Architectural Support for Thread-Level Data Speculation

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

November 1997
CMU-CS-97-188

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Contact information: {steffan,colohan,tcm}@cs.cmu.edu.

Abstract

Thread-Level Data Speculation (TLDS) is a technique which enables the optimistic parallelization of applications despite uncertainty as to whether data dependences exist between the resulting threads which would normally make them unsafe to execute in parallel. The basic idea is to speculate that dependences do not exist, and to then recover and restart whenever dependences do occur dynamically. TLDS can accelerate performance when the gain from increased thread-level parallelism outweighs the overhead of failed speculation. The bulk of the support for TLDS is managed by software, but hardware provides two key pieces of functionality through an extension to invalidation-based cache coherence: (i) detecting dependence violations, and (ii) buffering speculative side-effects until they can be safely committed to memory. This paper explores a number of issues regarding architectural support for TLDS, including the software interface to our new hardware mechanisms (which includes extensions to the instruction set), the cache coherence protocol, and further details on how this scheme might be implemented within a modern memory hierarchy. We note that this document is a snapshot of our work in progress in this area, and we expect to refine our scheme in the future as we conduct further experiments as part of the STAMPede project.

Todd C. Mowry is partially supported by a Faculty Development Award from IBM.

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.

Thread-**L**evel **D**ata **S**peculation (**TLDS**) [13] and other similar techniques [10, 12] 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. Under TLDS, a program is broken into dynamic instruction sequences called *epochs* which the compiler believes are likely to be independent. For example, the iterations of a loop might each be an epoch if the compiler believes that cross-iteration dependences are unlikely. Associated with each epoch is a software-managed *epoch number* which specifies the original ordering of the epochs under sequential execution. The epochs are executed in parallel using special hardware support which uses the epoch numbers to detect whether data dependences have been violated, and which also buffers speculative side-effects until they can be safely committed to memory. If a dependence violation is detected, the hardware then notifies software of the problem, and software reacts by executing whatever recovery code is necessary to safely resume execution using the correct data.

One of our target architectures in the STAMPede project[1] is a generic single-chip multiprocessor where each processor has its own primary data cache and all processors (on the same chip) physically share a secondary cache. Our goals are to minimize the amount of new hardware that must be added to support TLDS, and also to avoid degrading the performance of applications which to not utilize TLDS. We propose extending the instruction set to provide new instructions which which enable software to manage TLDS, to use the caches to buffer speculative state, and to extend the cache coherence scheme to detect data dependence violations. This paper presents our initial design of the architectural support for TLDS, as well as a discussion of alternatives.

The remainder of this paper is organized as follows. Section 2 gives a brief example of TLDS and introduces some key concepts. Section 3 discusses many of the issues in designing a software interface to TLDS. Section 4 describes how an invalidation-based cache coherence scheme can be augmented to detect data dependence violations, and Section 5 gives a possible hardware implementation of that scheme. Finally, we conclude in Section 6.

2 Key Concepts

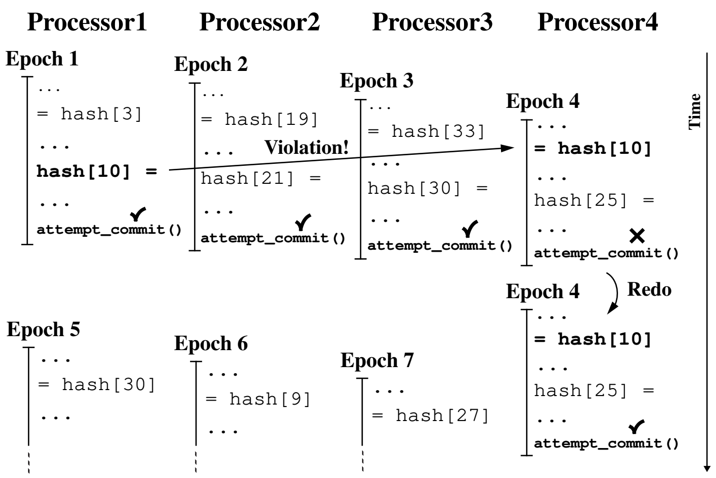
To illustrate the idea behind TLDS, consider the simple `while` loop in Figure 1(a) which accesses elements in a hash table. This loop cannot be statically parallelized due to possible data dependences through the array `hash`. While it is possible that a given iteration will depend on data produced 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). This is the fundamental idea behind TLDS.

We make a number of assumptions in this example. First, 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 single loop iteration as a unit of work (which is called an *epoch*), and *epoch numbers* are used to specify

[1] "STAMPede" stands for "**S**ingle-chip, **T**ightly-coupled **A**rchitecture for **M**ulti**P**rocessing".

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

(a) Pseudo-Code



(b) TLDS Execution

Figure 1: Example of TLDS execution.

and maintain the ordering of memory references so that they correspond to the original sequential execution of the program. Any *violation* of the original program order is detected with the assistance of the TLDS mechanism. In Figure 1(b), 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 has illustrated several key elements of TLDS:

- the ability to run multiple threads in parallel;
- the ability to run a thread *speculatively*, and to discard its side effects if the speculation fails;
- the ability to specify a partial ordering of memory operations (which corresponds to the original sequential execution order), and a mechanism which uses that ordering to detect dependence violations (and hence failed speculation).

Appendix A provides a glossary of terms related to TLDS which will be used throughout this paper. Having introduced these concepts, we now describe an interface between software and the TLDS hardware mechanisms.

3 Software Interface

In this section we explore issues related to the design of a software interface for TLDS, we present a sample implementation of such an interface, and we illustrate the use of this interface with several examples.

3.1 TLDS Interface Design Issues

There are a number of issues that one must consider when designing a software interface for TLDS. Some issues involve variations on things normally done for purely parallel applications, such as creating threads and managing the stacks, and other issues are unique to TLDS, such as generating epoch numbers, passing the *homefree* token, and recovering from dependence violations. We explore a number of these issues in this section, offering design alternatives and discussing tradeoffs between the different designs when appropriate.

3.1.1 Thread Creation

Before we discuss the issues which are unique to speculation, let us first consider one of the requirements for executing even truly parallel programs: the creation of threads. Figure 2(a) shows an example of a `while` loop where the elements of the array `x` are updated using array `y` as an index into `x`. The array `y` has been initialized such that the sixth iteration of the loop (i.e. when `i` = 6) will depend on the output of its preceding iteration (i.e. when `i` = 5); otherwise, the initial set of iterations are independent.

One model of thread creation is to statically create threads on all available processors just once at the start of the loop, and to then reuse these threads to execute all of the epochs. Figures 2(b) and 2(c) show examples of this *recycled threading* model, where the epochs are scheduled dynamically and statically, respectively. These examples assume that parameters are passed to the threads through shared memory (note that `my_i` is a local variable in Figure 2(b)), and that synchronization is performed (with the help of special primitives) upon the check if speculation was successful and upon `join_threads()`.

An alternative to the static thread creation of the recycled threading model is to dynamically fork a new thread per epoch, as illustrated in Figure 3. Under this *one-shot threading* model, a chain of threads is created where each thread is responsible for creating its successor thread (possibly passing along arguments as part of the fork). When there are no more available processors, a fork request can either enqueue the resulting thread so that it will execute when a processor becomes available, or else return a special value (e.g., NULL) indicating that the fork attempt failed. One advantage of creating this chain of one-shot threads is that each epoch will have a *thread descriptor* (`td`) pointing to the next epoch in sequential order. When an epoch needs to pass information quickly to its successor epoch—e.g., the *homefree* token, as discussed later in Section 3.1.3—the thread descriptor can be used to name the target for a fast hardware-based inter-thread communication mechanism (e.g., direct point-to-point messages between processors, rather than the push/pull model of communicating through shared memory). In contrast, without a thread descriptor, an epoch may need to locate the thread which is executing the next epoch through a less direct (and probably less efficient) means, such as looking up its location in a table in memory.

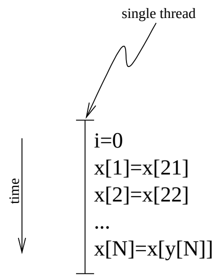
The cost of thread creation is one important consideration in choosing a threading model. One advantage of statically creating threads under the recycled threads model is that the cost of thread creation can be amortized over the many epochs executed by those threads. However, recycled threads still pay some overhead for scheduling work (i.e. epochs) onto the threads. The usual tradeoffs exist between static and dynamic scheduling of epochs: static scheduling (see Figure 2(c)) offers lower run-time scheduling overhead, but can suffer from load imbalance if the epoch sizes are non-uniform; dynamic scheduling (see Figure 2(b)) can distribute the load more evenly, but pays the overhead of accessing a shared work queue (in this case, the variable `i`).

For one-shot threads to be practical, the new thread forking operation must be extremely lightweight since it will be used to start every epoch. Fortunately, there is very little work to be performed upon a thread fork under TLDS. The operating system has already allocated a fixed number of processors to the process of the given TLDS application, and these new TLDS threads are managed strictly at the user level. Our new fork operation simply involves passing an instruction address to another processor where it should begin execution, and possibly including a few argument values (somewhat like a procedure call). Stacks can be pre-allocated and recycled through a pool of free stacks. Due to the simplicity of this lightweight fork operation, there is a reasonable chance that it can be implemented such that it is sufficiently inexpensive.

As we will discuss later in this section, the one-shot thread model simplifies a number of other design issues, such as how to pass the *homefree* token, how to cancel threads beyond control speculation, and how to suspend and resume epochs either to avoid unnecessary dependence violations or to improve load balance. Also, since it is easier to explain TLDS code which uses one-shot threads, we will primarily use this model throughout the remainder of this paper. We would like to note, however, that it is possible for both the one-shot and recycled threads models to co-exist on the same TLDS hardware. As we continue to refine our design in the future, we will explore whether one threading model clearly dominates the other, whether both threading models are viable, and whether there are cases where it is advantageous to combine both models.

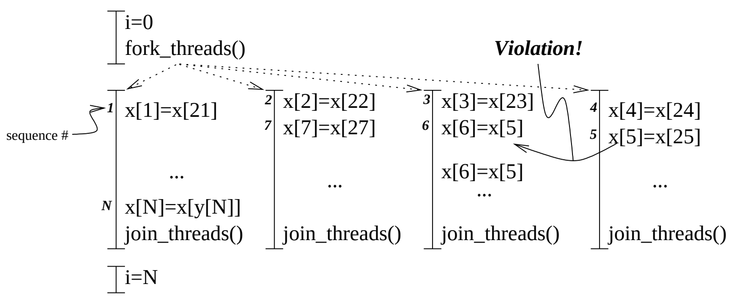
(a) A simple `while` *loop.*

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



(b) Loop executed speculatively using recycled threads and dynamic scheduling.

```
  i = 0;
  /* Fork to all available processors: */
  fork_threads(&start);
start:
  /* ++i is assumed to be atomic,
   * my_i is a local variable: */
  while ((my_i = ++i) < N) {
    begin_speculation(my_i);
    x[my_i] = x[y[my_i]];
    end_speculation();
    /* speculation_succeeded() doesn't return
     * until this is the oldest epoch: */
    if(!speculation_succeeded()) {
      /* Try again non-speculatively: */
      x[my_i] = x[y[my_i]];
    }
  }
  join_threads();
  i = N;
```



(c) Loop executed speculatively using recycled threads and static scheduling.

```
  i = 0;
  /* Fork to all processors, returns number
   * of threads created: */
  num_threads = fork_threads(&start);
start:
  my_i = ++i;
  while (my_i < N) {
    begin_speculation(my_i);
    x[my_i] = x[y[my_i]];
    end_speculation();
    if(!speculation_succeeded()) {
      /* Try again non-speculatively: */
      x[my_i] = x[y[my_i]];
    }
    /* Schedule the next epoch statically: */
    my_i = my_i + num_threads;
  }
  join_threads();
  i = N;
```

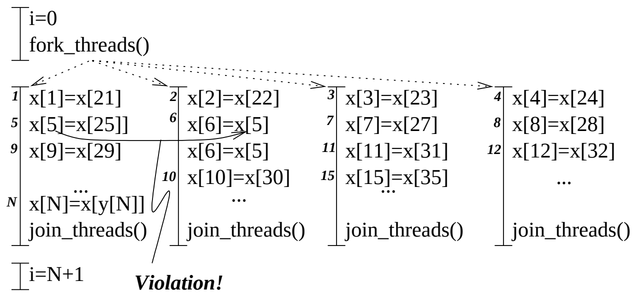
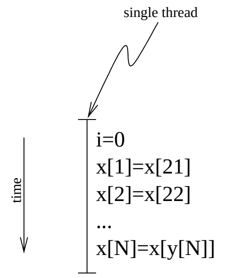


Figure 2: Speculative execution illustrated using a `while` loop.

(a) A simple `while` *loop.*

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



(b) Loop executed speculatively using one-shot threads and dynamic scheduling.

```
  my_i = 0;
start(my_i):
  if(++my_i < N) {
    /* Pass value of my_i as a parameter to the
     * next thread: */
    td = fork_next_thread(&start, my_i);
    begin_speculation(my_i);
    x[my_i] = x[y[my_i]];
    end_speculation();
    if(!speculation_succeeded()) {
      /* Try again non-speculatively: */
      x[my_i] = x[y[my_i]];
    }
    end_thread();
  }
  /* Falls through after last iteration: */
  i = my_i;
```

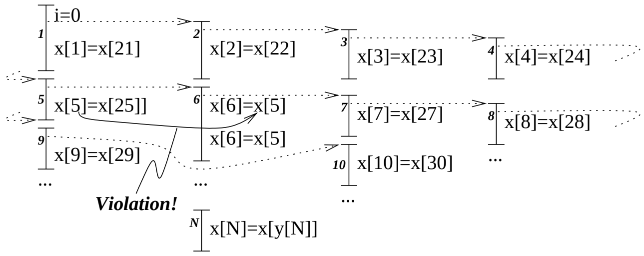


Figure 3: One-shot threads executing a `while` loop.

3.1.2 Stacks in TLDS

Another important issue for TLDS is how to handle references to the stack. As a simplistic starting point, one might consider treating the stack exactly the same as in a sequential execution of the program, with a single stack pointer shared between epochs and a stack in memory that is kept consistent between epochs through the TLDS mechanism. The problem with this approach is that it can result in a large number of unnecessary dependence violations as separate epochs store disparate local values—e.g., procedure linkage information (including return addresses), spilled register values, etc.—into the same locations on the stack. In addition, if the stack pointer changes in the middle of an epoch, the new value must be forwarded to the next epoch. These avoidable dependence violations can effectively serialize execution.

To illustrate this problem, consider the example in Figure 4, where we would like to create two epochs which will call the procedures `foo()` and `bar()` in parallel. If these two epochs share a single stack, then the return addresses for both `foo()` and `bar()` will be written to the same address on the stack, thus resulting in an unnecessary write-after-write dependence violation.

To avoid these problems, we can instead create a separate *stacklet* [7] for each epoch which will hold local variables, spilled register values, return addresses and other procedure linkage information, etc. Since the stacklets have short lifespans they can be kept in a pool for efficient reuse. Figure 5 illustrates that through the use of stacklets, we can safely store the return addresses of the parallel procedure calls in Figure 4 without causing dependence violations.

Once we are executing inside the `foo()` and `bar()` procedures in Figure 5, the fact that they are using stacklets will be indistinguishable from them using a normal stack. If these procedures run to completion without creating any further epochs inside of themselves, then they can simply *(a) Original code.*

```
func() {
  x[foo()] = expr;
  y = x[bar()];
  ...
}
```

(b) Running functions `foo()` *and* `bar()` *in parallel.*

```
func() {
  fork_next_thread(&L1, sequence_num + 1);
  begin_speculation(sequence_num);
  x[foo()] = expr;
  end_speculation();
  end_thread();
L1(sequence_num):
  fork_next_thread(&L2, sequence_num + 1);
  begin_speculation(sequence_num);
  y = x[bar()];
  end_speculation();
  end_thread();
L2(sequence_num):
  ...
}
```

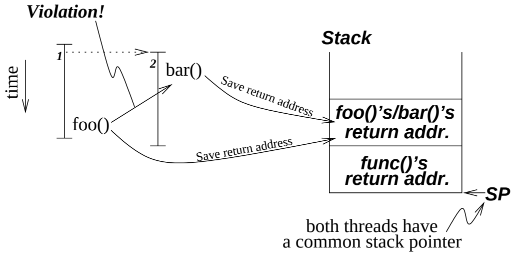


Figure 4: Unnecessary dependence violations caused by two epochs making procedure calls while sharing a common stack.

treat the stacklet that was passed to them as a normal stack, and everything will work correctly. However, if one of these procedures wanted to spawn additional epochs inside itself to exploit more parallelism, then there is one caveat: the procedure must restore its stack pointer to the original value upon entering the procedure, as illustrated by the accesses to SP in Figure 5. The reason for this is that the stack pointer of the thread which executes a procedure return may be pointing to a stacklet which is different from the original stacklet when the procedure was entered. Hence for any procedure which can can spawn new epochs, the stack pointer should be saved in the procedure prologue and restored in the procedure epilogue.

3.1.3 The Homefree Token

We cannot determine whether speculation has succeeded until all previous epochs which a given epoch might depend on have committed their stores to memory. Hence the act of committing speculative modifications to memory is effectively serialized. In our examples so far, we have assumed that if we mark the epoch boundaries with `begin_speculation()` and `end_speculation()`, then the hardware can then figure out which epoch is the oldest and therefore cannot suffer any further dependence violations. For the hardware to do this, it must know all of the epoch numbers currently in use and compare them to determine which one is sequentially earliest. The problem with this approach is that it requires either centralized coordination or else broadcasting all active epoch numbers to all processors, both of which are undesirable.

Rather than computing the earliest epoch based on the set of outstanding epoch numbers, a more efficient approach would be to directly pass an explicit token—which we call the *homefree token*—from the oldest epoch to its successor epoch whenever the oldest epoch finishes committing its stores to memory. Receipt of the homefree token is a green flag which tells the receiving epoch that it now has enough information to evaluate whether or not its speculation has succeeded. If we use the chained one-shot thread creation model (illustrated earlier in Figure 3), then a given thread already has a thread descriptor which can be used to pass the homefree token directly to the next

```
func() {
   /* Save a copy of the stack pointer
    * so we know where to return to in
    * the last thread. */
   parent = SP;

   td = fork_next_thread(&L1, ++sequence_num);
   begin_speculation(sequence_num);
   x[foo()] = expr;
   end_speculation();
   end_thread();

L1(sequence_num):
   td = fork_next_thread(&L2, ++sequence_num);
   /* In second and subsequent threads allocate
    * a stack to hold local variables: */
   SP = new_stack();
   begin_speculation(sequence_num);
   y = x[bar()];
   end_speculation();
   /* Free the allocated stack before ending
    * the thread: */
   free_stack(SP);
   end_thread();

L2(sequence_num):
   ...
   /* Restore the stack of the calling
    * procedure: */
   SP = parent;
}
```

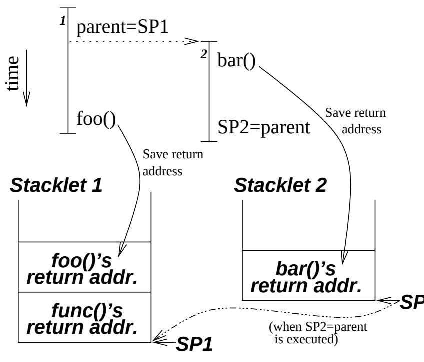


Figure 5: Allocating stacklets for each epoch and for procedure calls.

epoch, thereby avoiding the need for central coordination. Figure 6 illustrates how this might be implemented for a simple loop.

Since the passing of the homefree token is simply a form of producer/consumer synchronization, it can be implemented using normal producer/consumer synchronization primitives. One option is to perform this synchronization purely at a user level through shared memory, without distinguishing it to the hardware as being different from any other synchronization operation. Another option is to make homefree token passing an explicitly hardware-visible primitive. The advantage of this latter approach is that the hardware can perform a number of performance optimizations (discussed later in Section 4.6) if it can recognize the arrival of the homefree token itself.

3.1.4 Notifying Software of Dependence Violations

When the TLDS hardware detects a dependence violation, it must notify software so that it can begin the recovery process. There are two major alternatives for doing this: polling and interrupts. The advantage of polling (illustrated in Figure 7(a)) is that it requires relatively little hardware support—simply a flag which can be set by the violation detection hardware, and which can be read by software. Polling also allows the software to explicitly control when it wishes to react to a dependence violation. The disadvantage of polling is that the processor might not react immediately to a violation. In contrast, the interrupt approach (illustrated in Figure 7(b)) enables a faster response to a violation, but requires more sophisticated hardware support.

Once a violation has been observed, the offending epoch can either restart immediately in the hope that it will not suffer a violation again, or else it can first wait until the homefree token arrives *(b) Threaded code.*

```
  i = 0;
start(i):
  if(i++ < N) {
    td = fork_next_thread(&start, i);
    begin_speculation(i);
    x[i] = x[y[i]];
    end_speculation();
    wait_for_homefree_token();
    if(!speculation_succeeded()) {
      /* Try again non-speculatively: */
      x[i] = x[y[i]];
    }
    pass_homefree_token(td);
    end_thread();
  }
```

(a) Original code.

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

Figure 6: Passing of the homefree token in a `while` loop.

(a) Polling after execution.

```
start:
  begin_speculation(i);
  x[i] = x[y[i]];
  end_speculation();
  wait_for_homefree_token();
  /* If we were violated,
   * run the epoch again: */
  if(is_violated()) {
    x[i] = x[y[i]];
  }
```

(b) Using interrupts and a handler.

```
  begin_speculation(sequence_num);
  /* Jump immediately to "handler" if
   * there is a violation: */
  set_violation_handler(&handler);
  x[i] = x[y[i]];
  end_speculation();
  wait_for_homefree_token();
  goto continue;
handler:
  wait_for_homefree_token();
  x[i] = x[y[i]];
continue:
```

Figure 7: Polling vs. interrupts for handling violations.

before restarting, thereby guaranteeing that no further violations will occur. To determine which approach is better, we plan to conduct experiments in future work to measure how critical response time is when handling a violation, and how likely it is that additional violations will occur if we restart the epoch early.

3.1.5 Generating Epoch Numbers

The epoch numbers represent the partial ordering between epochs on the machine. They include a "process" or "thread" identifier so that memory accesses across truly parallel threads or programs do not interact, as well as a sequence number which determines the ordering within a set of memory accesses that are ordered. Hence an epoch number provides a partial ordering, rather than a total ordering, of memory accesses in a machine. (Further details on epoch numbers will be given later in Section 5.1.) For the sake of simplicity, we will refer only to the sequence number portion of the epoch number in our examples. When we refer to the "epoch" number, we generally mean simply the sequence number.

The simplest way of choosing sequence numbers is to assign consecutive numbers to consecutive epochs in the program. For example, if the epochs corresponded to loop iterations, then the first sequence number might be epoch i, the next $i+1$, the next $i+2$, etc. However, there are also cases where we may want to start epochs out of order. For example, consider the loop in Figure 8(a), *(a) A while loop.*

```
i = 0;
while (foo()) {
    i++;
    body(i);
}
after_loop();
```

(b) Running the loop with a gap inserted.

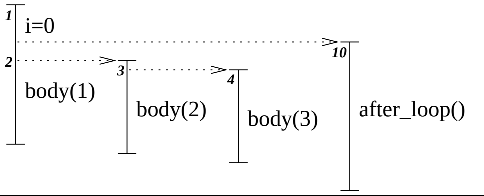


Figure 8: An example illustrating a gap in the epoch numbers.

where the number the number of loop iterations is unknown. If we expect that the number of loop iterations is likely to be relatively small (perhaps based on profiling information), then we may want to spawn an epoch to execute the `after_loop()` code in parallel with the loop iteration epochs. Since we do not know how many epoch numbers are needed for the loop, we can simply create a gap in the epoch numbers which we expect to be large enough to cover all of the loop iterations, and assign the subsequent number to the `after_loop()` epoch. Figure 8(b) shows the case where a gap of nine epoch numbers was created for the loop body, and the code following the loop was assigned epoch number 10. If the loop turns out to have more iterations than the number of reserved epoch numbers, then either the `after_loop()` code can be cancelled, or else the remaining iterations of the loop can be run sequentially.

This technique of leaving gaps in the epoch numbers is useful for more than just loops. When calling a procedure, we may also want to spawn an epoch to execute the code following the procedure in parallel with the procedure itself. A gap can be allocated for the procedure, and the gap size can be passed to the procedure as a parameter. The procedure then uses the sequence numbers in that gap, and can further subdivide the gap if it calls more procedures. An example of how this is done will be provided later in Section 3.3.2.

3.1.6 Epoch Cancellation

Control speculation requires that we be able to speculatively start epochs and decide later that they are not needed. Figure 9(a) shows a `while` loop with an unknown ending condition: we would like to start the epochs in it speculatively, and then cancel any incorrect speculation.

Epoch cancellation is the ability to destroy an epoch and make it appear as though it was never executed. Just like a violation, a cancellation is a message sent from an earlier epoch to a later one, which will set a flag in the target epoch. Unlike a violation, the cancelled epoch does not restart. The cancellation flag may be polled to find out if the epoch was cancelled as in Figure 9(b), or the epoch may just be interrupted and terminated. If an epoch is acquiring locks or allocating memory it may register a handler to clean up after itself if it is cancelled, as shown in Figure 9(c). If the target epoch is to be interrupted, then it helps to have a directed pointer in the previous thread to know where to send the cancellation message. This is another instance where chaining the thread forks helps. Determining which method of cancellation is better requires simulation to find out how often threads are cancelled, and how much time is wasted if we let an epoch complete before it is cancelled.

3.1.7 Thread Switching

If there is a load imbalance between epochs, then a given epoch may be delayed as it waits for the homefree token to arrive. Rather than having the processor remain idle during this time, it may potentially improve performance if we suspend the current epoch, begin working on a newer epoch, and switch back to the original epoch once the homefree token finally arrives. We refer to this switching between epochs in progress as *thread switching*.

An important question is when to switch. Thread switching should be done when the expected delay in the current epoch is larger than the thread switch time, and the new epoch has a reasonable *(a) A simple* `do-while` *loop with unknown bounds.*

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

(b) Cancelling extra loop iterations using polling.

```
  my_i = 0;
start(my_i):
  /* Stop forking new threads if we are cancelled: */
  if(cancelled()) end_thread();
  td = fork_next_thread(&start, my_i + 1);
  begin_speculation(my_i);
  x[my_i] = x[y[my_i]];
  end_speculation();
  if(cancelled()) {
    /* Cascade the cancel to subsequent threads: */
    cancel_thread(td);
    discard_speculative_writes();
    end_thread();
  }
  my_i++;
  if(my_i < x[my_i]) {
    end_thread();
  }
  /* Cancel extra loop iterations and continue: */
  cancel_thread(td);
  i = my_i;
```

(c) Cancelling extra loop iterations using interrupts and a cancel handler.

```
  my_i = 0;
start(my_i):
  /* Set a cancel handler to kill
   * subsequent threads: */
  set_cancel_handler(&cancel_handler);
  td = fork_next_thread(&start, my_i + 1);
  begin_speculation(my_i);
  x[my_i] = x[y[my_i]];
  end_speculation();
  my_i++;
  if(my_i < x[my_i]) {
    end_thread();
  }
  /* Cancel extra loop iterations and continue: */
  cancel_thread(td);
  i = my_i;
  ...

cancel_handler:
  /* Cascade the cancel to subsequent threads: */
  cancel_thread(td);
  discard_speculative_writes();
  end_thread();
```

Figure 9: Cancelling epochs.

chance of succeeding (not being violated). By switching we impose a penalty on the current epoch, since we must switch back to it before it can complete. In return, we allow future epochs to make progress, improving overall throughput if they are not violated.

When we switch, we must preserve the context information including the register state and speculative cache state. As we continue to execute, we must check for violations in the suspended epochs as well as the active epochs, since any epoch can potentially suffer a dependence violation until it is homefree. Section 5.5 discusses an implementation that stores this context information in a way which allows rapid switching between threads, as well as tracking violation status for suspended epochs. For the software interface, we must be aware that each processor may support multiple simultaneous speculative contexts, each with its own epoch number.

A scheduling decision must be made every time a thread switch occurs regarding which epoch to run next. A good heuristic is to run the oldest epoch that is available to be run, since this will cause code on the critical path to be run first.

There are two variations of thread switching: cooperative thread switching and preemptive thread switching. Cooperative thread switching is where an epoch explicitly gives up control of the processor to another epoch. One scenario where a cooperative thread switch might occur is if an epoch reaches the point of waiting for the homefree token, and realizes that the token is not available. At that point, the epoch might explicitly suspend itself and give control to a more speculative epoch. To switch back to the original epoch upon the arrival of the homefree token, however, it is likely that a *preemptive* thread switch is necessary. Hence it is not clear that anything useful can be done with cooperative thread switching alone—preemptive thread switches appear to be necessary if we are to perform thread switching at all.

If hardware is going to preemptively switch between epochs, then it must know which epochs are available to run. Under the one-shot threading model, a suspended epoch corresponds to a suspended thread—if hardware is aware of threads, it seems quite reasonable for it to be able to suspend and resume epochs. Under the thread recycling model, however, it is less clear that preemptive threading is practical, since suspending a given thread suspends more than one epoch—instead, the hardware would be limited to a coarser granularity of switching between these recycled threads. Hence the one-shot threading model appears to be a more natural fit if thread switching is beneficial. To determine whether it is beneficial, we will be conducting future experiments to evaluate how significant the load imbalance problem is, and whether it can be alleviated using thread switching.

3.1.8 Thread Migration

When one processor is overloaded and another processor is idle, it may be desirable to migrate threads from the busy processor to a less heavily loaded one. The operating system may also require the ability to relocate threads if it decides to migrate an entire process to another set of processors. While it may be reasonable to capture and migrate the register state of a TLDS thread, any speculative state in the cache will be wiped out in the course of the move. However, this does not create a problem, since the thread will simply experience the equivalent of a dependence violation when it wakes up, and will have to re-execute its epoch. Since thread migrations are expected to be extremely rare events, the price of a single failed speculation does not appear to be a significant problem. Migrating speculative threads frequently between processors for the sake of improving load balancing is not a viable option for TLDS unless those processors share a primary data cache, since otherwise the speculation is guaranteed to fail.

3.2 Instructions to Support TLDS

We now describe a potential software/hardware interface for TLDS, along with a discussion of why it was chosen. Many of the design decisions were made to simplify subsequent examples and to minimize the complexity of hardware support required. Although we tended to break down more complex TLDS actions into their elemental steps and gave them each explicit instructions, a number of the common sequences of instructions could be combined into a single instruction, and other actions could be made implicit. The interface supports both the one-shot threading and recycled threading models (see Section 3.1.1), although it is biased toward one-shot threading. A precise definition for each interface instruction can be found in Figure 10.

The first thing that is required in a TLDS program is the ability to create threads. There have been many different degrees of complexity and robustness in previous threading implementations, from simple user-level threads to robust kernel-level threads [3]. For TLDS we are primarily concerned with providing concurrency at a very low cost, so we will implement a lightweight `fork()` instruction. Parameters can be passed to the child thread, and a thread descriptor for the child is returned as a handle. To simplify the assignment of sequence numbers, the child thread is assigned the current sequence number plus one by default. A thread that has been created also must end, and this is the purpose of the `end_thread()` instruction.

It is possible for a `fork()` to fail due to a lack of thread contexts, or perhaps by running on a uniprocessor. If the `fork()` fails then the thread can continue executing the current epoch, but will execute the next epoch itself by using `set_sequence_number()` to become the next epoch (this is thread recycling). The `set_sequence_number()` instruction is also useful for initializing the sequence number when a program first starts.

The `become_speculative()` instruction indicates the start of speculation, and the `become_nonspeculative()` instruction marks its end. The address of the violation handler is specified as a parameter to `become_speculative()`. If no violation handler is provided then the default is to invalidate all speculative writes and resume execution at the point where `become_speculative()` was last invoked. A violation handler would be used if the recovery code wanted to do something more sophisticated, such as restoring register state, executing code tailored to recovering only computation that depends on speculative operations, etc.

Before a thread can commit its speculative work, it must be sure that it has not violated any data dependences. At the point when an epoch can no longer violate a data dependence, the epoch becomes *homefree*. Once an epoch is ready to commit its speculative writes, it must first block until it is homefree. This implementation explicitly passes a token from thread to thread called the *homefree token* which determines which epoch is homefree. The instructions that implement this token passing are `pass_homefree_token()` and `wait_for_homefree_token()`. The recipient of the homefree token is specified by a thread descriptor.

When speculation has succeeded, the speculative state in the cache can be written back to memory using the `commit_speculative_writes()` instruction. This instruction is used to show when speculative writes are committed in our examples. In reality, if hardware knows that the homefree token has arrived, it can commit the writes immediately (as shown in section 4.6). There is no corresponding `invalidate_speculative_writes()` instruction, since this is done implicitly whenever a violation is received.

The instructions presented so far are sufficient to write a simple TLDS program. The most common case where TLDS can be used is to parallelize the iterations of a loop, such as a `for` loop. Figure 11 shows how this code would look, and the illustration shows its execution on a four processor multiprocessor. The illustration is idealistic since it assumes uniform length epochs and no violations: in reality this "perfect" ordering would shift due to load imbalance.

The bounds of the `for` loop may not be known at compile time—a program could continue to speculate off the end of a loop in a form of coarse-grain control speculation. The extra iterations must be cancelled once the end of the loop is found through the `cancel_thread()` and `set_cancel_handler()` instructions. The cancel handler is a short routine responsible for relinquishing resources and then terminating the cancelled thread. If no handler is specified, then the thread just ends immediately.

Figure 12 shows how a loop with unknown bounds might be implemented. Control speculation is used, through the `cancel_thread()` instruction, to allow speculation to continue past the unresolved loop test. This example shows a naive technique where the next iteration of the loop is always launched, and extra iterations are cancelled when the end of the loop is found. Control flow prediction techniques could be used to improve this approach. If the number of loop iterations is large, then the number of cancelled iterations will be small compared to the whole loop, and it will be worth devoting all available resources to speculating on the loop. If profiling information shows that the loop usually iterates only a small number of times and the basic block following the

thread_descriptor fork(start_address, ...): Start a new thread in a new speculative context. The start address is where execution will begin, and a thread descriptor is returned which can be used to refer to the thread in future calls. A stacklet is allocated for the new thread. If no processor is able to enqueue this request for a new thread then a thread descriptor of zero is returned. Arguments are passed to the newly created thread using an agreed upon calling convention. The sequence number of the new thread is set to the sequence number of the current thread plus one, although `set_sequence_number()` can also be used to override this setting. {R_Type}

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

void set_sequence_number(sequence_number): Sets the sequence number of the current thread to set up a partial ordering in relation to other threads. If speculative writes are buffered and hardware does not support multiple contexts per thread, then this instruction may block until the homefree token arrives. {R_Type}

void become_speculative(violation_handler_address): Informs the processor that subsequent memory references should be treated as being speculative. If the homefree token is possessed by the current thread then this instruction is a no-op. If a violation occurs during speculation then the processor discards all speculative writes and asynchronously jumps to the failed address. If no failed address is given then the default behavior on a violation is to discard all speculative writes and return to the address of the `become_speculative()` instruction. Calling `become_speculative()` while already speculative is treated as a no-op. {Jmp_Type}

void become_nonspeculative(): Turns off speculation. New memory references are now kept consistent between processors and not buffered. Violations may still occur for previous speculative reads and writes, which remain buffered until the homefree token is received or `commit_speculative_writes()` is called. Calling `become_nonspeculative()` when already nonspeculative is treated as a no-op. {R_Type}

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

void pass_homefree_token(thread_descriptor): Pass the homefree token to the specified thread. This will create a new token unless one has been received by calling wait_for_homefree_token(), in which case it will be passed on. {R_Type}

void commit_speculative_writes(): Makes buffered speculative modifications visible to other threads. This may be performed automatically by the hardware once the homefree token is received (see Section 4.6). The behavior of this instruction is undefined if the thread is speculative. {R_Type}

void set_cancel_handler(handler_address): Sets the address of the cancel handler. If set to null, the thread will be immediately terminated when cancelled. A thread is initially created with a null cancel handler. {Jmp_Type}

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

void suspend_speculation(): Temporarily suspends speculation so that writes are not buffered (to support data forwarding). Incoming violations are still detected while speculation is suspended. This instruction is ignored if the thread is not speculative, and subsequent occurrences of `suspend_speculation()` are ignored. {R_Type}

void resume_speculation(): Resumes speculation after it is suspended. Writes are buffered once again. This is ignored if speculation is not suspended. {R_Type}

error_code violate_thread(thread_descriptor): If the specified thread is speculative it is notified of a violation. If the thread descriptor is null or invalid then this is a no-op. {R_Type}

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

Figure 10: Interface to a TLDS multiprocessor (continued on the next page).

`void restore_sp(sp)`: Frees the current stacklet and sets the stack pointer to the provided one. {R_Type}

In brackets following each description is the encoding form that could be used if these instructions were mapped on to a MIPS type architecture. It is assumed that a new architected register must be created to hold the epoch and sequence numbers.

Figure 10: Interface to a TLDS multiprocessor (continued from the previous page).

loop is likely to be independent of the loop, then a small fixed number of sequence numbers can be reserved for the use of the loop and the post-loop block can be executed immediately as shown in Figure 13. This requires that code be added to forward the homefree token correctly from the last non-cancelled epoch in the loop, and also to handle the special case where not enough sequence numbers were reserved.

In order to forward data from one epoch to another through shared memory we need to suspend speculation so that modifications are visible. This could be implemented with non-speculative memory accesses: by having new write instructions, or by changing the mode of the processor. We implement the latter, with the `suspend_speculation()` and `resume_speculation()` instructions.

It is possible for an epoch to be notified of a violation after forwarding data to a successor, or for it to be violated and know that a successor will also be violated as a result. The `violate_thread()` instruction is used to force a successor epoch to restart.

3.3 Further Examples

We have presented a sample interface for TLDS. Now we will show more complete examples which demonstrate the flexibility of this interface.

3.3.1 Linked List Traversal

Consider the code in Figure 14 which traverses a linked list and does some computation on each list member. List traversal is essentially sequential, but the computation performed on the list elements may be independent. Assuming that the computation on each element is significant, the traversal of the list could be done sequentially while the computation on each element is performed in parallel. Each child epoch is given the next element in the list to operate on, until the end of the list is reached.

3.3.2 Procedure Calls

Another form of speculation involves executing a procedure call in parallel with the code that follows it. In the simple case, the entire procedure is contained within a single epoch as shown in Figure 15. The parent thread forks a child thread, giving the child the follow-on code as a starting point. The procedure and follow-on code are then executed speculatively in parallel, and then committed in sequential order.

However, many procedures contain more than one epoch, and the number of epochs is not likely to be known at compile time—we must therefore leave a gap in the sequence numbers as described earlier in Section 3.1.5. Furthermore, the expected number of epochs in the procedure must be small if speculating on the follow-on code is to be of any benefit. We also must manage the stacks so that the returning thread is consistent with the calling thread.

Figure 16 shows the code for executing a multi-epoch procedure speculatively in parallel with its follow-on code. The number of reserved sequence numbers is passed as an argument to the procedure. If the procedure runs out of reserved sequence numbers, then the thread that was assigned the last sequence number completes the procedure itself. At the beginning of the procedure, the stack pointer is saved so that the last thread may restore the caller's stacklet when the procedure is completed (see Section 3.1.2 for more details). To execute the procedure speculatively, we simply fork the follow-on code and then call the procedure.

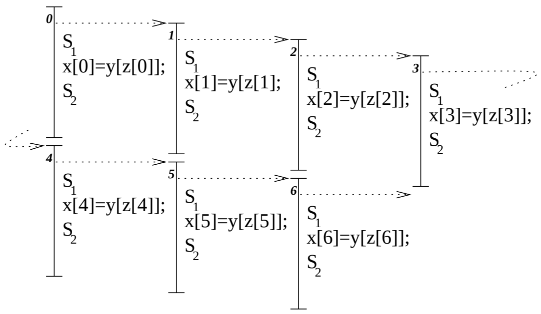
A for loop with a possible dependence:

```
for(i = 0; i < N; i++) {
  S1;
  x[i] = y[z[i]];
  S2;
}
```

expressed as low-level code:

```
  i = 0;
top:
  if(i < N) {
    S1;
    x[i] = y[z[i]];
    S2;
    i++;
    goto top;
  }
```

can be transformed into:




```
  i = 0;
start(i):
  if(i < N) {
    /* Forks the next thread: */
    td = fork(&start, i+1);
    become_speculative();
    S1;
    x[i] = y[z[i]];
    S2;
    wait_for_homefree_token();
    become_nonspeculative();
    commit_speculative_writes();
    if(td == 0) {
      /* Fork failed, recycle the thread: */
      goto start(i+1);
    } else {
      pass_homefree_token(td);
      end_thread();
    }
  }
  /* This is executing in parallel with
   * earlier threads, and may not be
   * homefree yet. Wait until we are.
   * This is not necessary if the code
   * following the loop is also
   * speculative: */
  wait_for_homefree_token();
  /* Code following loop */
```

Figure 11: Threads in a basic for loop.

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

```
for(i = 0; i < f(i); i++) {
    S1;
    x[i] = x[y[i]];
    S2;
}
S3;
```

(b) Expressed as low-level code.

```
    i = 0;
    if(i < f(i)) {
top:
        S1;
        x[i] = y[z[i]];
        S2;
        i++;
        if(i < f(i)) {
            goto top;
        }
    }
    S3;
```

(c) Transformed for TLDS.

```
    i = 0;
    if(i < f(i)) {
start(i):
        set_cancel_handler(&cancelled);
        td = fork(&start, i + 1);
        become_speculative();
        S1;
        x[i] = x[y[i]];
        S2;
        i++;
        loop_test = i < f(i);
        wait_for_homefree_token();
        become_nonspeculative();
        commit_speculative_writes();
        if(loop_test) {
            if(td == 0) {
                goto start;
            } else {
                pass_homefree_token(td);
                end_thread();
            }
        } else {
            /* This was the last iteration. */
            if(td != 0) {
                /* Cancel subsequent threads: */
                cancel_epoch(td);
            }
            goto continue;
        }
cancelled:
        /* Cancel any more epochs if
         * forked: */
        cancel_thread(td);
        end_thread();
    }
continue:
    S3;
```

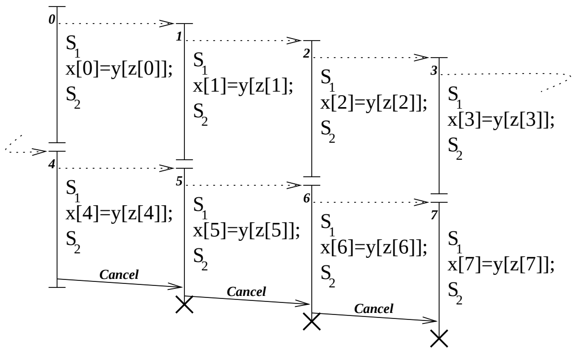


Figure 12: Threads in a `for` loop with unknown ending condition.

```
  i = 0;
  if(i < f(i)) {
    cont_td = fork(&continue);
    gap_size = 10;
start(i, cont_td, gap_size):
    set_cancel_handler(&cancelled);
    if(gap_size > 0) {
      td = fork(&start, i + 1, cont_td,
        gap_size - 1);
    } else {
      td = 0;
    }   become_speculative();
    S1;
    x[i] = x[y[i]];
    S2;
    i++;
    loop_test = i < f(i);
    wait_for_homefree_token();
    become_nonspeculative();
    commit_speculative_writes();
    if(loop_test) {
      if(td == 0) {
        goto start;
      } else {
        pass_homefree_token(td);
        end_thread();
      }
    } else {
      /* This was the last iteration. */
      if(td != 0) {
        /* Cancel subsequent threads: */
        cancel_epoch(td);
      }
      if(cont_td == 0) {
        goto continue;
      } else {
        pass_homefree_token(cont_td);
        end_thread();
      }
    }
cancelled:
    /* Cancel any more epochs if
     * forked: */
    cancel_thread(td);
    end_thread();
  }
continue:
  /* Leave a gap for the loop to execute in: */
  set_sequence_number(sequence_number + 10);
  become_speculative();
  S3;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
```

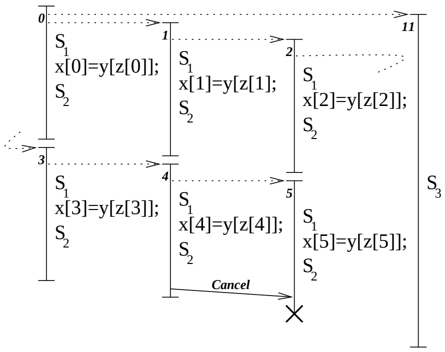


Figure 13: The for loop running the continuation in parallel with the loop body.

```
for(ptr = head; ptr != NULL; ptr = ptr->next) {
  do_computation(ptr->data);
}
```

This can be transformed into (assuming `foo()` *does not change* `ptr->next`*):*

```
    ptr = head;
  start(ptr):
    if(ptr != NULL) {
      td = fork(&start, ptr->next);
      become_speculative();
      do_computation(ptr->data);
      wait_for_homefree_token();
      become_nonspeculative();
      commit_speculative_writes();
      /* Check if fork() failed: */
      if(td == 0) {
        start(ptr->next);
      } else {
        pass_homefree_token(td);
        end_thread();
      }
    }
```

Figure 14: Threaded version of linked list traversal.

3.3.3 Recursive Procedure Calls

The same technique used for procedure calls can be used for recursive calls as well, as shown in Figure 17. An estimate of the recursion depth is used to create a gap in the sequence numbers, and the code that follows the recursion may be run in parallel with the recursion itself.

In Figure 17(a) the first statement is an `if` conditional. When we are executing speculatively, we do not know whether to return at this point or to continue recursing. Only once the `test` is resolved do we know what code to execute next. The `test` is resolved once we are homefree. So if we wait for the homefree token before deciding to continue or return, correct execution is preserved. Figure 17 shows how this would execute. The statement S_1 that follows the conditional is serialized, because it cannot begin until the `if` is homefree, and is therefore fully resolved. This limits the amount of parallelism that can be achieved from this recursive call, but is simpler than doing control speculation.

If more parallelism is desired, then control speculation can be used, as shown in Figure 18 (for the original code in Figure 17(a)). Rather than waiting for the `test` to be resolved, we assume it will be false and continue to run S_1 speculatively. If this assumption was wrong, we then cancel the incorrect speculation once we receive the homefree token after the `if`. Extra code is needed to handle the control speculation, to deal with cancellation and track thread descriptors for cancellation. This extra complexity is the price paid for the extra parallelism that the control speculation makes available.

4 Coherence Scheme

So far we have only described TLDS from a software perspective: we gave the possible thread models and listed the augmented instruction set, assuming a means of detecting data dependence violations and also assuming the ability to buffer speculative memory modifications from memory. We now present possible hardware support for TLDS which provides these mechanisms through an extended cache coherence scheme.

The procedure call followed by further code:

```
foo();
S1;
```

Can be transformed into:

```
  td = fork(&follow_on_code);
  become_speculative();
  foo();
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
  if(td != 0) {
    pass_homefree_token(td);
    end_thread();
  }
follow_on_code:
  become_speculative();
  S1;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
```

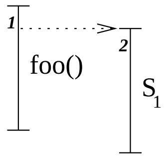


Figure 15: Threading a simple procedure call where the procedure body is executed in a single epoch.

(a) A procedure call with potential to speculate inside the procedure:

```
foo()
{
  for(i = 0; i < N; i++) {
    S1;
    x[i] = y[z[i]];
    S2;
  }
}
/*** The call site is below ***/
foo();
S3;
```

Figure 16: Threading a procedure call (continued on the next page).

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.

There are three options for resolving data dependences, and for each option a different entity is responsible for detecting dependence violations. First, a third-party mechanism could observe all memory operations and ensure that they are properly ordered. This is the approach taken in the Wisconsin Multiscalar's *address resolution buffer* (ARB) [12, 6]. This centralized approach has the drawback of longer load latencies, which hinders the performance of non-speculative execution.

(b) The procedure call run speculatively.

```
/* The number of sequence numbers available
 * is passed as an argument: */
foo(num_seq_nums)
{
  /* Save the stack pointer so the last thread
   * in this function knows where to return to: */
  old_sp = save_sp();
  i = 0;
start(i, num_seq_nums):
  if(i < N) {
    /* Only fork if we have sequence numbers left: */
    if(num_seq_nums > 0) {
      td = fork(&start, i+1, num_seq_nums - 1);
    } else {
      td = 0;
    }
    become_speculative();
    S1;
    x[i] = y[z[i]];
    S2;
    wait_for_homefree_token();
    become_nonspeculative();
    commit_speculative_writes();
    if(td == 0) {
      /* Fork failed, recycle the thread: */
      goto start(i+1, num_seq_nums);
    } else {
      pass_homefree_token(td);
      end_thread();
    }
  }
  /* Wait until we are homefree before changing SP: */
  wait_for_homefree_token();
  /* Restore the caller's stacklet: */
  restore_sp(old_sp);
}
  /*** The call site is below ***/
  td = fork(&followon_code);
  /* Leave a gap of 10 sequence numbers and call
   * foo() outside of a speculative region: */
  foo(10);
  /* We should now be homefree. Pass on the token: */
  if(td != 0) {
    pass_homefree_token(td);
    end_thread();
  }
followon_code:
  /* Create a gap of 10 sequence numbers: */
  set_sequence_number(sequence_number + 10);
  become_speculative();
  S3;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
```

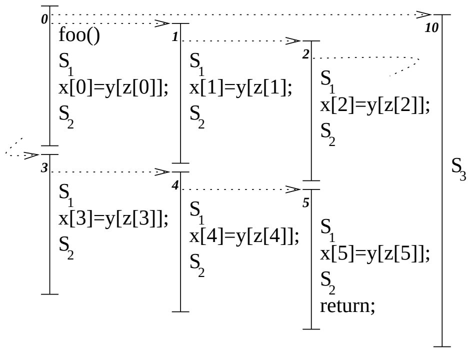


Figure 16: Threading a procedure call (continued from the next page).

(a) A generic recursive procedure:

```
foo(depth)
{
  if(test) return;
  S1;
  foo(depth + 1);
  S2;
}
/*** The call site is below ***/
foo(1);
S3;
```

(b) Threading the recursive procedure without control speculation:

```
/* The number of sequence numbers available
 * is passed as an argument: */
foo(depth, num_seq_nums)
{
  become_speculative();
  condition = test;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
  if(condition) return;
  if(num_seq_nums > 0) {
    td = fork(&call_foo, num_seq_nums - 1);
  } else {
    td = 0;
  }
  /* We are now homefree, so we can run
   * S1 sequentially: */
  S1;
  if(td != 0) {
    pass_homefree_token(td);
    end_thread();
  }
call_foo(num_seq_nums):
  if(num_seq_nums > 0) {
    td = fork(&part_s2, num_seq_nums - 1);
  } else {
    td = 0;
  }
  foo(depth + 1, num_seq_nums - 1);
  if(td != 0) {
    pass_homefree_token(td);
    end_thread();
  }
part_s2(num_seq_nums):
  /* Create a gap in the sequence numbers: */
  set_sequence_number(sequence_number +
            num_seq_nums);
  become_speculative();
  S2;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
}
```

[continued in next column...]

```
  /*** The call site is below ***/
  td = fork(&followon_code);
  /* Leaving a gap of 10 sequence numbers: */
  foo(1, 10);
  if(td != 0) {
    pass_homefree_token(td);
    end_thread();
  }
followon_code:
  /* Create a gap of 10 sequence numbers: */
  set_sequence_number(sequence_number + 10);
  become_speculative();
  S3;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
```

Figure 17: Threads in a recursive procedure (continued on next page).

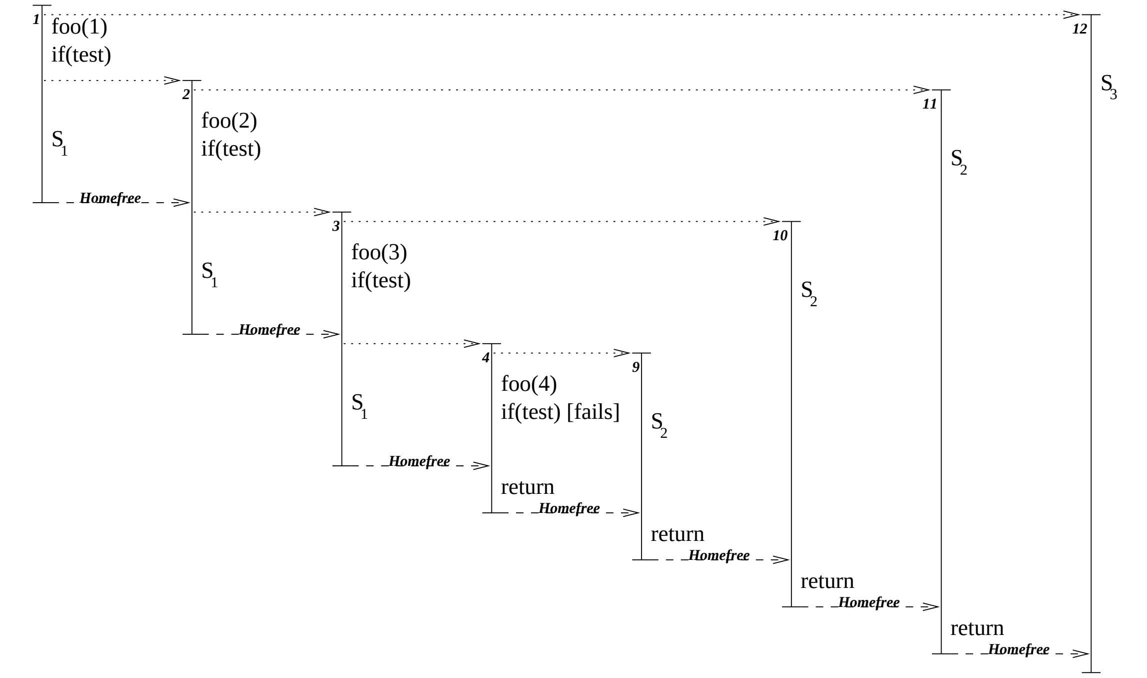


Figure 17: Threads in a recursive procedure (continued from previous page).

Second, the producer could detect dependence violations and notify the consumer. This approach requires the producer to be notified of all addresses consumed by more speculative epochs. On every store, the producer checks if a given address has been consumed by a more speculative epoch and if so notifies that epoch of the dependence violation. This scheme has the drawback that the onus of detection is on the least speculative epoch—we want the least speculative epoch to proceed unhindered. There are also possible race conditions in this approach.

Third, we could make consumers responsible for detecting data dependence violations—the producer reports to the consumers which locations it has produced, and the consumers track which locations have been speculatively consumed. This is similar to the behavior in an invalidation-based cache coherence scheme as follows: whenever a cache line is modified that has recently been read by other processors, an invalidation message is first sent to all other caches that have a copy of the line. To extend this behavior to detect data dependence violations, we simply need to track which locations have been *speculatively* loaded, and whenever a less speculative epoch modifies the same location (as indicated by an arriving invalidation message), we know that a violation has occurred.

4.1 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 19, 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 upgrade request containing its epoch number. When the upgrade request is received, three things must be true for this to be a RAW dependence violation. First, the same cache line that the upgrade request is for

```
/* The number of sequence numbers
 * available is passed as an argument: */
foo(depth, num_seq_nums, caller_s3_td)
{
  set_cancel_handler(&cancelled);
  if(num_seq_nums > 0) {
    td = fork(&part_s1,
        num_seq_nums - 1);
  } else {
    td = 0;
  }
  become_speculative();
  condition = test;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
  if(td != 0) {
    if(condition) {
      cancel_thread(td);
      return;
    } else {
      pass_homefree_token(td);
      end_thread();
    }
  } else {
    if(condition) return;
  }
part_s1(num_seq_nums):
  set_cancel_handler(&cancelled);
  if(num_seq_nums > 0) {
    td = fork(&call_foo,
        num_seq_nums - 1);
  } else {
    td = 0;
  }
  become_speculative();
  S1;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
  if(td != 0) {
    pass_homefree_token(td);
    end_thread();
  }
call_foo(num_seq_nums):
  set_cancel_handler(&cancelled);
  if(num_seq_nums > 0) {
    td = fork(&part_s2, num_seq_nums - 1);
  } else {
    td = 0;
  }
  foo(depth + 1, num_seq_nums - 1, td);
  if(td != 0) {
    pass_homefree_token(td);
    end_thread();
  }
```

[continued in next column...]

```
part_s2(num_seq_nums):
  set_cancel_handler(&b_cancelled);
  /* Create a gap in the sequence numbers: */
  set_sequence_number(sequence_number +
            num_seq_nums);
  become_speculative();
  S2;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
  return;
cancelled:
  if(td) cancel_thread(td);
  end_thread();
b_cancelled:
  /* We must cancel the S3 thread
   * created by our caller: */
  if(caller_s3_td)
    cancel_thread(caller_s3_td);
  end_thread();
}
  /*** The call site is below ***/
  td = fork(&followon_code);
  /* Leaving a gap of 20 sequence numbers: */
  foo(1, 20, 0);
  if(td != 0) {
    pass_homefree_token(td);
    end_thread();
  }
followon_code:
  /* Create a gap of 20 sequence numbers: */
  set_sequence_number(sequence_number + 20);
  become_speculative();
  S3;
  wait_for_homefree_token();
  become_nonspeculative();
  commit_speculative_writes();
```

Figure 18: Threads in a recursive procedure with control speculation (continued on next page).

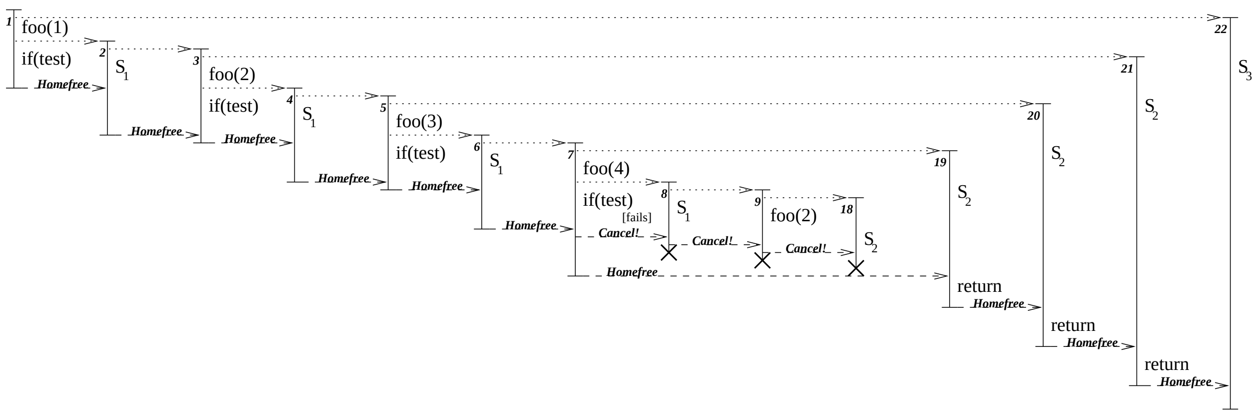


Figure 18: Threads in a recursive procedure with control speculation (continued from previous page).

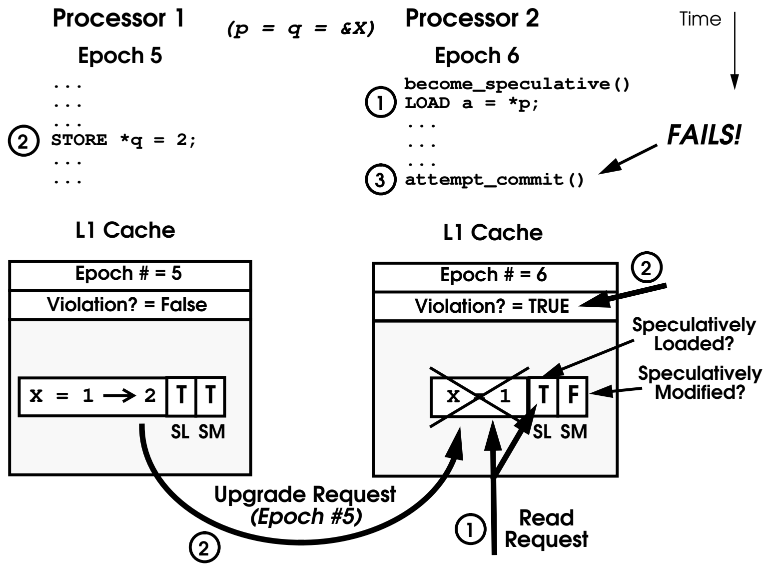


Figure 19: Using cache coherence to detect a RAW dependence violation.

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 upgrade request must tell us that the upgrade request 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.

4.2 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 multithreaded or single-threaded processors, and shared-memory architectures—not necessarily restricted to multiprocessors on a single chip. This

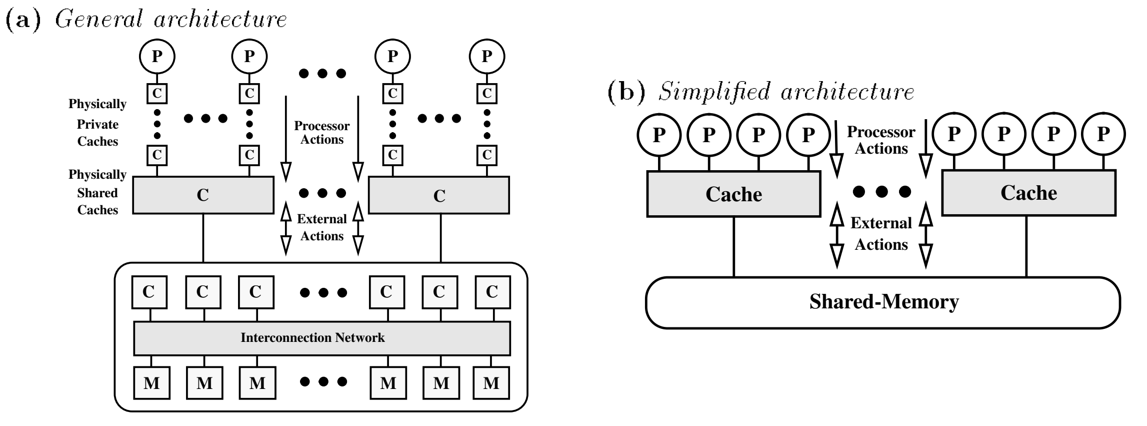


Figure 20: Base architecture for the TLDS coherence scheme.

includes a simultaneously multithreaded processor, single-chip multiprocessor, multiprocessors with shared caches, multiprocessors with private caches, or even software distributed-shared-memory architectures.

For simplicity, we assume some shared-memory architecture that supports an invalidation-based cache coherence scheme where all hierarchies enforce inclusion. Figure 20(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 20(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 20(a) is not necessary for the purposes of describing our cache coherence scheme. Instead, Figure 20(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.

4.3 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, upgrade requests 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

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 modifed exclusive
SpLMS	speculatively loaded and modified shared

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 upgrade requests to be sent out when side-effects are committed to memory. It is also interesting to note that the states *SpMS* and *SpLMS* imply that the cache line is both speculatively modified and shared. This means that it is possible for more than one modified copy of a cache line to exist as long as no more than one copy is non-speculative and the rest of the copies are speculative.

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

For speculation to succeed, any cache line with a speculative state must remain in the cache 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.

4.4 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 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 multi-threaded 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.

Table 2: Processor-initated 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

- 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.

4.5 Cache Actions

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

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

The next group of actions are performed at the shared cache controller. The violation action (*G.Viol*) indicates that speculation has failed and must recover. The (*G.Suspend*) action indicates that a violation is about to occur, and the only way to avoid it is to suspend the thread. For example, if a speculative cache line is about to be replaced, the thread which caused the replacement could be

Table 3: Actions generated by the shared cache controller

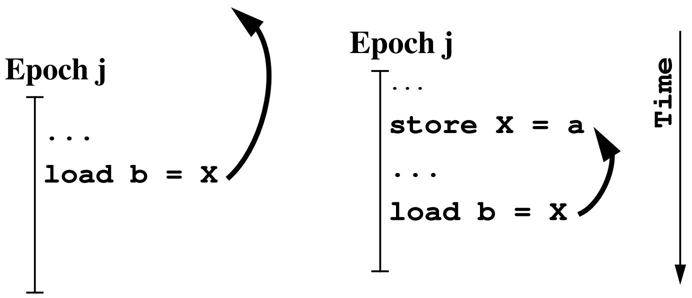
Sent To	Action	Description
External	G.ER	Generate external read.
	G.EREx	Generate external read exclusive.
	G.EWb	Generate external writeback (the line is no longer cached).
	G.EU	Generate external update-line (like a write, except the line remains cached).
	G.EUp	Generate external upgrade request (request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

suspended until the epoch which owns the speculative cache line becomes homefree. If a *G.Suspend* action is too complex to implement, it may be conservatively replaced with a real violation (*G.Viol*).

If two epochs speculatively modify the same cache line, there are two ways to resolve the situation. One option is to simply violate the 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, 4]. 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, and therefore the more speculative of two epochs which both speculatively modify the same cache line does not need to be violated: the speculative modifications may be combined later. A speculatively modified cache line may not change to the dirty state until *G.FlushORB* completes.



(a) Exposed Use **(b)** Un-exposed Use

Figure 21: Exposed and un-exposed uses.

The two conflict-misses described previously in Table 2 do not necessarily have to result in violations. If cache line replication is supported, a some violations can be avoided. However, when replication is not possible we must be careful which epoch is suspended or violated, since the wrong choice could result in deadlock or even livelock. The action *G.Progress* only suspends or violates 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* violates epoch `E`; otherwise, *G.Progress* performs *G.Suspend* for the current epoch.

The last group of actions are actually conditions evaluated by the coherence mechanism. Since we want to maintain 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]. In Figure 21(a) we see that the load is exposed since the location `p` has not already been defined by the epoch. Conversely, in Figure 21(b) we see that the load is not exposed since the location `p` has been defined by the a previous store in the same epoch. We only have to consider a location to be speculatively loaded if it is an upwards exposed use, otherwise the load cannot cause a data dependence violation. To differentiate these cases, we will check 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.

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

4.7 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 *EUpSp* $\times$ *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—success request occurs, as shown by *EUpSp* $\times$ *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 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 violate the epoch, or suspend until the epoch becomes homefree at which point we may allow the speculative modifications to be written-back to the external memory system.
- *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.

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

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

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

- *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

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

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

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

HFree × *SpLMS*, after waiting for *G.FlushORB* to complete, the speculatively modified cache line is combined with the current external copy before changing to the dirty state.

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

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

4.8 Coherence in the External Memory System

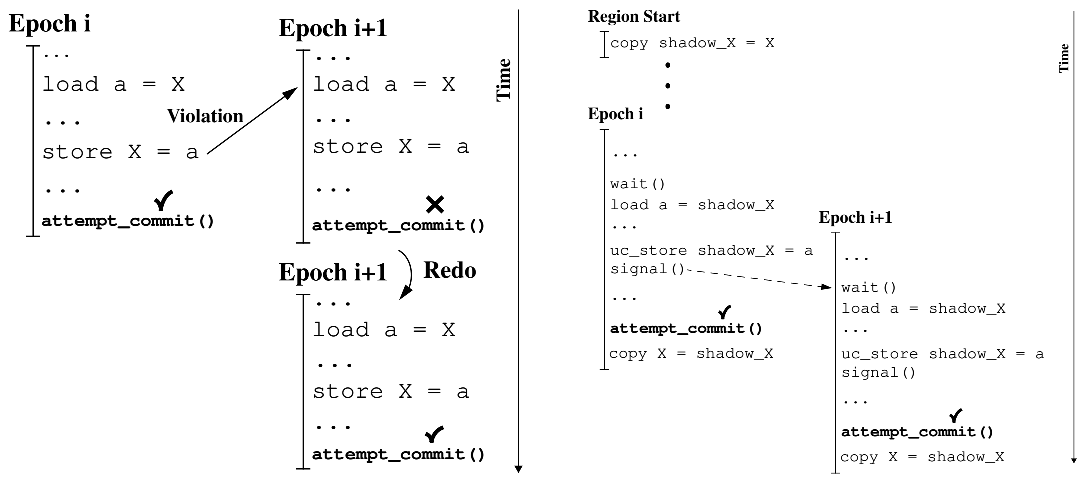
Coherence with support for speculation in the external memory system is quite similar to regular coherence. As listed in Table 4, we require the following standard coherence actions: read (*ER*), read-exclusive (*EREx*), invalidate (*EI*), and upgrade requests (*EUp*). A read is a request for a copy 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 requestor 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.

4.9 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



(a) Data Dependence Violation **(b)** Violation Avoided Through Forwarding

Figure 22: Simple forwarding, when the forwarded location has provably not had its address taken (uc_store represents an uncached store).

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 [5, 8, 11].

As shown in Figure 22(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.

Figure 22(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 is left as future work.

5 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.

(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*

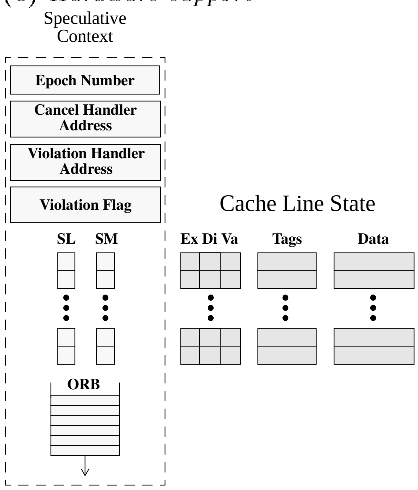


Figure 23: Encoding of L1 cache line states

5.1 Epoch Numbers

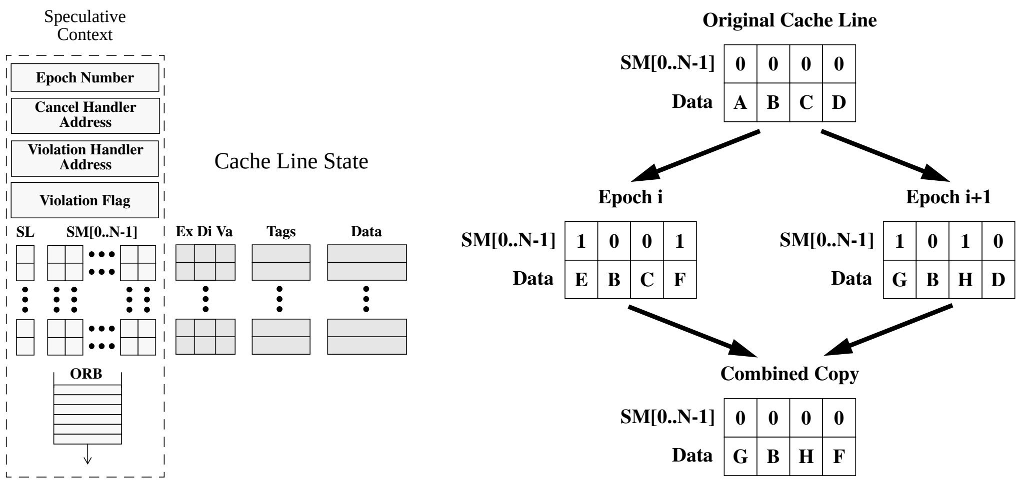
In previous sections we describe the use of epoch numbers in determining the relative order of two epochs. In the coherence scheme, an epoch number is associated with every speculative cache line and every speculative coherence action. The implementation of epoch numbers must address several issues. 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 or leave gaps in the epoch number sequence as described in Section 3.1.5. One solution is to have large integer epoch numbers (such as 32 bits or even larger) and then use signed comparisons to determine the relative order of the corresponding epochs. Signed comparisons 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.

5.2 Cache Line State Encoding

We encode the speculative cache line states given in Table 1 using five bits as shown in Figure 23(a). Two bits, speculatively loaded (*SL*) and speculatively modified (*SM*), differentiate speculative states from non-speculative states. Figure 23(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. $Va = Va$ & $!SM$).



(a) Fine-grain *SM* bits **(b)** Example of cache line combining

Figure 24: Support for cache line combining.

5.3 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 a lot of 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 23(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 violated. 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.

5.4 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 4.5, 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 24(a). 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.

Figure 24(b) 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

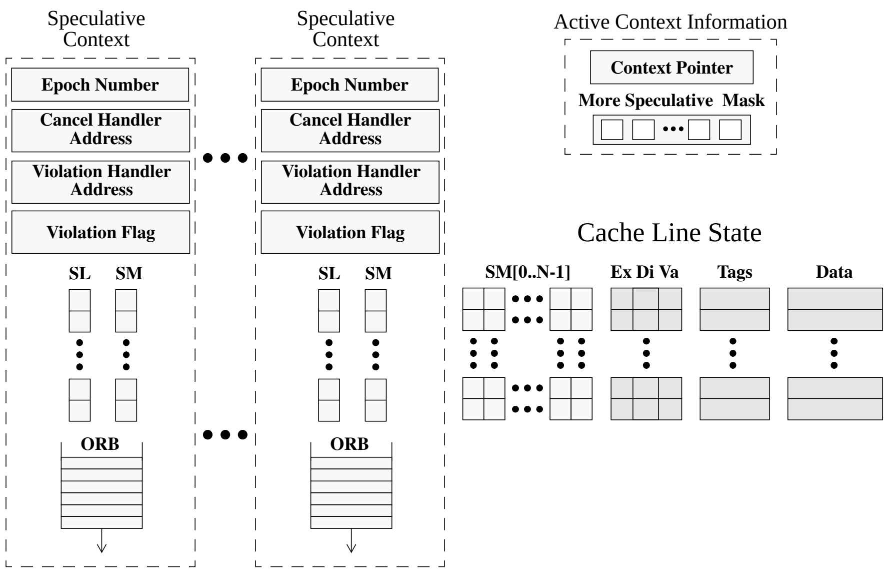


Figure 25: Support for multiple epochs per processor

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 4.5, 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.

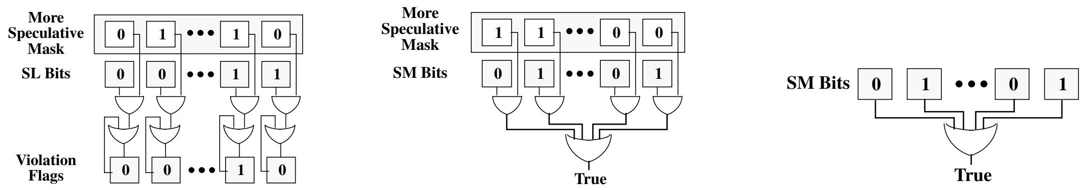
5.5 Support for Multiple Epochs per Processor

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 multi-threading* (SMT) [14].

The coherence scheme as described in Section 4.7 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.

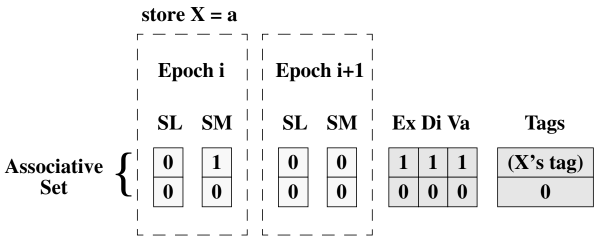
Figure 25 shows hardware support for multiple epochs per processor where we implement several speculative contexts. The *Ex*, *Di*, and *Va* bits for each cache line are shared between all speculative contexts, but each speculative context has its own *SL* and *SM* bits. If fine-grain *SM* bits are implemented, 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 allow fast switching between speculative contexts, we will use a form of indirection as follows: an active context will indicate which speculative context is currently active, and maintain a mask indicating which speculative contexts contain epochs that are more speculative than the active epoch.

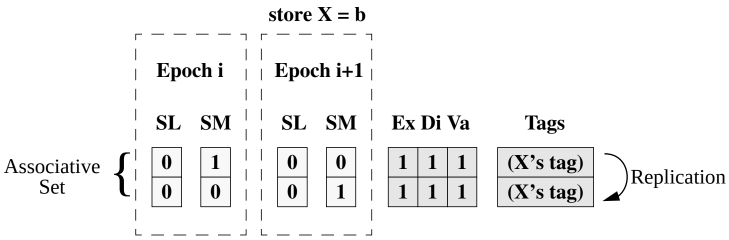


(a) Violation detection **(b)** Read conflict-miss detection **(c)** Write conflict-miss detection

Figure 26: Support for efficient epoch number comparison.



(a) *Epoch i executes a speculative store to location X*



(b) *Epoch i+1 executes a speculative store to location X causing replication*

Figure 27: Example of cache line replication.

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.

As shown in Figure 26(a), we can use the *more speculative mask* to detect violations. If the active epoch stores to a location, then any more speculative epoch which has already speculatively loaded the same location has committed a violation. By taking the `AND` of the more speculative mask with each *SL* bit and `OR`'ing the result into each violation flag, we can detect violations in one operation.

Figure 26(b) illustrates how a read conflict-miss can be detected. Remember that a read conflict miss occurs when the active epoch executes a speculative load and a more speculative epoch has already modified the same cache line. This condition may be checked by taking the `AND` of the more speculative mask with each *SM* bit and checking the `OR` of the results. A write conflict-miss occurs when the active epoch executes a speculative store and any other epoch has already speculatively stored that cache line and may similarly be checked, as shown in Figure 26(c), by taking the *OR* of all *SM* bits.

Another feature of the coherence scheme which supports multiple epochs per processor is the ability to perform replication. Replication is necessary to avoid suspension or violation whenever there is a conflict miss. When replication occurs, we differentiate the copies by checking the *SM* and *SL* bits to find which epoch a given cache line belongs to—if there are two copies of the same cache line, an epoch's *SM* or *SL* bits will only be set for the copy which it owns. In other words, we can consider the *SM* and *SL* bits to be part of the tag match. However, an incoming invalidation can now target more than one cache line—this might make replication difficult to implement. If implementing replication is infeasible, we can instead suspend or violate the epoch requiring replication as described in Section 4.5.

Figure 27 shows an example of cache line replication. In the figure, only the speculative state of the appropriate associative set is shown. In Figure 27(a), `epoch i` executes a speculative store to location `X` which causes the *SM* bit for that cache line to be set. In Figure 27(b), `epoch i+1` executes a speculative store to the same location resulting in a write conflict-miss. Since the other cache line in the associative set is available, the cache line is replicated and `epoch i+1` may proceed.

When an epoch becomes homefree, any replicated cache lines are merged into the most recent copies. If a replicated cache line has only been speculatively loaded, then it is invalidated. If it has been speculatively modified, then it must be combined with the most up-to-date copy as described in Section 5.4.

6 Conclusions

This paper is intended to be a preliminary survey of possible TLDS hardware and software interface designs. To support TLDS, a certain amount of complexity is required: our approach is to minimize the complexity of hardware and maintain its generality, therefore doing as much as possible in software except where absolutely necessary for performance. We have shown that a cache coherence scheme may be extended in a straight-forward manner to detect data dependence violations, and that this scheme can be efficiently implemented in a single-chip multiprocessor. There are many options for the architecture, especially for the software interface, which will require detailed simulation to find the optimal approach.

A Glossary of Terms for Thread-Level Data Speculation

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 commit a violation. If the epoch has not already committed a violation when it receives the homefree token, it may then commit its speculative modifications to memory.

One-Shot Threading: A mode of execution where each thread executes a single epoch then ends.

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

Parallel Thread: A thread which is created by a program to exploit true parallelism. Parallel threads are independent of each other, and violations do not occur between them. Each parallel thread may contain speculative threads.

Parent Stacklet: A stacklet which contains the context of a calling function, to be restored when the function returns.

Recycled Threading: A mode of execution where a single thread executes multiple epochs.

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

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

Stacklet: A pre-allocated stack which is used during the execution of a single epoch.

Thread Descriptor: A handle which uniquely identifies a speculative thread.

TLDS: Thread-Level Data Speculation.

Violation: A thread has committed a true data dependence violation if it has read a memory location that was later modified by a sequentially earlier epoch.

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] Dick Buttlar Bradford Nichols and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., 1996.

[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. 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.

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

[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.

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

[13] 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.

[14] 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.