homefree, it generates an upgrade request for each entry in the ORB, as described by the *G.FlushORB* action. If cache line combining is supported, *G.FlushORB* may instead generate speculative upgrade requests for each line address in the ORB. Since write-after-write (WAW) dependences are not true dependences, they may be eliminated through renaming. The more speculative of two epochs which both speculatively modify the same cache line does not need to be squashed, and the speculative modifications may be combined later. A speculatively modified cache line may not change to the dirty state until *G.FlushORB* completes.

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

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

Support may exist to detect whether a load is an *upwards exposed use*—i.e. the use of a location without a prior definition of that location by the same epoch [1]. We only have to consider a location to be speculatively loaded if it is an upwards exposed use, otherwise the load cannot cause a data dependence violation. To differentiate these cases, we will check the condition *Exposed*.

Some speculative actions received from the external memory system will have different effects depending on whether they were originated by a less speculative epoch or a more speculative epoch. The *Older* condition is true if the epoch which generated the action is older than the current epoch and false otherwise.

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

2.6 Other Actions

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

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

Table 4: Other actions

Action	**Description**
ER	External read.
EREx	External read exclusive (copy of line is supplied with ack).
EI	External invalidate.
EISp	External invalidate speculative. Only invalidate the line if this from a less speculative epoch.
ERExSp	External read exclusive speculative (copy of line is supplied with ack).
HFree	Epoch has become homefree.
Viol	Epoch has suffered a violation or been cancelled.
$\rightarrow$X	Transition to new state X.
(A)?(B):(C)	If A then B else C.

2.7 State Transition Diagram

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

We now investigate several "*action* $\times$ *state*" pairs of interest.

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

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

- This version of the coherence scheme is implemented with the objective of slowing down a non-speculative thread as little as possible. For this reason, a cache line in a non-speculative state is not invalidated when a speculative upgrade-request occurs, as shown by *EISp* $\times$ *E*. Alternatively, the cache line could be relinquished in order to give exclusiveness to the speculative thread, possibly eliminating the need for that thread to obtain ownership when it becomes homefree. These two options must be analyzed experimentally in future work to decide which approach results in better performance.

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

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

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

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

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

Action	Cache Line State: I	Cache Line State: E	Cache Line State: S
PRM	G.ER; (Ack=Excl)?(→E):(→S);	G.ER; (Ack=Excl)?(→E):(→S);	G.ER; (Ack=Excl)?(→E):(→S);
PRH	-	→E;	→S;
PWM	G.EREx; →D;	G.EREx; →D;	G.EREx; →D;
PWH	-	→D;	G.EUp; →D;
PRMSp	G.ER; (Ack=Excl)?(→SpLE):(→SpLS);	G.ER; (Ack=Excl)?(→SpLE):(→SpLS);	G.ER; (Ack=Excl)?(→SpLE):(→SpLS);
PRCMSp	-	-	-
PRHSp	-	→SpLE;	→SpLS;
PWMSp	G.ERExSp; (Ack=Excl)?(→SpME):(G.ORB; →SpMS);	G.ERExSp; (Ack=Excl)?(→SpME):(G.ORB; →SpMS);	G.ERExSp; (Ack=Excl)?(→SpME):(G.ORB; →SpMS);
PWCMSp	-	-	-
PWHSp	-	→SpME;	G.EUpSp; (Ack=Excl)?(→SpME):(G.ORB; →SpMS);
ER	-	→S;	→S;
EREx	-	→I;	→I;
EI	-	→I;	→I;
EISp	-	→S;	→S;
EUp	-	→I;	→I;
ERExSp	-	→S;	→S;
HFree	→I; G.FlushORB;	→E; G.FlushORB;	→S; G.FlushORB;
Viol	→I;	→E;	→S;

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

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

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

2.8 Coherence in the External Memory System

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

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

2.9 Forwarding Data Between Epochs

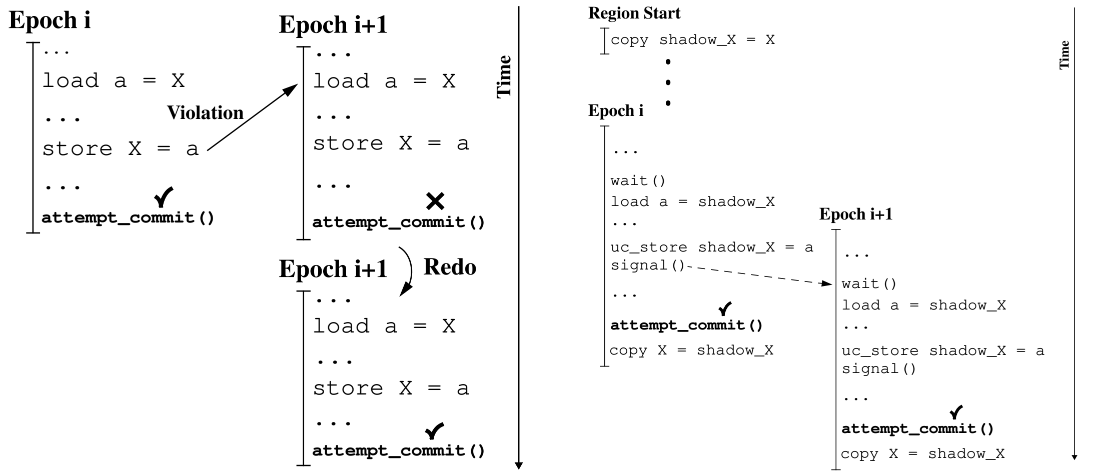
A key requirement of TLDS is the ability to forward data between epochs. For register values, forwarding is required for correctness and must be performed. For memory locations, such as scalars, forwarding may be performed to avoid frequent data dependence violations but is not required for correctness. There are three cases where forwarding is used. In the first case, initial information is forwarded from a thread to its child upon creation, including the initial register values as well as any other data needed to start the new epoch. This type of forwarding could occur through shared memory or possibly by some other faster means of communication.

The second and third cases both involve forwarding values at some point after the initialization of the child epoch. The second case involves forwarding locations that do not have ambiguous data dependences, such as registers or scalars which have provably not had their address taken. The third case is forwarding in the midst of ambiguous memory references. Both cases require the ability to issue a non-speculative store in the midst of speculative memory references, possibly implemented as an uncached store. Producer/consumer style synchronization is also required, and could be implemented by synchronizing on specially allocated memory locations, or by implementing something with similar functionality to full/empty bits [4, 8, 11].

As shown in Figure 5(a), an inefficient way to forward a memory location from one epoch to another is simply to allow a data dependence violation to occur—the epoch which consumes the value is re-executed once the producing epoch has committed its speculative modifications. This causes some code to be re-executed unnecessarily.

Figure 5(b) shows how forwarding works at a high level for the second case, when the location to be forwarded has no ambiguous data dependences. First, a shadow copy of the location to be forwarded must be created—since the forwarded values are speculative, we do not want them to corrupt the correct value which is stored in the real location. The epoch then accesses the shadow location as though it were the original location, and synchronizes before the first use and and after the last definition of the location in question. Once an epoch is no longer speculative, the value in the shadow location is copied into the real location since this is now the true and current value. Should a violation occur due to some other data dependence, the shadow location must be restored with the most recent committed value. The epoch can then be re-executed.

The third case is difficult: forwarding a location that might have ambiguous data dependences. What makes this case difficult is the possibility that ambiguous loads and stores might occur between separate epochs or even within an epoch. The mechanisms required to successfully forward values in the midst of ambiguous memory references are beyond the scope of this study, and so in our evaluation we conservatively allow violations to occur, guaranteeing correctness.



(a) Data Dependence Violation

(b) Violation Avoided Through Forwarding

Figure 5: Simple forwarding, when the forwarded location has provably not had its address taken (uc_store represents an uncached store).

3 Implementation

In this section, we describe an implementation of our coherence scheme, starting 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 multiple epochs per processor.

3.1 Epoch Numbers

In previous sections we describe the use of epoch numbers in determining the relative order of two epochs. In the coherence scheme, an epoch number is associated with every speculative cache line and every speculative coherence action. The implementation of epoch numbers must address several issues. First, we must be able to differentiate between two epochs from independent programs or even from independent chains of speculation within the same program. We solve this problem by having each epoch number consist of two parts: a thread identifier (TID) and a sequence number. When the shared cache controller receives external speculative actions, it only applies them to speculative cache lines which have a matching TID. If the TID's match, then *Older* is computed by comparing the sequence number portion of the epoch numbers. If the TID's do not match, then *Older* is false.

The second issue concerns this comparison—we need the comparison of two epoch numbers to be fast. We also have the opposing desire to have large epoch numbers so that we may have many epochs. One solution is to have large integer epoch numbers (such as 32 bits or even larger) and then use signed differences to determine the relative order of the corresponding epochs. Signed differences have the benefit of preserving comparative order when the sequence numbers wrap around.

A third issue is storage. We do not want to store a 32 bit epoch number in the tag of every cache line. We will show that this is not necessary and that the epoch number may be stored in a single location for each epoch.

3.2 Cache Line State Encoding

We encode the speculative cache line states given in Table 1 using five bits as shown in Figure 6(a). Two bits, speculatively loaded (*SL*) and speculatively modified (*SM*), differentiate speculative states from non-

(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
SpLE	1	0	1	0	1
SpLS	1	0	0	0	1
SpME	0	1	1	1	1
SpMS	0	1	0	1	1
SpLME	1	1	1	1	1
SpLMS	1	1	0	1	1

(c) *Hardware support*

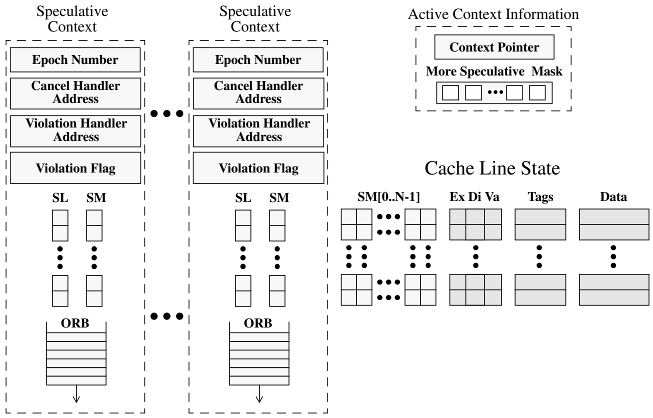


Figure 6: Encoding of L1 cache line states

speculative states. Figure 6(b) shows the state encoding which is designed to have the following two useful properties. First, when the *SM* and *SL* bits are reset, the state will change from a speculative state to the corresponding non-speculative state as required when an epoch becomes homefree. 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 *Va* bit to the `AND` of its *Va* bit with the inverse of its *SM* bit (i.e. $Va = Va \,\&\, !SM$).

3.3 Implementation of Speculative State

A naive implementation of the cache line state would be to place the speculative state bits and epoch number in each cache line. One problem with this approach is that there will be significant overhead associated with storing an epoch number with every cache line. We also want to avoid traversing the entire cache, for example when we invalidate all cache lines that have been speculatively modified.

Speculative state will be arranged as shown in Figure 6(c). We wire the *SL* bits as well as the *SM* bits so that the entire column of bits can be simultaneously reset using a single control signal. We also wire the *SM* bits to the corresponding *Va* bits so that all cache lines which have been speculatively modified may 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]

3.4 Support for Cache Line Combining

If an epoch has speculatively modified a cache line and another epoch sharing the same cache line on a different processor modifies its own copy, the coherence scheme will not signal a violation. However, when the less speculative epoch commits its speculative state, the more speculative epoch will receive an upgrade request and will therefore fail. Since these write-after-write (WAW) or output dependences are not *true* dependences, we would like speculation to succeed in their presence.

As discussed in Section 2.5, we need to provide support for multiple-writers. One possibility is to replicate the *SM* column of bits so that there are as many *SM* columns as there are words or even bytes in a cache line, as shown in Figure 6(c). We will call this *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 less speculative epoch writes the same cache line.

[1] The cancel and violation routines are used to manage unwanted and violated epochs respectively. See [14] for more details.

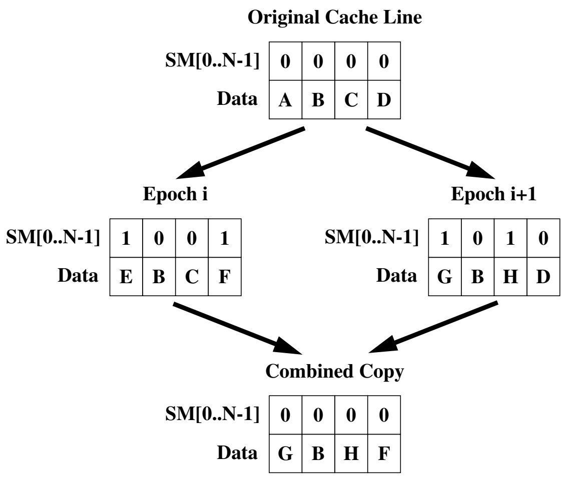


Figure 7: Support for cache line combining.

Figure 7 shows an example of cache line combining. Two epochs speculatively modify the same cache line simultaneously, setting the fine-grain *SM* bit for each location modified. Once the cache lines are committed, they are combined with the last committed cache line and the modifications of the most recent epoch are given precedence. In the example, both epochs have modified the first location. Since `epoch i+1` is more speculative, its value `G` takes precedence over the value `E`.

False violations can be a source of performance loss in TLDS. Since dependence violations are caused by speculative loads, or more specifically by cache lines that are in a speculatively-loaded state, then tracking the notion of speculatively loaded more precisely will result in fewer false violations. As described in Section 2.5, a cache line only needs to change to a speculatively loaded state (setting the *SL* bit) for speculative loads that are *exposed*. If fine-grain *SM* bits are implemented, a speculative load is considered exposed only if it has not yet been defined by the current epoch, which is indicated by the *SM* bit for that address not being set. This support should therefore reduce the number of false violations detected.

3.5 Support for Multiple Epochs per Processor

We would like to support multiple speculative contexts on a single processor for three reasons. First, we want to maintain speculative state across OS-level context switches so that we can support TLDS in a multiprogramming environment. Second, we can use multiple speculative contexts to allow the processor to execute another epoch when the current one is suspended (i.e. causes a suspending violation). Finally, multiple speculative contexts will allow TLDS to run under *simultaneous multithreading* (SMT) [16].

The coherence scheme as described in Section 2.7 supports multiple epochs per processor. Epochs from the same program may access the same cache lines, except in two cases: two epochs may not modify the same cache line, and an epoch may not read the modifications of a more speculative epoch. The coherence scheme avoids these cases either through the use of cache line replication, or else by suspending or violating the appropriate epoch.

Figure 6(c) shows hardware support for multiple epochs per processor 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 the fine-grain *SM* bits.

To determine whether a load or store results in a conflict miss requires comparing epoch numbers 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, so we maintain a *more speculative mask* indicating which speculative contexts contain epochs that are more speculative than the active epoch. Read and write conflict-misses can also be detected quickly using a

Table 6: Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Integer, 2 FP, 1 Memory, 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 Scheme	2-bit Counters

Memory Parameters	
Line Size	32B
Instruction Cache	32KB, 2-way set-assoc
Data Cache	32KB, 2-way set-assoc
Unified Secondary Cache	2MB, 2-way set-assoc
Data Cache Banks	2
Data Cache Fill Time (Requires Exclusive Access)	4 cycles
Miss Handlers	8 for data, 2 for insts
Main Memory Bandwidth	1 access per 20 cycles
Total Miss Latency to Secondary Cache	10 cycles
Total Miss Latency to Local Memory	75 cycles

similar mechanism [14]. To allow fast switching between speculative contexts, we will use an active context to indicate which speculative context is currently active.

Another feature of the coherence scheme which supports multiple epochs per processor is the ability to perform cache line replication. Replication is necessary to avoid suspension or violation whenever there is a conflict miss (see Section 2.4). For more details about how replication is implemented, see [14].

4 Performance Evaluation

We now quantify the performance of our scheme using detailed simulation. The goal of this performance study is to quantify how well our architecture supports TLDS, not to prove that TLDS can offer compelling performance benefits as shown previously [6, 10, 15]. For this reason, we concentrate on four applications with different behavior and explore in detail the effects of different aspects of our scheme.

4.1 Experimental Methodology

There are several steps involved in simulating the TLDS execution of an application. First, we compile the application with `-O2` optimization using the standard MIPS C compiler under IRIX 6.4. Second, we profile the application to find suitable regions for speculative execution—usually loops with a reasonable amount of computation per iteration that have infrequent loop-carried data dependences. Third, we annotate the binary to indicate the start and end of each epoch, and the points at which the epochs should synchronize (to allow data forwarding).

Our simulator reads traces generated by the MIPS `pixie` utility [12] and breaks the sequential trace into parallel traces according to our annotations. The simulator models out-of-order, superscalar processors with issues widths of four instructions similar to the MIPS R10000 [17]. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including contention) are all modeled, and are parameterized as shown in Table 6. Because our simulator is relatively fast, we are able to simulate all applications to completion.

Our baseline architecture has four tightly-coupled, single-threaded processors, each with their own primary caches, and sharing a single secondary cache among all processors. Our simulator implements the coherence scheme described earlier in Section 2, and modeling the hardware support described in Section 3. Although we do model communication latency and the limited bandwidth associated with flushing the ORB (as described later), we assume that the interconnection network has sufficient bandwidth such that coherence traffic is not otherwise delayed due to contention.

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 with a delay of the specified communication latency (10 cycles for the baseline architecture). This same delay applies to synchronizing two epochs when forwarding occurs, and to sending out upgrade requests when flushing the ORB. For the baseline architecture, we only allow one upgrade request to be issued per cycle. 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 epochs that are more speculative.

Table 7: Applications and Speculative Regions.

Suite	Application	Input Data Set	Speculative Region (src_file:line)	Number of Times Unrolled	Coverage (% sequential cycles)
NAS-Parallel	buk	64kB	buk.f:105, do loop	7	0.6%
			buk.f:117, do loop	3	31.8%
			buk.f:123, do loop	0	33.4%
Spec92	compress	test.in	compress.c:787, while loop	0	84.1%
	sc	loada1	interp.c:1001, for loop	0	62.5%
Spec95	ijpeg	specmun.ppm	jccolor.c:138, for loop	4	9.9%
			jidctint.c:171, for loop	1	6.0%
			jidctint.c:276, for loop	1	5.0%

Table 8: Application performance on baseline architecture.

Application	Average Number of Instructions per Epoch	Overall Region Speedup on Baseline Architecture	Coverage (% sequential cycles)	Program Speedup on Baseline Architecture
buk	50.0	1.59	65.8%	1.35
compress	74.9	1.05	84.1%	1.04
sc	417.0	2.90	62.5%	1.74
ijpeg	297.2	2.38	20.9%	1.15

In some cases, our region and program speedups differ from those in previous work [15]. There are three reasons for this discrepancy. First, we are not rescheduling the code to minimize the critical paths for forwarding (which had a large performance benefit in previous work [15]), so we must execute the code as compiled for a sequential machine. Second, we are now modeling superscalar processors, so all caches (instruction, data, and branch prediction) may suffer from decreased locality. Third, good speedup is more difficult to achieve when comparing against a superscalar execution of the sequential application.

Table 7 lists the applications that are used in this study. `Buk` is an implementation of the bucket sort algorithm, `compress` performs data compression, `sc` computes a spreadsheet, and `ijpeg` performs various algorithms on images. We perform speculation on loops that were hand-selected as good candidates for TLDS (TLDS is not limited to loops, but other speculative regions are more difficult to implement [14]). To decrease the relative overheads of TLDS, some of the loops have been unrolled, as indicated in Table 7. (Note that the "*number of times unrolled*" column in Table 7 corresponds to the number of *additional* copies of the original loop body that exist after unrolling; hence a value of *zero* indicates that the loop has not been unrolled, a value of *three* indicates that there are *four instances* of the original loop body, etc.)

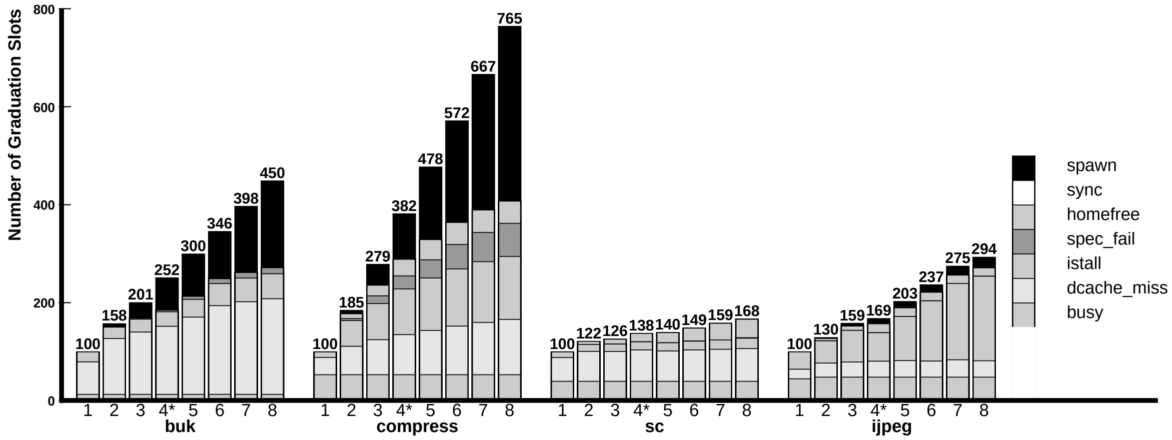
4.2 Baseline Performance

Table 8 summarizes the performance of each application on the baseline architecture. We achieve a variety of region speedups ranging from 5% to 190%. Program speedups are limited by *coverage* (the portion of the application that performs TLDS execution), and range from 4% to 74%. `Sc` and `ijpeg` achieve the best region speedups in part because their large epochs help hide TLDS overheads.

In the results that follow, we will only analyze the performance of speculative regions so that we may focus on the performance of our architecture during speculative execution. Figure 8(a) shows processor utilization for a varying number of processors. Each bar is broken down into seven categories explaining what happened during all potential graduation slots.[2] The top four sections represent non-graduating slots attributed to the following TLDS-related reasons: waiting to begin a new epoch (*spawn*); waiting for synchronization for a

[2] The number of graduation slots is the issue width (4 in this case) multiplied by the number of cycles multiplied by the number of processors.

(a) Processor Utilization



(b) Impact on Execution Time

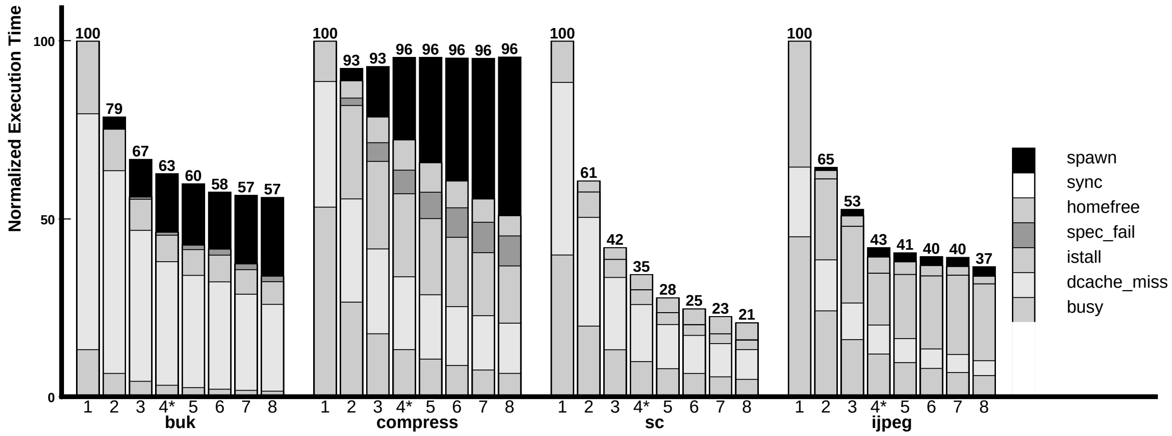


Figure 8: Performance of TLDS on a varying number of processors. The number of processors in our baseline architecture is 4, as indicated by the *. Part (a) is scaled to the number of processors multiplied by the number of cycles, and shows the utilization across all processors. Part (b) is shows normalized execution time.

forwarded location (*sync*); waiting to become homefree (*homefree*); and graduating speculative instructions for an epoch that is violated (*spec_fail*). The remaining sections represent regular execution: the *busy* section is the number of slots when instructions actually graduate and commit; the *dcache_miss* section is the number of non-graduating slots attributed to data cache misses; and the *istall* section is all other slots where instructions do not graduate.

As we increase the number of processors and begin to speculatively execute in parallel, the heights of the bars increase—if we were achieving linear speedup, we would expect the bars to remain the same height. Although we do not achieve linear speedup, the growth of the bars for `sc` and `ijpeg` is slow, indicating that they scale well. `Compress` does not scale as well: as the number of processors increases, `compress` spends more time waiting for epochs to spawn and performing speculation that fails.

In `Compress`, we synchronize around a particular set of references to a location to decrease the number of violations that occur (effectively forwarding the value). Interestingly, the time spent waiting for forwarded values to arrive (*sync*) is negligible. This is likely due to the large amount of time spent waiting for epochs to spawn. `Buk` also spends a significant amount of time waiting for epochs to spawn. For `sc` and `ijpeg`, the spawn time is almost negligible—the amount of work per epoch is sufficient to hide this overhead for these

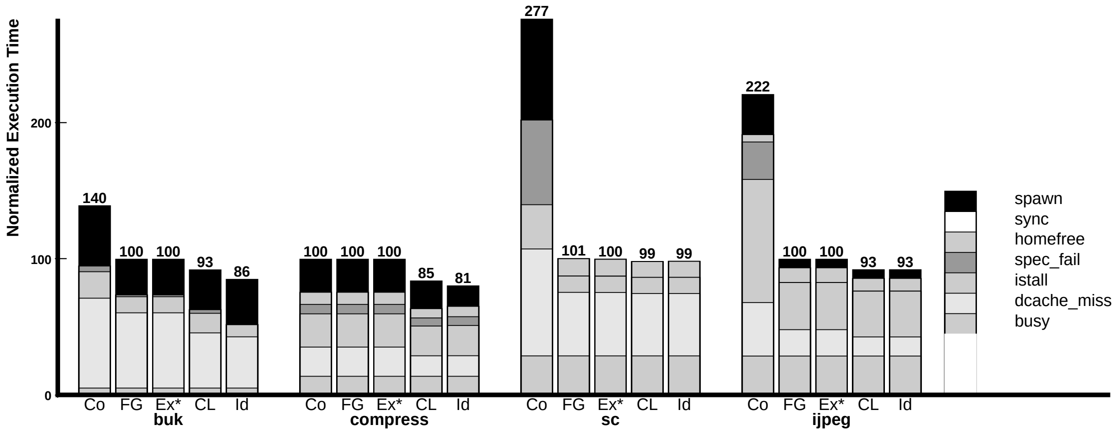


Figure 9: Impact of using cache coherence to detect dependences (**Co** is the basic coherence scheme, **FG** builds on **Co** by adding fine-grain SM bits, **Ex** builds on **FG** by tracking exposed uses, **CL** builds on **Ex** by no longer modeling coherence, and **Id** builds on **CL** by tracking data dependences at a word granularity). The baseline architecture is **Ex**, as indicated by the *.

two applications. Notice that time spent performing failed speculation and time spent awaiting the homefree token only account for a small portion of the execution time for all applications.

Figure 8(b) shows the bars from Figure 8(a) normalized by the sequential execution time. We see that `buk`, `sc`, and `ijpeg` benefit from increased speedup as the number of processors increase, indicating that there is much available speculative parallelism. The performance of `compress` improves less as the number of processors increases from two, so our baseline architecture of four processors does not perform as well as a two-processor machine in this case. This is an artifact of the violation polling scheme modeled by our simulator: when there are many violations and epochs are short, the time to restart epochs can be a bottleneck. An interrupt-based approach (which is difficult to simulate with our current simulator) where epochs can restart immediately when a violation occurs would not suffer from this effect. `Sc` scales quite well, achieving a region speedup of 4.76 on 8 processors.

4.3 Using Cache Coherence to Track Data Dependences and Buffer Speculative State.

We now compare several implementations of TLDS hardware support. These implementations are of varying complexity, including two which are not practical to implement. Figure 9 shows execution time normalized to the baseline architecture for the different implementations. The first bar, **Co**, represents the basic coherence scheme. For this case, output dependences cause violations. The **FG** case builds on **Co** by adding fine-grain SM bits, therefore allowing speculation to succeed in the presence of output dependences. We see that `buk`, `sc`, and `ijpeg` have crucial output dependences which limit performance unless handled, while `compress` does not. The **Ex** case builds on **FG** by only setting the SL bit for *exposed* uses (see Section 2.5)—this is the baseline architecture, although `sc` is the only application which benefits from tracking exposed uses. The next two cases cannot be built, but they indicate the efficiency of our scheme. Case **CL** builds on **Ex** by no longer modeling coherence between the caches (no longer sending upgrade requests), but still checking dependences at a cache line granularity. By comparing with **Ex**, we see that having coherence increases the number of data cache misses, but that the impact on performance is reasonable. Finally, **Id** builds on **CL** by tracking dependences at a word granularity. The difference in height between the **CL** and **Id** bars indicates whether a application suffers from violations due to false dependences,[3] which is the case for both `buk` and `compress`. This "ideal" architecture only performs between 1% and 19% better than our baseline architecture, indicating that our baseline architecture is quite effective.

[3] A false dependence occurs when separate parts of a cache line are speculatively read and written, thereby triggering a violation unnecessarily.

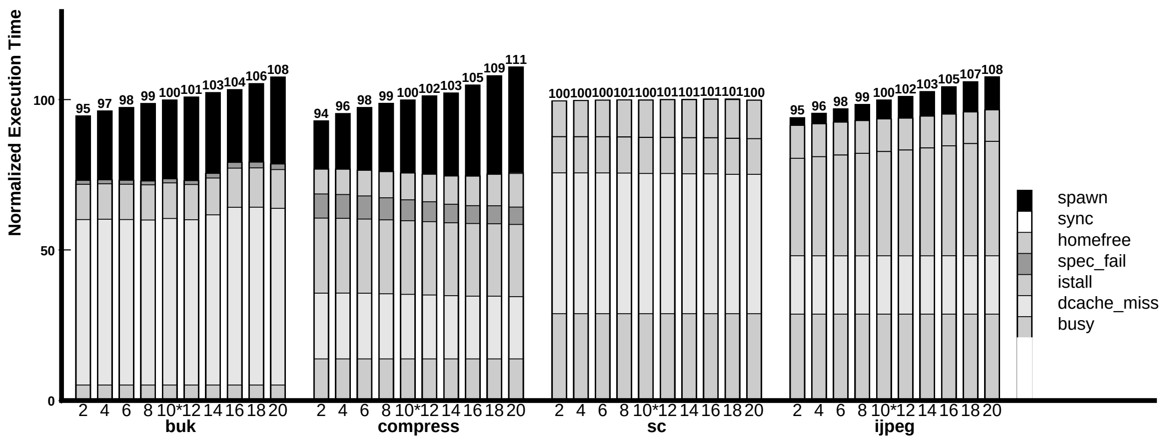


Figure 10: Impact of varying communication latency (in cycles). The baseline architecture has a communication latency of 10 cycles, as indicated by the *.

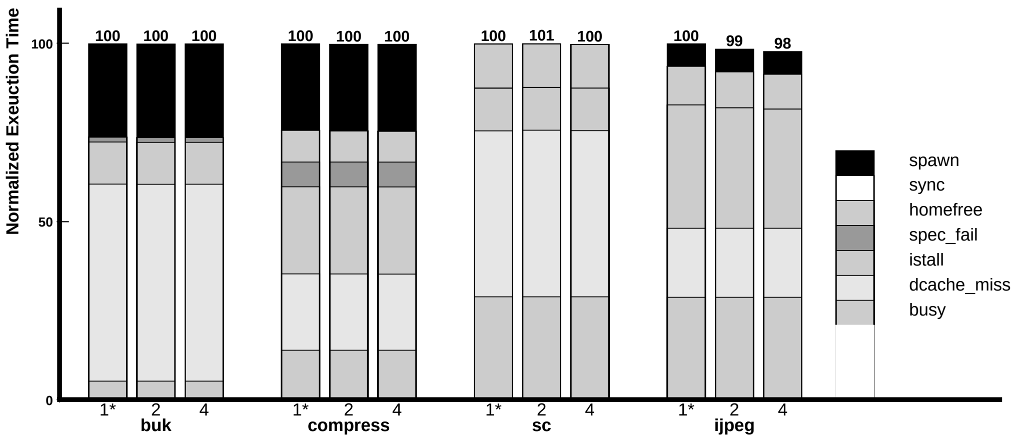


Figure 11: Impact of varying the number of upgrade requests issued per cycle when flushing the ORB. The baseline architecture can issue 1 upgrade request per cycle as indicated by the *.

4.4 Communication Latency

In Figure 10, we vary the communication latency from two to twenty cycles (for our baseline architecture it is ten cycles). Evidently, `compress` and `buk` are the most sensitive to communication latency. Increasing communication latency increases epoch spawn time for both applications, but decreases the amount of failed speculation for `compress`. This decrease is fortuitous: the epochs become less and less overlapped and therefore have fewer out-of-order memory references which could cause violations. `Sc` provides a substantial amount of work per epoch, and as a result it is almost completely insensitive to communication latency. Given the region and program speedups in Table 8, we see that `buk`, `sc`, and `ijpeg` would still speedup with a higher communication latency than our baseline, while `compress` would not.

4.5 Flushing the ORB

Figure 11 shows the impact of increasing the number of upgrade requests that the ORB may issue each cycle when committing an epoch. Contrary to expectations, execution time increases slightly for `sc`.[4] This experiment has no effect on `buk` or `compress`, indicating that for those applications each epoch modifies only a small number of cache lines. `Ijpeg`, which issues an average of 4.3 upgrade requests per epoch, benefits only slightly from increased upgrade request bandwidth. This experiment demonstrates that the delay of passing the homefree token is not a performance bottleneck.

5 Conclusions

Extending the cache coherence protocol is a viable way of exploiting TLDS on architectures supporting multiple threads. We have shown a detailed design of such a protocol, and demonstrate that it can be used to exploit speculative parallelism. The hardware which we have simulated is realistic, and does not put any onerous requirements on hardware designers. Of course further performance analysis will be required to design a complete TLDS architecture, but the benchmarks in this paper show the feasibility of the protocol. In the future we will build upon this scalable platform and focus on compilation techniques to target TLDS machines.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] 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.

[3] 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.

[4] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine Multicomputer. In *Proceedings of ISCA 28*, December 1995.

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

[6] 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.

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

[8] S. W. Keckler and W. J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213, May 1992.

[9] 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.

[10] Jeffery Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Lab, February 1997.

[11] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[4] This anomaly is likely due to a change in the execution pattern of the epochs.

[12] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

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

[14] 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.

[15] 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.

[16] 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.

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

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