The STAMPede Approach to Thread-Level Speculation

J. GREGORY STEFFAN
University of Toronto
CHRISTOPHER COLOHAN
Carnegie Mellon University
ANTONIA ZHAI
University of Minnesota
and
TODD C. MOWRY
Carnegie Mellon University

Multithreaded processor architectures are becoming increasingly commonplace: many current and upcoming designs support chip multiprocessing, simultaneous multithreading, or both. While it is relatively straightforward to use these architectures to improve the throughput of a multithreaded or multiprogrammed workload, the real challenge is how to easily create *parallel software* to allow single programs to effectively exploit all of this raw performance potential. One promising technique for overcoming this problem is *Thread-Level Speculation (TLS)*, which enables the compiler to optimistically create parallel threads despite uncertainty as to whether those threads are actually independent. In this article, we propose and evaluate a design for supporting TLS that seamlessly scales both within a chip and beyond because it is a straightforward extension of write-back invalidation-based cache coherence (which itself scales both up and down). Our experimental results demonstrate that our scheme performs well on single-chip multiprocessors where the first level caches are either private or shared. For our private-cache design, the program performance of two of 13 general purpose applications studied improves by 86% and 56%, four others by more than 8%, and an average across all applications of 16%—confirming that TLS is a promising way to exploit the naturally-multithreaded processing resources of future computer systems.

This research has been supported by grants from the National Science Foundation, NASA, IBM, and Intel.
Authors' addresses: J. G. Steffan, Department of Electrical and Computer Engineering, Computer Engineering Research Group, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4; email: steffan@eecg.toronto.edu; C. B. Colohan, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: colohan@cs.cmu.edu; A. Zhai, Department of Computer Science and Engineering, University of Minnesota, 4-192 EE/Csci Building, 200 Union Street, SE, Minneapolis, MN 55455; email: zhai@cs.umn.edu; T. C. Mowry, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: tcm@cs.cmu.edu.

Categories and Subject Descriptors: C.1.4 [**Processor Architectures**]: Parallel Architectures; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Thread-level speculation, chip-multiprocessing, automatic parallelization, cache coherence

1. INTRODUCTION

Due to rapidly increasing transistor budgets, today's microprocessor architect is faced with the pleasant challenge of deciding how to translate these extra resources into improved performance. In the last decade, microprocessor performance has improved steadily through the exploitation of *instruction-level parallelism* (ILP), resulting in superscalar processors that are increasingly wider-issue, out-of-order, and speculative. However, this highly interconnected and complex approach to microarchitecture is running out of steam. Cross-chip wire latency (when measured in processor cycles) is increasing rapidly, making large and highly interconnected designs infeasible [Palacharla et al. 1996; Agarwal et al. 2000]. Both development costs and the size of design teams are growing quickly and reaching their limits. Increasing the amount of on-chip cache will eventually show diminishing returns [Farrens et al. 1994]. Instead, an attractive option is to exploit *thread-level parallelism*.

The transition to new designs that support multithreading has already begun: Sun's MAJC [Tremblay 1999], IBM's Power4 [Kahle 1999], and HP's PA-8800, are all *chip-multiprocessors* (CMPs), in that they incorporate multiple processors on a single die. Alternatively, the Alpha 21464 [Emer 2001], and Intel Pentium IV Xeon support *simultaneous multithreading* (SMT) [Tullsen et al. 1995], where instructions from multiple independent threads are simultaneously issued to a single processor pipeline. The more recently announced Sun Niagara and IBM Power5 are each a CMP composed of SMT-enabled processors.

While designing cost-effective CMP and SMT architectures is relatively well-understood, the real issue is how to use thread-level parallelism to improve the performance of the software that we care about. Multiprogramming workloads (running several independent programs at the same time) and multithreaded programs (using separate threads for programming convenience, such as in a web server) both naturally take advantage of the available concurrent threads. However, we are often concerned with the performance of a single nonthreaded application. To use a multithreaded processor to improve the performance of a single application requires the application to be *parallel*.

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

1.1 Related Work

There are currently many ways of supporting TLS. Of these, one class of schemes is implemented entirely in software [Gupta and Nim 1998; Rundberg and Stenstrom 2000; Rauchwerger and Padua 1995; Cintra and Llanos 2003] but require explicit code and storage to track data dependences and buffer speculative modifications (or provide an undo-log); these schemes are thus only effective for array-based, numeric applications where the portions of memory for which cross-thread dependences need to be tracked are well defined. A software-only approach to TLS support for a large number of arbitrary memory accesses (pointers) is infeasible due to high bookkeeping overheads.

Approaches to TLS that involve hardware support may be divided into two classes: those that are implemented entirely in hardware [Rotenberg et al. 1997; Akkary and Driscoll 1998; Marcuello and Gonzlez 1999], and those that use both hardware and software [Park et al. 2003; Krishnan and Torrellas 1999b; Zhang et al. 1999; Cintra et al. 2000; Hammond et al. 1998; Prvulovic et al. 2001; Frank et al. 1999; Gopal et al. 1998; Krishnan and Torrellas 1999a; Ooi et al. 2001]. Hardware-only approaches have the advantage of operating on unmodified binaries, but are limited, since hardware must decide how to break programs into speculative threads without knowledge of high-level program structure. Hardware-only approaches are generally more complex: transformations and optimization of TLS execution as well as selecting speculative threads can be done in hardware, but is often simpler to do in software.

These schemes may also be classified by their underlying architectures: those that focus on chip-multiprocessor (CMP) architectures [Gopal et al. 1998; Hammond et al. 1998; Krishnan and Torrellas 1999b; Oplinger et al. 1999; Zhang et al. 1999; Frank et al. 1999; Ooi et al. 2001],[1] those that focus on simultaneously-multithreaded (SMT) or other shared-cache architectures [Akkary and Driscoll 1998; Marcuello and Gonzlez 1999; Park et al. 2003], and those that scale beyond a single chip to multiprocessor (MP) architectures [Cintra et al. 2000; Gupta and Nim 1998; Prvulovic et al. 2001; Rauchwerger and Padua 1995; Rundberg and Stenstrom 2000; Zhang and Torrellas 1995; Zhang et al. 1998]. With the exception of the scheme by Cintra et al. [2000], no related approach is scalable both within a chip and also beyond a chip to multiprocessor systems; no related approach is applicable to all multiprocessor systems, chip-multiprocessors, and shared-cache architectures (such as SMT).

[1]Note that the SUDS scheme [Frank et al. 1999] is implemented using the MIT RAW reconfigurable architecture, as opposed to extending a conventional CMP.

Related approaches can also be differentiated by the class of applications that are supported. Of the past evaluations of TLS hardware, two use only numeric applications [Zhang et al. 1999; Cintra et al. 2000], while the rest focus on general purpose applications. In general, only numeric applications contain enough coarse-grain parallelism to tolerate the communication latency of scaling beyond chip boundaries; general purpose applications have more fine-grain parallelism, and hence are limited to scaling within chip boundaries. However, it is desirable for one system to support both forms of speculative parallelism. Any scheme that supports general-purpose applications should also function correctly for numeric applications, but the converse is not necessarily true (ie., a system that works well for numeric applications may not be able to expliot the fine-grain parallelism in general-purpose applications). It is important to note that the work of Cintra et al. [2000] focuses solely on numeric applications.

Finally, related approaches demonstrate a wide variety of hardware implementations, of which the most important feature is the mechanism for buffering speculative state and tracking data dependences between speculative threads. For this purpose, dynamic multithreading (DMT) [Akkary and Driscoll 1998], SUDS [Frank et al. 1999], and Zhang et al. [1999] introduce a new buffer near the processor; the latter two approaches speculatively modify memory and use the buffers to maintain an undo log, while the former uses its buffers to separate speculative modifications from memory. Implicitly-Multithreaded processors [Park et al. 2003] use modified load-store queues, while the Hydra [Hammond et al. 1998] introduces speculative buffers between the write-through first-level caches and the unified second-level cache. These speculative buffers must be sufficiently large to be effective, but adding large amounts of speculation-specific buffering to the memory hierarchy is undesirable. The remaining approaches [Marcuello and Gonzlez 1999; Krishnan and Torrellas 1999a; Gopal et al. 1998; Cintra et al. 2000][2] use the existing caches as speculative buffers. A comprehensive summary and quantitative comparison of several schemes for TLS support is provided by Garzaran et al. [2003].

While there are many important issues regarding the role of the compiler in TLS, this article focuses on the design of the underlying hardware support. The hardware support for TLS presented in this article has the following four goals:

(1) to handle arbitrary memory accesses—not just array references;
(2) to preserve the performance of nonspeculative workloads;
(3) to scale seamlessly within a chip and provide a framework for scaling beyond chip boundaries;
(4) to fully exploit the compiler and minimize the amount and complexity of the underlying hardware support.

[2]The Trace Processor [Rotenberg et al. 1997] approach does not specify means for buffering speculative modifications and tracking data dependences, assuming instead that other work will solve this.

1.2 Contributions

This article makes contributions in two major areas. The first is the proposal of a cooperative approach to TLS that capitalizes on the strengths of both the compiler and hardware. The second is the design and detailed evaluation of a unified scheme for TLS hardware support.

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

1.2.2 *Unified Hardware Support for TLS.* Previous approaches to TLS hardware support apply to either speculation within a chip-multiprocessor (or simultaneously-multithreaded processor) or to a larger system composed of multiple uniprocessor systems. The hardware support for TLS presented here is unique because it scales seamlessly within chip boundaries, and provides a framework for scaling beyond chip boundaries [Steffan et al. 2000]—allowing this single unified design to apply to a wide variety of multithreaded processors and larger systems that use those processors as building blocks. We demonstrate that tracking data dependences by extending invalidation-based cache coherence and using first level data caches to buffer speculative state is both elegant and efficient. This article also contributes a thorough evaluation of chip-multiprocessors with varying numbers of processors and cache organizations, as well as a detailed exploration of design alternatives and sensitivity to various architectural parameters.

2. ARCHITECTURAL SUPPORT FOR TLS

We introduce our cooperative approach to TLS in this section by defining the TLS execution model, the roles of compiler and hardware, and the interface between them.

2.1 Execution Model

The following describes the execution model for TLS as targeted by the compiler and implemented in hardware. First, we divide a program into speculatively-parallel units of work called *epochs*. For this article, each epoch is associated

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

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

Forwarding Frame: A special region on the stack where parameters are passed to new threads, and forwarded values are passed between threads.

Homefree Token: A token that indicates that all previous *epochs* have made their speculative modifications visible to memory. The holder of the token knows that speculation for the current *epoch* is successful.

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

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

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

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

Spawn: A lightweight fork operation that creates and initializes a new *speculative thread*. To make this operation fast, the corresponding *speculative context* can be allocated ahead of time and reused.

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

Speculative Thread: A lightweight thread that is used to exploit speculative parallelism within an *OS-level thread*. While there are several ways to map *epochs* to *speculative threads* [Steffan et al. 1997], in this work we assume each *speculative thread* executes one *epoch*.

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

Fig. 1. Glossary of terms.

with its own underlying *speculative thread*, and creates the next epoch through a lightweight fork called a *spawn*. The spawn mechanism forwards initial parameters and a program counter (PC) to the appropriate processor. An alternative approach (which we do not evaluate in this article) would be to have a fixed pool of speculative threads that grab epochs from a work queue.

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

In the case when speculation fails for a given epoch, all logically-later epochs that are currently running are also violated and squashed. Although more aggressive strategies are possible (for certain speculative regions), this conservative approach ensures that an epoch does not continue to execute when it may have consumed incorrect data.

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

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

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

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

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

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

Fig. 2. TLS instructions for thread and stack management.

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

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

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

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

`void pass_homefree_token(thread_descriptor):` Passes the homefree token to the specified thread.

Fig. 3. TLS instructions for speculation management.

2.2 Software Interface

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

`void set_forwarding_frame(forward_struct_address, size):` Sets the base address and size of the forwarding frame (within the stack frame), specifying a portion of the stack to be copied to the child epoch upon a `spawn`, as well as individual locations to be forwarded mid-epoch.

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

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

Fig. 4. TLS instructions for value forwarding.

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

To speculatively parallelize certain code structures we require support for *control speculation*. For example, a `while` loop with an unknown termination condition can be speculatively parallelized, but superfluous epochs beyond the correct termination of the loop may be created. For correct execution we require the ability to cancel any such superfluous epochs, which is provided by the `cancel_thread` primitive. Each speculative context provides a flag that indicates whether the current epoch has been cancelled.[3] Any epoch that is cancelled also cancels its child epoch if one exists. In contrast to an epoch that suffers a violation, a cancelled epoch is not reexecuted by the run-time system.

2.2.2 *Managing Stacks.* A key design issue is the management of references to the stack. A naive implementation would maintain a single stack pointer (shared among all epochs) and a stack in memory that is kept consistent by the underlying data dependence tracking hardware. The problem with this approach is that speculation would fail frequently and unnecessarily: for example, whenever multiple epochs write/read short-lived temporary values to the same location on the stack, the resulting read-after-write (RAW) dependence violations would effectively serialize execution. In addition, whenever the stack pointer is modified, the new value must be forwarded to all successive epochs. An alternative approach is to create a separate *stacklet* [Goldstein

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

et al. 1996] for each epoch to hold local variables, spilled register values, return addresses, and other procedure linkage information. These stacklets are created at the beginning of the program, assigned to each of the participating processors, and re-used by the dynamic threads. The stacklet approach allows each epoch to perform stack operations independently, allowing speculation to proceed unhindered. We implement instructions for saving and restoring the stack pointer, and returning to the regular stack after using stacklets during the speculative region.

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

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

Several instructions manage the homefree token and associated actions. The `wait_for_homefree_token` primitive stalls the processor until the homefree token is received from the previous epoch. The `commit_speculative_writes` primitive instructs hardware to make all speculative modifications visible to the memory system before returning. Finally, the `pass_homefree_token` primitive sends the homefree token to the next epoch.

2.2.4 *Managing Value Forwarding.* There are often variables that are, at compile time, provably dependent between epochs: for example, a local scalar that is both used and defined every epoch. Without taking any special action the compiler would allocate the variable in memory, and this would then cause dependence violations between all consecutive epochs. Instead it is preferable that the compiler explicitly synchronize and forward that variable between consecutive epochs, avoiding dependence violations. In our approach, the compiler allocates forwarded variables on a special portion of the stack called the *forwarding frame*: the forwarding frame supports the communication of values between epochs, and synchronizes the accesses to these variables. The forwarding frame is defined by a base-address within the stack frame and an offset from that base address; this way, any regular load or store to an address within the predefined forwarding frame address range can be treated appropriately. The address range of the forwarding frame is defined through the software interface at the beginning of every speculative region. Accesses to the forwarding frame are exempt from the data dependence tracking mechanisms of the underlying hardware. Previous work provides a thorough treatment of issues related to the communication of values between speculative threads [Breach et al. 1994; Moshovos et al. 1997; Zhaia et al. 2002; Steffan et al. 2002].

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

2.2.5 *Example of Transformed Code.* These new TLS instructions are sufficient to generate a broad class of TLS programs. Figure 5 shows the transformation of an example loop for TLS execution. The variable `i` is updated and copied to the next epoch through the forwarding frame at each `spawn`, and also serves as the epoch number which is set by the `set_sequence_number` primitive. The code is constructed such that the `spawn` instruction may fail and the current speculative thread will continue and execute the next epoch itself. The final speculative thread will branch around the outermost `if` construct, wait to be homefree, and then continue to execute the code following the loop.

Figure 5(a) illustrates two cross-epoch dependences: one ambiguous dependence (through the array `x`) and one definite dependence (through the scalar `v`). Our TLS interface allows us to speculate on the ambiguous dependence while directly satisfying the definite dependence through value forwarding: as shown in Figure 5(b), the array `x` is modified speculatively while `v` is synchronized using the TLS instructions for forwarding values. The entire forwarding frame is copied when each epoch is spawned, initializing each epoch with the proper value of `i`.

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

3. PRIVATE-CACHE SUPPORT FOR TLS

Having described our TLS execution model, instruction interface, and compiler infrastructure, we now turn our attention to hardware support. In this section we present a unified mechanism for supporting thread-level speculation,

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

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

```
  struct forwarding_frame {
     int i; /* offset 0 */
     int v; /* offset sizeof(int) */
  } ff;
  int i = 0; // Each thread will have a private
  int v = 0; // copy of i and v on it's stacklet
  set_forwarding_frame(&ff,sizeof(struct forwarding_frame));
  ff.i = i;
  ff.v = v;
start:
  i = ff.i;
  if (i < N) {
     ff.i = i + 1;
     td = spawn(&start);
     set_sequence_number(i);
     become_speculative();
     S1;
     x[i] = y[x[i]];
     S2;
     wait_for_value(offsetof(ff,v)); // wait for ff.v
     v = ff.v;
     v += y[i];
     ff.v = v;
     send_value(td,offsetof(ff,v)); // send ff.v to td
     S3;
     wait_for_homefree_token();
     commit_speculative_writes();
     if (td == 0) {
        i++;
        goto start; // fork failed, recycle this thread
     } else {
        pass_homefree_token(td);
        end_thread();
     }
  }
  wait_for_homefree_token();
```

(b) Transformed TLS code.

Fig. 5. An example loop transformed for TLS execution.

which can handle arbitrary memory access patterns (not just array references), and which is appropriate for any scale of architecture with parallel threads, including: simultaneous-multithreaded processors [Tullsen et al. 1995], chip-multiprocessors [Olukotun et al. 1996; Tremblay 1999], and more traditional shared-memory multiprocessors of any size [Laudon and Lenoski 1997]. We focus on private-cache chip-multiprocessor support in this section, and evaluate shared-cache support in Section 6.[4]

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

There are three possible ways to track data dependences at run time; for each option, a different entity is responsible for detecting dependence violations. First, a third-party mechanism could observe all memory operations and ensure that they are properly ordered—similar to the approach of the Wisconsin Multiscalar's *address resolution buffer* (ARB) [Sohi et al. 1995; Franklin and Sohi 1996]. Such a centralized approach has the drawback of increasing load hit-latency which would hinder the performance of nonspeculative workloads.[5] Second, the producer could detect dependence violations and notify the consumer. This approach requires the producer to be notified of all addresses consumed by logically-later epochs, and for the producer to save all of this information until it completes. On every store, the producer checks if a given address has been consumed by a logically-later epoch and if so, notifies that epoch of the dependence violation. This scheme has the drawback that the logically-earliest epoch must perform the detection—but we want the logically-earliest epoch to proceed unhindered!

A third approach is to detect data dependence violations at the consumer. In this approach, consumers track which locations have been speculatively consumed, and each producer reports the locations that it produces to the consumers. Hence a producer epoch that stores to a location must notify all consumer epochs that have previously loaded that location, so that the consumer epochs can verify that proper ordering has been preserved. Our key insight is that this behavior is similar to that of an invalidation-based cache coherence scheme: whenever a cache line is modified that has recently been read by another processor, an invalidation message is sent to the cache that has a copy of that line. To extend this behavior to detect data dependence violations, we simply need to track which locations have been *speculatively* loaded, and whenever a logically-earlier epoch modifies the same location (as indicated by an arriving invalidation message), we know that a violation has occurred.

[4]Steffan [2003] provides an evaluation of TLS on multiprocessors composed of CMPs.

[5]Subsequent designs of the ARB were more distributed [Breach et al. 1996; Gopal et al. 1998].

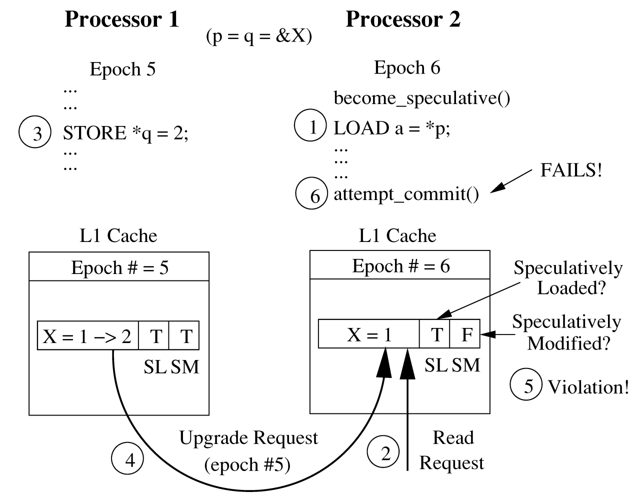


Fig. 6. Using cache coherence to detect a RAW dependence violation.

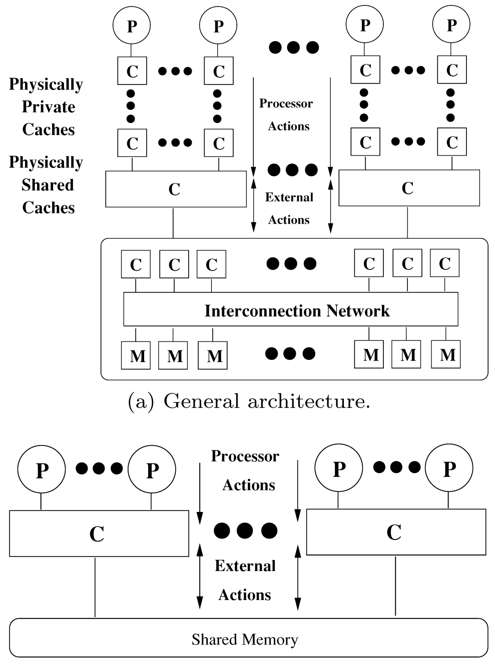
3.1 An Example

To illustrate the basic idea behind our scheme, consider an example of how it detects a read-after-write (RAW) dependence violation. Recall that a given speculative load violates a RAW dependence if its memory location is subsequently modified by another epoch such that the store should have preceded the load in the original sequential program. As shown in Figure 6, the state of each cache line is augmented to indicate whether the cache line has been speculatively loaded (SL) and/or speculatively modified (SM). Each cache maintains a logical timestamp (*epoch number*), which indicates the sequential ordering of that epoch with respect to all other epochs, and a flag indicating whether a data dependence violation has occurred.

In the example, *epoch 6* performs a speculative load (1), so the corresponding cache line is marked as speculatively loaded (2). *Epoch 5* then stores to that same cache line (3), generating an upgrade-request containing its epoch number (4). When the upgrade-request (invalidation) is received, three things must be true for this to be a RAW dependence violation. First, the target cache line of the invalidation must be present in the cache. Second, it must be marked as having been speculatively loaded. Third, the epoch number associated with the upgrade-request must be from a *logically-earlier* epoch. Since all three conditions are true in the example, a RAW dependence has been violated and *epoch 6* is notified (5). When *epoch 6* tries to commit its writes, it fails and restarts (6). The coherence scheme presented in this section handles many other cases, but the overall concept is analogous to this example.

3.2 Underlying Architecture

The goal of our coherence scheme is to be both general and scalable to any size of machine. We want the coherence mechanism to be applicable to any



(a) General architecture.

(b) Simplified architecture.

Fig. 7. Base architecture for the TLS coherence scheme.

combination of single-threaded or multithreaded processors within a shared-memory multiprocessor (not restricted simply to chip-multiprocessors, and so on).

We assume that the shared-memory architecture supports an invalidation-based cache coherence scheme, where all hierarchies enforce the inclusion property. Figure 7(b) shows a generalization of the underlying architecture. There may be a number of processors or perhaps only a single multithreaded processor, followed by an arbitrary number of levels of physical caching. The *speculation level* is the first level where invalidation-based cache coherence begins. We generalize the levels below the speculation level (further away from the processors) as an interconnection network providing access to main memory with some arbitrary number of levels of caching.

The amount of detail shown in Figure 7(a) is not necessary for the purposes of describing our cache coherence scheme. Instead, Figure 7(b) shows a simplified model of the underlying architecture. The speculation level described above happens to be a level of physically shared caches: above these 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 (closer to the processors), we maintain speculative state and buffer speculative modifications. Below the speculation level (further from the processors), we simply propagate speculative coherence actions and enforce inclusion.

3.3 Overview of the Coherence Scheme

The following provides a summary of the key features of our coherence scheme, which requires these key elements: (i) a notion of whether a cache line has been speculatively loaded and/or speculatively modified; (ii) a guarantee that a speculative cache line will not be propagated to regular memory, and that speculation will fail if a speculative cache line is replaced; and (iii) an ordering of all speculative memory references (provided by epoch numbers and the *homefree token*). The full details of the coherence scheme are available in a previous publication [Steffan et al. 1997].

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

To detect data dependences and to buffer speculative memory modifications, we extend the standard set of cache line states as shown in Figure 8(a). For each cache line, we need to track whether it has been *speculatively loaded* (*SL*) and/or *speculatively modified* (*SM*), in addition to exclusiveness. Rather than enumerating all possible permutations of *SL*, *SM*, and exclusiveness, we instead summarize by having two speculative states: *speculative-exclusive* (*SpE*) and *speculative-shared* (*SpS*).

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

3.3.2 *Coherence Messages.* To support thread-level speculation, we also add the three new speculative coherence messages shown in Figure 8(a): *read-exclusive-speculative*, *invalidation-speculative*, and *upgrade-request-speculative*. These new speculative messages behave similarly to their non-speculative counterparts except for two important distinctions. First, the epoch number of the requester is piggybacked along with the messages so that the receiver can determine the logical ordering between the requester and itself. Second, the speculative messages are only hints and do not compel a cache to relinquish its copy of the line (whether the line is relinquished, is indicated in the acknowledgment message).

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

(a) Cache line states.

Message	Description
Read	Read a cache line.
ReadEx	Read-exclusive: return a copy of the cache line with exclusive access.
Upgrade	Upgrade-request: gain exclusive access to a cache line that is already present.
Inv	Invalidation.
Writeback	Supply cache line and relinquish ownership.
Flush	Supply cache line but maintain ownership.
NotifyShared	Notify that the cache line is now shared.
ReadExSp	Read-exclusive-speculative: return cache line, possibly with exclusive access.
UpgradeSp	Upgrade-request-speculative: request exclusive access to a cache line that is already present.
InvSp	Invalidation-speculative: only invalidate cache line if from a logically-earlier epoch.
Condition	**Description**
=Shared	The request has returned shared access.
=Excl	The request has returned exclusive access.
=Later	The request is from a logically-later epoch.
=Earlier	The request is from a logically-earlier epoch.

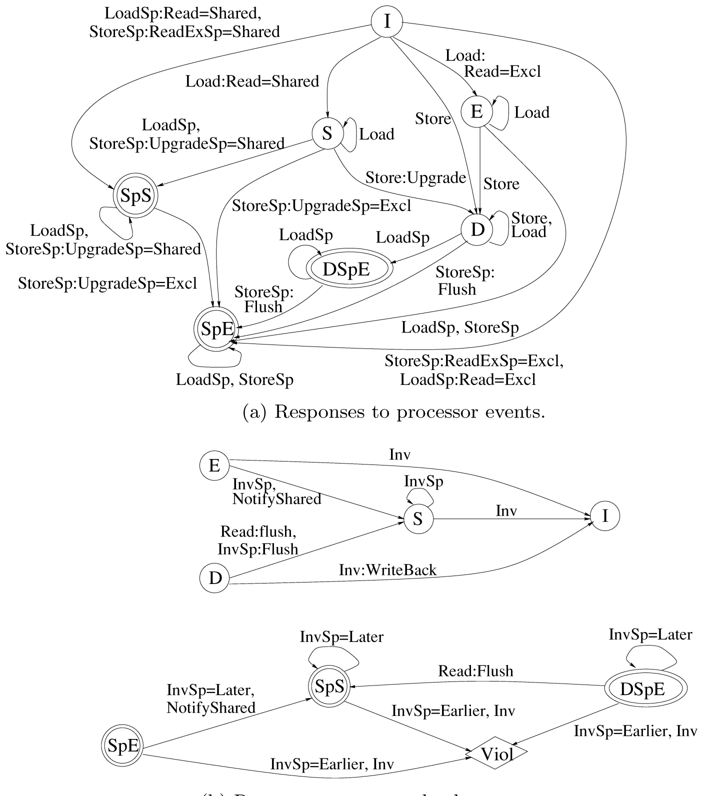
(b) Coherence messages.

Fig. 8. States and messages used in our coherence scheme.

3.4 State Transitions

Our coherence scheme for supporting TLS is summarized by the two state transition diagrams shown in Figures 9(a) and 9(b). The former shows transitions in response to processor-initiated events (speculative and non-speculative loads and stores), and the latter shows transitions in response to coherence messages from the external memory system.

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



(a) Responses to processor events.

(b) Responses to external coherence events.

Fig. 9. State transition diagram for our coherence scheme.

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

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

When a cache line is *dirty*, the cache owns the only up-to-date copy of the cache line and must preserve it. When a speculative store accesses a *dirty* cache line, to ensure that the only up-to-date copy of the cache line is not corrupted with speculative modifications, we generate a *flush*; a flush sends a copy of the cache line to the next level of cache.

3.4.1.1 *Speculative Invalidation of Non-Speculative Cache Lines.* A goal of the coherence scheme is to avoid slowing down non-speculative threads to the extent possible—hence a cache line in a non-speculative state is not invalidated when an *invalidation-speculative* arrives from the external memory system. For example, a line in the *shared* (S) state remains in that state whenever an *invalidation-speculative* is received. Alternatively, the cache line could be relinquished to give exclusiveness to the speculative thread, possibly eliminating the need for that speculative thread to obtain ownership when it becomes *homefree*. Since the superior choice is unclear without concrete data, we compare the performance of both approaches later in Section 5.3.1.

3.4.1.2 *Dirty and Speculative Exclusive State.* As illustrated in Figure 9, when a speculative store attempts to write a cache line that is in the *dirty* (D) state, we generate a *flush*, ensuring that the only up-to-date copy of a cache line is not corrupted with speculative modifications. However, since a speculative load cannot corrupt the cache line, it is safe to delay writing the line back until a speculative store occurs. This minor optimization is supported with the addition of the *dirty and speculative exclusive* state (*DSpE*), which indicates that a cache line is dirty but has been speculatively loaded. Since it is trivial to add support for this state, we include it in our scheme.

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

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

For each line in the *speculative-shared* (*SpS*) state that has been speculatively modified (i.e. the *SM* flag is set), we must issue an *upgrade-request* to acquire exclusive ownership. Once it is owned exclusively, the line may transition to the *dirty* (D) state—effectively committing the speculative modifications to regular memory. Maintaining the notion of exclusiveness is therefore important since a speculatively modified line that is exclusive (i.e. *SpE* with *SM* set) can commit its results immediately simply by transitioning directly to the *dirty* (D) state.

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

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

3.4.4 *Forwarding Data Between Epochs.* As described in Section 2.2.4, we can avoid violations due to predictable data dependences between epochs through the forwarding frame and wait/signal synchronization. As defined by the instruction interface, the forwarding frame and synchronization primitives may be implemented a number of ways, the most simple being through regular memory. Our coherence scheme can be extended to support value forwarding through regular memory by allowing an epoch to make non-speculative memory accesses while it is still speculative. Hence an epoch can perform a non-speculative store whose value will be propagated to the logically-next epoch without causing a dependence violation.

3.5 Implementation

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

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

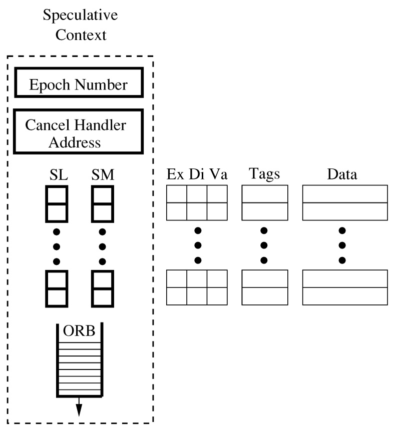
The speculation system must guarantee that there are twice as many consecutive epoch sequence numbers in the sequence number space as there are speculative contexts in the system; given this constraint, epoch numbers may

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

(a) Cache line state bits.

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

(b) State encoding.



(c) Hardware support.

Fig. 10. Encoding and implementation of cache line states. In (b), X means "don't care."

safely wrap around, and the sign of the difference between two non-equal sequence numbers indicates which epoch sequence number is *logically earlier*. Such a comparison is trivial to implement, and should only require a small amount of dedicated hardware. This implementation of sequence numbers is quite similar to the *inums* implemented in the Alpha 21464 [Emer 2001].

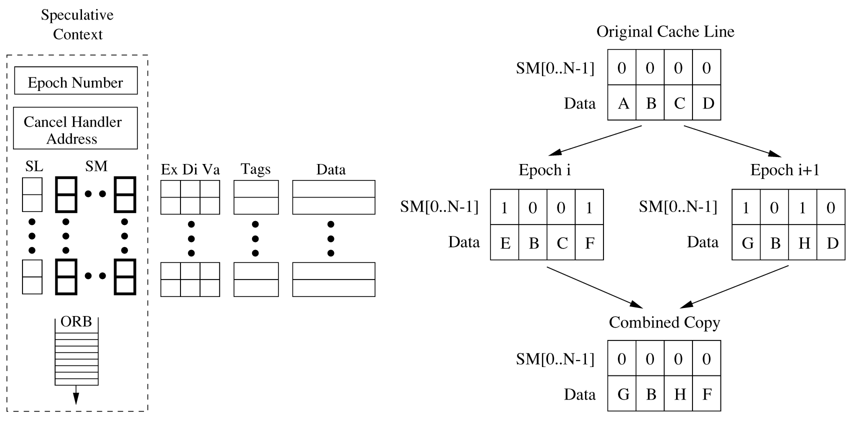
3.5.2 *Implementation of Speculative State.* We encode the speculative cache line states given in Figure 8(a) using five bits, as shown in Figure 10(a). Three bits are used to encode basic coherence state: *exclusive* (*Ex*), *dirty* (*Di*), and *valid* (*Va*). Two bits—*speculatively loaded* (*SL*) and *speculatively modified* (*SM*)—differentiate speculative from non-speculative states. Figure 10(b) shows the state encoding, which is designed to have the following two useful properties. First, when an epoch becomes *homefree*, we can transition from speculative to appropriate non-speculative states simply by resetting the *SM* and *SL* bits. Second, when a violation occurs, we want to invalidate the cache line if it has been speculatively modified; this can be accomplished by setting its *valid* (*Va*) bit to the AND of its *Va* bit with the complement of its *SM* bit ($Va = Va \wedge \overline{SM}$).

Figure 10(c) illustrates how the speculative state can be arranged. Notice that only a small number of bits are associated with each cache line, and that only one copy of an epoch number is needed per speculative context. The *SL* and *SM* bit columns are implemented such that they can be flash-reset by a single control signal. The *SM* bits are also wired appropriately to their corresponding *Va* bits such that they can be simultaneously invalidated when an epoch is squashed. Also associated with the speculative state are an ownership required buffer (ORB, described in Section 3.4.2), and the address of the cancel routine.

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

3.5.4 *Allowing Multiple Writers.* It is often advantageous to allow multiple epochs to speculatively modify the same cache line at the same time. Such support can reduce failed speculation due to false dependences, as well as allow write-after-write dependences to proceed successfully by effectively renaming memory. Similar support is implemented in related schemes that use caches as speculative buffers [Gopal et al. 1998; Cintra et al. 2000].

Supporting a multiple writer scheme requires the ability to merge a modified cache line with a previous copy of that line; this in turn requires the ability to identify partial modifications. One possibility is to replicate the *SM* column of bits so that there are as many *SM* columns as there are words (or even bytes) in a cache line, as shown in Figure 11(a). We will call these *fine-grain SM* bits. When a write occurs, the appropriate *SM* bit is set. If a write occurs that is of lesser granularity than the *SM* bits can resolve (for example a byte-write when there is only one SM bit per 4-byte word), we must conservatively set the *SL* bit for that cache line since we can no longer merge this cache line with others—setting the *SL* bit ensures that a violation is raised if a logically-earlier epoch writes the same cache line. With support for multiple writers, a speculative upgrade-request from a logically-earlier epoch no longer causes speculation to



(a) State for multiple writers. (b) Combining cache lines.

Fig. 11. Support for multiple writers.

fail when the corresponding cache line is only speculatively modified (and not speculatively loaded).

Figure 11(b) shows an example of how we can combine speculatively modified versions of a cache line with a non-speculative one. Two epochs speculatively modify the same cache line simultaneously, setting the *fine-grain SM* bit for each location modified. A speculatively modified cache line is committed by updating the current non-speculative version with only the words for which the *fine-grain SM* bits are set. In the example, both epochs have modified the first location. Since *epoch* $i + 1$ is logically-later, its value (`G`) takes precedence over *epoch i's* value (`E`).

Because dependence violations are normally tracked at a cache line granularity, another potential performance problem is *false violations*—where disjoint portions of a line were read and written. To help reduce this problem, we observe that a line only needs to be marked as *speculatively loaded* (*SL*) when an epoch reads a location that it has not previously overwritten (the load is *exposed* [Aho et al. 1986]). The *fine-grain SM* bits allow us to distinguish exposed loads, and therefore can help avoid false violations. We evaluate the performance benefits of support for multiple writers in Section 5.3.2, and show that this support has a significant impact.

4. EXPERIMENTAL FRAMEWORK

In this section we describe the compiler support, benchmark applications, and simulation infrastructure that are used throughout this article to evaluate our support for TLS.

4.1 Compiler Support

In contrast with hardware-only approaches to TLS, we rely on the compiler to define where and how to speculate. Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [Tjiang et al. 1992], which operates on `C` code, and it performs the following phases when compiling an application to exploit TLS.

4.1.1 *Deciding Where to Speculate.* One of the most important tasks in a thread-speculative system is deciding which portions of code to execute speculatively. Performing this task without the use of detailed profile information is an open research problem, although significant progress has been made [Vijaykumar 1998; Marcuello and González 2002; Zilles and Sohi 2002; Roth and Sohi 2001; Prabhu and Olukotun 2003]. For the evaluations in this article, the compiler uses profile information to decide which loops in a program to speculatively parallelize. We limit our focus to loops for two reasons: first, loops comprise a significant portion of execution time (coverage) and hence can impact overall program performance; second, loops are fairly regular and predictable, hence it is straightforward to transform loop iterations into epochs.

The following gives a basic description of the loop selection process used to compile benchmark applications. The first step is to measure every loop in every benchmark application by instrumenting the start and end of each potential speculative region (loop) and epoch (iteration). Second, we remove from consideration those loops that are unlikely to contribute to improved performance. In particular, we do not want to consider (i) loops that comprise an insignificant fraction of execution time, (ii) loops with so many instructions per iteration that cross-iteration dependences are inevitable and speculative buffering will be insufficient, and (iii) loops with so few instructions per iteration that the overheads of parallelization will overcome any benefit. Hence we filter the loops to only consider those that meet the following criteria:

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

In the third step, we unroll each loop by factors of 1 (no unrolling), 2, 4, and 8, generating several versions of each benchmark to measure. Next we measure the expected performance of each loop and unrolling when run speculatively in parallel, using detailed simulation on our baseline hardware support for TLS (see Section 5). We select loops for the purposes of evaluating our hardware support by maximizing performance: we select the loops that contribute the greatest performance gain. The best performing unrolling factor is used for each loop, and chosen independently for the sequential and speculative versions of each application.

4.1.2 *Transforming to Exploit TLS.* Once speculative regions are chosen, the compiler inserts the TLS instructions that interact with hardware to create and manage the speculative threads and forward values, as described in Section 2.2 and illustrated in Figure 5.

4.1.3 *Optimization.* Without optimization, execution can be unnecessarily serialized by synchronization (through `wait` and `signal` operations). A pathological case is a "`for`" loop in the `C` language where the loop counter is read at

Table I. Benchmark Use

Benchmark		Description	Input
SPECint2000	BZIP2_COMP	compression	input.source from ref, comp phase
	BZIP2_DECOMP	decompression	input.source from ref, decomp phase
	CRAFTY	chess board solver	ref
	GAP	group theory interpreter	ref
	GCC	compiler	expr.i from ref
	GZIP_COMP	compression	input.source from ref, comp phase
	GZIP_DECOMP	decompression	input.source from ref, decomp phase
	MCF	combinatorial optimization	ref
	PARSER	natural language parsing	ref
	PERLBMK	perl interpreter	diffmail.pl from ref
	TWOLF	place and route for standard cells	ref
	VORTEX	OO database	bendian1.raw from ref
	VPR_PLACE	place and route for FPGAs	place portion of ref input
	VPR_ROUTE	place and route for FPGAs	route portion of ref input
SPECint95	COMPRESS	compression/decompression	reduced ref input (5.6MB)
	GO	game playing, AI, plays against itself	9stone21.in from ref
	IJPEG	image processing	vigo.ppm from ref
	LI	lisp interpreter	ref
	M88KSIM	microprocessor simulator	ref

the beginning of the loop and then incremented at the end of the loop—if the loop counter is synchronized and forwarded, then the loop will be serialized. However, scheduling can be used to move the `wait` and `signal` closer to each other, thereby reducing this critical path. Our compiler schedules these critical paths by first identifying the computation chain leading to each `signal`, and then using a dataflow analysis that extends the algorithm developed by Knoop [Knoop and Ruthing 92] to schedule that code in the earliest safe location. We can do even better for any loop induction variable that is a linear function of the loop index; the scheduler hoists the associated code to the top of the epoch and computes that value locally from the loop index, altogether avoiding any extra synchronization. These optimizations have a large impact on performance [Zhaia et al. 2002, 2004].

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

4.2 Benchmarks

We evaluate our support for TLS using the SPECint95 and SPECint2000 integer benchmarks [SPEC 2000], with the exception of EON, which is written in C++ and not supported by SUIF. A brief description of each benchmark and the input data set used is given in Table I—we use the *ref* input set for every benchmark. BZIP2, GZIP, and VPR are split into two phases each, so that their executions are more representative of the entire benchmark. To maintain reasonable simulation time, we truncate the execution of all appropriate benchmarks

Table II. Benchmark Statistics

Benchmark	Portion of Dynamic Execution Parallelized (Coverage)	Number of Unique Parallelized Regions	Average Epoch Size (dynamic insts)	Average Number of Epochs Per Dynamic Region Instance
BZIP2_COMP	28%	14	268.6	230587.3
BZIP2_DECOMP	13%	3	176.4	693495.1
CRAFTY	9%	7	1022.0	11.6
GCC	11%	58	700.7	60.9
GO	17%	26	1249.1	16.6
IJPEG	84%	12	449.0	152.4
LI	8%	2	5994.2	1.4
M88KSIM	57%	5	424.1	85.6
MCF	97%	4	125.5	5328.2
PARSER	7%	23	312.6	17205.2
PERLBMK	10%	5	971.0	165.9
VORTEX	4%	5	6543.7	8.7
VPR_PLACE	76%	8	340.7	4.3
average	32%	13	1429.0	72855.7

by fast-forwarding the initialization portion of execution and simulating up to the first billion instructions, beginning simulation with a "warmed-up" memory system loaded from a presaved snapshot. Since the sequential and TLS versions of each benchmark are compiled differently, the compiler instruments them to ensure that they terminate at the same point in their executions relative to the source code so that the executions are comparable.

Table II shows an analysis of the benchmarks studied. For COMPRESS, GAP, GZIP, TWOLF, and VPR_ROUTE, the region selection algorithm has opted to select no regions at all. We do not claim that there is no speculative parallelism available in these applications, but that they require more advanced compiler and/or support for TLS than we investigate in this article—for example, applying more advanced techniques for improving value communication between speculative threads can increase the amount of available speculative parallelism [Moshovos et al. 1997; Steffan et al. 2002; Zhaia et al. 2002, 2004; Cintra and Torrellas 2002].

For the remaining integer benchmarks, coverage (the portion of dynamic execution of the sequential version that is speculatively parallelized) ranges from 4% to 97%, and averages 32%. For some applications, good coverage is achieved by selecting several significant loops, such as for IJPEG which has a coverage of 84% through 12 different selected loops. In contrast, MCF has a coverage of 97% through only 4 unique loops. The SPECint2000 version of GCC has 58 loops selected, but a coverage of only 11.0%.

Epoch size is another important characteristic. If epochs are too small, the overheads of speculative parallelization will be overwhelming. If epochs are too large, then they will likely be dependent, or we may not have enough buffer space to hold all of the speculative state. Table II indicates a wide range of average epoch sizes for the selected regions, from just 125.5 dynamic instructions in

Table III. Simulation Parameters

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

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

MCF to over six thousand dynamic instructions in VORTEX. Over all benchmarks, the average number of dynamic instructions per epoch is 1429, which is quite large. However, the average number of epochs per dynamic region instance can be quite small for some benchmarks: for example, 1.4 on average for LI. On average, only two processors will be busy when executing speculatively for LI, which will limit the overall speedup for this benchmark.

4.3 Simulation Model

We evaluate our approach using a detailed model—built upon the `MINT+` [Veenstra 2000] MIPS emulator—that simulates 4-way issue, out-of-order, superscalar processors similar to the MIPS R10000 [Yeager 1996] but with more modern structure sizes (a 128 entry reorder buffer, larger caches and so on). Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table III.

5. EVALUATION OF PRIVATE-CACHE SUPPORT

We now present the performance of our coherence scheme for TLS, and evaluate the overheads of our approach. In this section we focus on chip-multiprocessor architectures where each processor has a private first-level data cache—later (in Section 6) we evaluate architectures that share the first-level data cache.

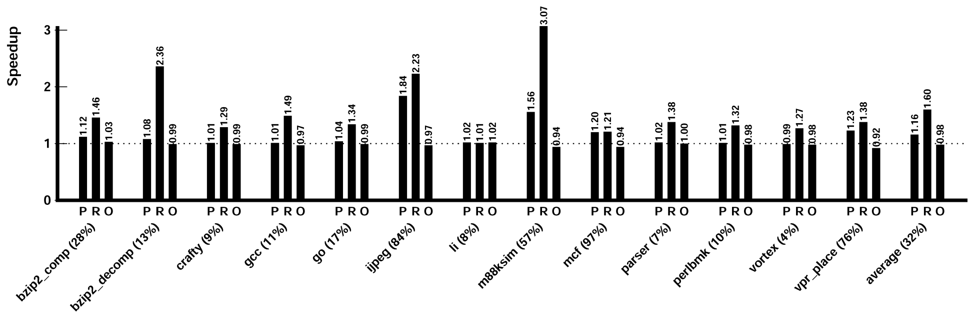


Fig. 12. Overall performance of our approach to TLS on four processors. *P* is program speedup, *R* is region speedup, and *O* is outside-region speedup. The percent coverage is shown in parentheses next to each benchmark name.

5.1 Performance of the Baseline Coherence Scheme

Figure 12 summarizes the performance of each application on our baseline architecture (as reported by the detailed simulation model described in Section 4.3), which is a four-processor chip-multiprocessor that implements our coherence scheme. Throughout this article, all speedups (and other statistics relative to a single processor) are with respect to the *original* executable (without any TLS instructions or overheads) running on a single processor. Hence our speedups are *absolute speedups* and not self-relative speedups.

As we see in Figure 12, we achieve speedups on the speculatively-parallelized regions of code (*R*) ranging from 1% to 207%, and averaging 60%. We also report the speedup for the portion of execution that has not been parallelized (outside-region speedup, *O*), which for the most part indicates that performance is reduced for the TLS versions: this portion of execution has an average 2% slowdown across all applications. This slowdown is the result of hampered compiler optimization (due to our inserted TLS instructions and code transformations) and decreased data-cache locality (due to the spreading of cache lines to multiple caches during speculative region execution). In addition to the slight slowdown of the non-parallelized portions, the overall program speedups (*P*) are also limited by the *coverage* (the fraction of the original execution time that was parallelized) which ranges from 4% to 97%, and averages 32%.

Looking at program performance (*P*), IJPEG and M88KSIM are 84% and 56% faster respectively, and four other applications improve by at least 8%. Six other applications, show more modest improvement, while VORTEX performs slightly worse than the original sequential version. On average across all applications, we achieve a program speedup of 16%. To simplify our evaluation of hardware support for TLS, throughout the remainder of this section, we will focus solely on the regions of code that have been speculatively parallelized.

Figure 13 shows region execution time normalized to that of the original sequential version; each of the bars are broken down into eight segments explaining what happened during all potential graduation slots.[6] The top segment,

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

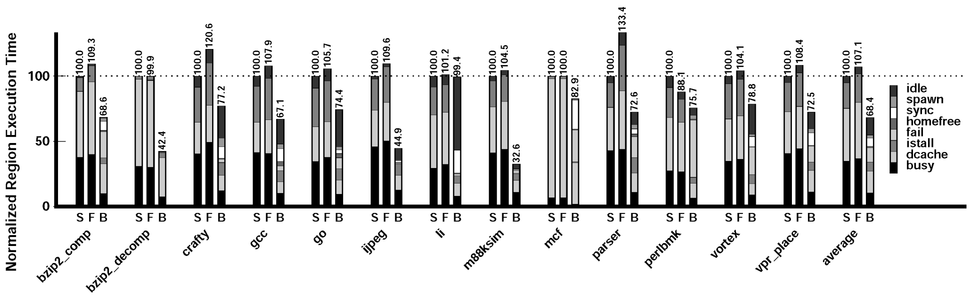


Fig. 13. Impact on region execution time of our baseline hardware support for TLS. *S* is the original sequential version, *F* is the speculative version run sequentially, and *B* is the speculative version run in parallel on four processors.

idle, represents slots where the pipeline has nothing to execute—this could be due to either instruction-fetch latency, or simply a lack of work. The next three segments represent slots where instructions do not graduate for the following TLS-related reasons: waiting to begin a new epoch (*spawn*); waiting for synchronization for a forwarded value (*sync*); and waiting for the homefree token to arrive (*homefree*). The *fail* segment represents all slots wasted on failed speculation, including slots where misspeculated instructions graduated. The remaining segments represent regular execution: the *busy* segment is the number of slots where instructions graduate; the *dcache* segment is the number of non-graduating slots attributed to data cache misses; and the *istall* segment is all other slots where instructions do not graduate.

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

The third bar (*B*) shows the TLS version executed speculatively in parallel on four processors. Some benchmarks show a large increase in *idle* time—for the most part this is caused by a small number of epochs per region instance, resulting in idle processors. For example, LI has an average of only 1.4 epochs per region instance (see Table II), which is not enough parallelism to completely occupy 4 processors. To some extent this is an artifact of our simulation, since

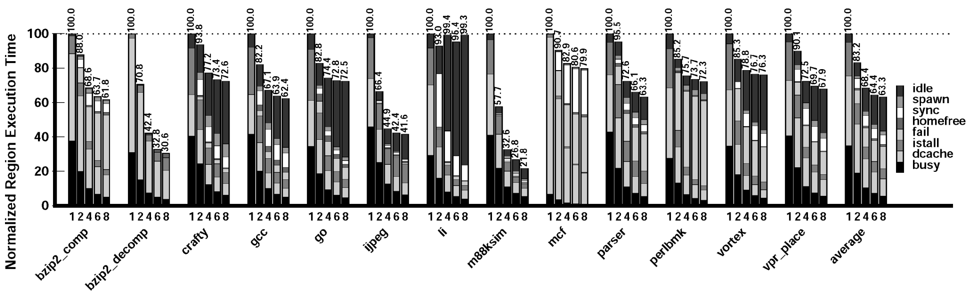


Fig. 14. Varying the number of processors (1, 2, 4, 6, 8). The baseline architecture has four processors.

a more sophisticated TLS system could increase the number of speculative threads by allowing speculation beyond the end of a loop (the loop *continuation* could be executed speculatively in parallel with the iterations of the loop). It is important to note that the failed speculation component (*fail*) includes data cache misses, pipeline stalls, synchronization, and so on, for all cycles spent on failed speculation—hence when a bottleneck such as synchronization is reduced, the failed speculation component may be reduced as well.

At the beginning and end of each epoch we measure the time spent waiting for an epoch to be spawned (*spawn*), and the time spent waiting for the *homefree* token. Spawn time is not a significant bottleneck for any benchmark, but is most evident in CRAFTY. Passing the homefree token (*homefree*) is an insignificant portion of execution time for every benchmark, indicating that in general this aspect of our scheme is not a performance bottleneck.

5.1.1 *Scaling Within a Chip.* Figure 14 shows how performance varies across a number of different processors. As we increase the number of processors, performance continues to improve for every case but LI, which has a limited amount of available parallelism (hence the dramatic increase in `idle` time for that application). However, the speedup achieved is not linear with the number of processors—the benefits of additional processors beyond four is modest. For BZIP2_COMP, MCF, PARSER, and PERLBMK, the amount of failed speculation increases with the number of processors, indicating a limit to the independence of epochs for these applications. The amount of time spent waiting for the *homefree* token remains negligible for all applications.

5.2 Overheads of the Baseline Coherence Scheme

We now investigate the overheads of our baseline scheme in greater detail. The most significant overheads are from flushing the ORB (described in Section 3.4.2), failed speculation, and decreased cache locality.

5.2.1 *Ownership Required Buffer (ORB).* Recall that the ORB maintains a list of addresses of speculatively-modified cache lines that are in the *speculative-shared* (*SpS*) state. When the *homefree token* arrives, we must issue and complete upgrade requests to obtain exclusive ownership of these lines (thereby committing their results to memory) prior to passing the homefree token to the

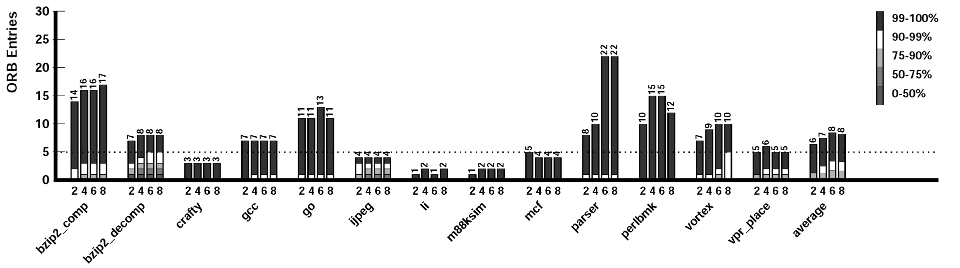


Fig. 15. Evaluating the size of the *ownership required buffer* (ORB) per epoch as we vary the number of processors (2, 4, 6, 8). We show the size of the ORB required to capture all ORB entries for 50%, 75%, 90%, 99%, and 100% of all epochs.

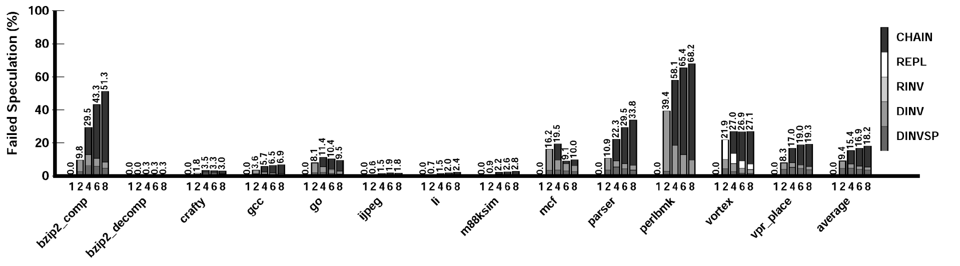


Fig. 16. Percentage of execution time wasted on failed speculation, and the breakdown of reasons for violations as we vary the number of processors (1, 2, 4, 6, 8).

next *logically-later* epoch. In addition, speculation fails if the ORB overflows. For these reasons, we desire the average number of required ORB entries per epoch to be small.

Figure 15 shows the resulting sizes of the ORB for a varying number of processors. First we see that averaging across all applications, 99% of all epochs require three ORB entries or less. We also note that an ORB of five entries would be sufficient for every application 99% of the time. An ORB with eight entries is sufficient for the average application 100% of the time; however an ORB with 22 entires is needed for PARSER to work 100% of the time, which is probably too large. For BZIP2_COMP, PARSER, and VORTEX, the maximum number of ORB entries per epoch clearly increases as the number of processors increases, indicating that there are shared cache lines for which the number of sharers increases with the number of processors. From these results we can conclude that an ORB of 5 entries is sufficient for all benchmarks.

5.2.2 *Failed Speculation.* Figure 16 shows the percentage of execution for each benchmark lost to failed speculation for a varying number of processors. Each bar is broken down, showing the fraction of speculation that failed for each of five reasons. The first segment, *CHAIN*, represents time spent on epochs that were squashed because logically-earlier epochs were previously squashed. This *violation chaining* prevents an epoch from using potentially incorrect data that was forwarded from a logically-earlier epoch. The next two segments (*REPL* and *RINV*) represent violations caused by replacement in either the first-level data caches or the shared unified cache, respectively. The last two segments

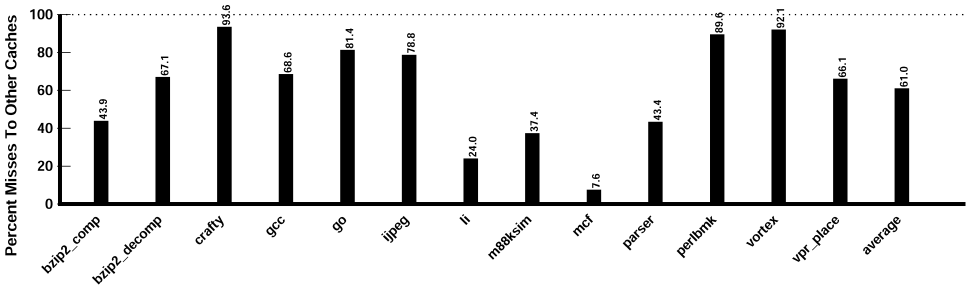


Fig. 17. Percentage of misses where the cache line is resident in another first-level cache, which indicates the impact of TLS execution on cache locality.

(*DINV* and *DINVSP*) represent violations caused by true data dependences. A *DINV* violation occurs when an epoch commits and flushes its ORB, generates a read-exclusive request, and invalidates a speculative cache line belonging to a logically-later epoch. A *DINVSP* violation is caused by a *speculative* invalidation, which is sent before an epoch commits.

BZIP2_DECOMP, CRAFTY, IJPEG, LI, and M88KSIM all have an insignificant amount of failed speculation, and for GCC and GO, the amount of failed speculation is quite small. For the other benchmarks with greater amounts of failed speculation, the fraction that is due to *CHAIN* violations increases with the number of processors; for a given violated epoch, the number of *CHAIN* violations is equal to the number of logically-later epochs currently in-flight. The portion of failed speculation due to *CHAIN* violations is usually greater than the portion due to other reasons, except for the two-processor cases, where at any given time only one epoch is speculative and can be violated (and the other is always non-speculative). Replacement in the unified cache (*RINV*) is not significant for any benchmark (since the data sets for these applications fit well within the 2MB unified secondary cache); failed speculation due to replacement from the first-level data caches (*REPL*) is evident for VORTEX. Speculative invalidations (*DINVSP*) are preferable over ORB-generated invalidations (*DINV*) because they give earlier notification of violations and also help reduce the size of the ORB itself. For three applications, *DINVSP* violations are roughly as frequent as *DINV* violations, while four other applications are dominated by *DINV* violations.

5.2.3 *Data-Cache Locality.* Finally, we estimate the impact of speculative parallelization on data cache locality by measuring the fraction of cache misses where the cache line in question is currently resident in another first-level cache, as shown in Figure 17. A high percentage indicates that cache locality for the corresponding application has been decreased. We see that this percentage is quite high for most applications, the average being 61.0%. This loss of locality, while not prohibitive, is an opportunity for improvement—possibly through prefetching or other techniques for dealing with distributed data access.

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

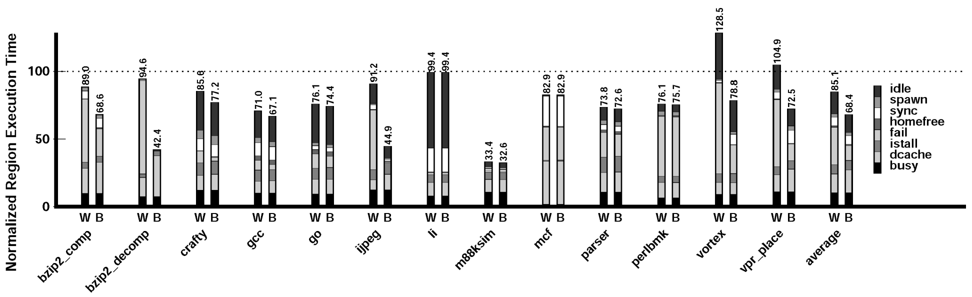


Fig. 18. Impact of support for multiple writers—*W* is without, while *B* (our baseline coherence scheme) includes, support for multiple writers.

5.3 Tuning the Coherence Scheme

Our performance analysis has shown that TLS is promising, that our scheme for extending cache coherence to track data dependences and buffer speculative state is efficient and effective, and that within a single chip it scales to extract the speculative parallelism available in applications. In this section we briefly evaluate the performance of two implementation alternatives: support for speculative invalidation of non-speculative cache lines, and for multiple writers.

5.3.1 *Speculative Invalidation of Non-Speculative Cache Lines.* As discussed earlier in Section 3.4.1, another subtle design choice is whether a speculative invalidation should invalidate a cache line in a non-speculative state. Recall that our baseline scheme does speculatively invalidate non-speculative cache lines. Previous work [Steffan 2003] shows that allowing speculative invalidation of non-speculative cache lines improves performance by only 0.6% on average (across all applications), indicating that this is not a crucial design decision [Steffan 2003].

5.3.2 *Support for Multiple Writers.* Recall from Section 3.5.4 that multiple writer support allows us to avoid both violations due to write-after-write dependences as well as violations due to false dependences where speculative loads are not *exposed*. Figure 18 compares the performance of our baseline hardware (*B*), which does support multiple writers, with that of less complex hardware (*W*), which does not support multiple writers. For five applications, support for multiple writers greatly reduces the amount of failed speculation; VORTEX and VPR_PLACE require this support to speed up at all. On average across all benchmarks, support for multiple writers improves the performance of our coherence scheme by 24%, hence we include it in our design.

5.4 Sensitivity to Architectural Parameters

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

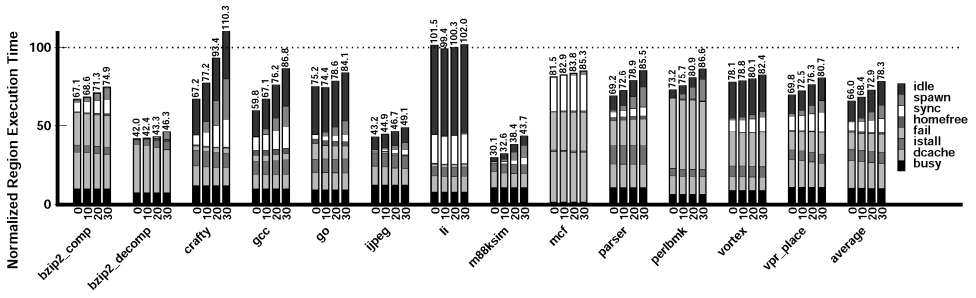


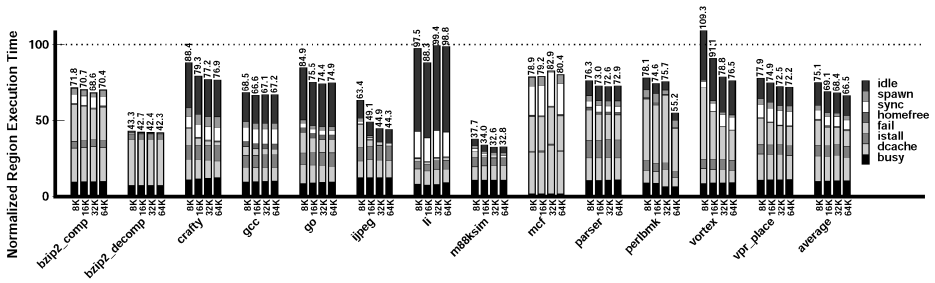
Fig. 19. Impact of varying communication latency (0, 10, 20, and 30 cycles).

complexity of several architectural features including inter-processor communication mechanisms and the memory system. Further evaluations are available in Steffan's dissertation [Steffan 2003].

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

In a previous study we investigated the potential impact of improving each of these latencies independently [Steffan 2003]. When spawn latency is zero, the remaining *spawn* segment for nearly all applications is negligible, indicating that spawn latency is not a bottleneck. When the forwarding latency is set to zero, the *sync* portion is reduced for most applications. However, this component of execution time is not removed completely, and in some cases it is only slightly reduced—this indicates that the actual communication of signal messages (once they are ready) is not a bottleneck. When the latency of sending the homefree token is set to zero, performance improves only slightly in most cases. These results confirm that the communication latency for passing the homefree token is not a bottleneck.

An interprocessor communication latency of 10 cycles is itself quite fast. Can speedups be achieved with larger communication latencies? Figure 19 shows the impact of varying all three communication latencies simultaneously from zero to thirty cycles. About half of the applications are very sensitive to the


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

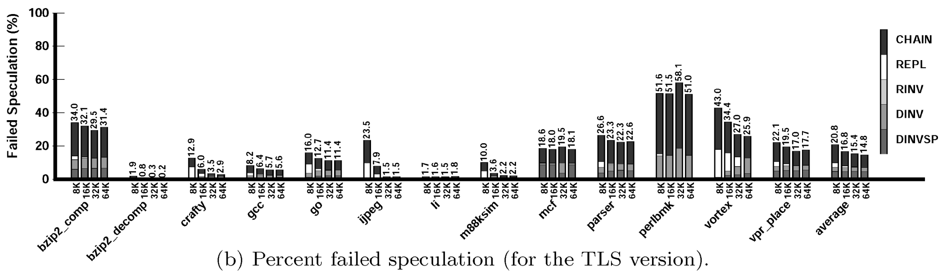

(b) Percent failed speculation (for the TLS version).

Fig. 20. Varying data cache size from 8kB to 64kB. Note that our baseline architecture has a 32kB data cache per processor.

communication latencies. *Spawn* time is the component that increases the most: since it is the first latency to occur for each epoch, it receives a majority of the blame. The increase in the *sync* segment is less pronounced, while the *home-free* segment remains minimal. Only CRAFTY and LI no longer speed up when communication latency is 30 cycles, although the trends indicate that latencies of 50–60 cycles would eliminate most of the benefits of TLS. Hence it is definitely important to minimize the communication latency between processors, although most benchmarks can withstand greater latency than we assume and still achieve decent speedups under TLS.

5.4.2 *Data Cache Size.* Our scheme for TLS uses the caches and coherence scheme to implement data dependence tracking and buffering speculative state, and hence requires an efficient underlying memory system. In this section we investigate the sensitivity of TLS execution to the size of the first-level data caches.

In Figure 20 we vary the size of the data caches from 8kB to 64kB—our baseline architecture has a 32kB data cache per processor. This experiment has a significant impact on both the sequential and TLS versions of the applications, with the TLS versions benefiting more than the sequential versions from larger caches. As is evident in Figure 20(b), larger caches also reduce the amount of failed speculation due to replacement (*REPL*). For the 8kB caches, CRAFTY, GO, IJPEG, M88KSIM, and VORTEX all suffer a significant amount of failed speculation due to replacement, indicating that 8kB caches alone are insufficient for implementing our scheme. Performance continues to improve as we increase cache size, although 32kB caches nearly eliminate failed speculation due to replacement.

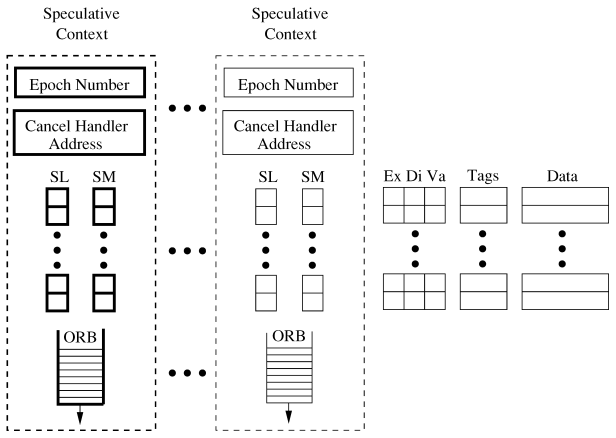


Fig. 21. Hardware support for multiple epoch contexts in a single cache.

6. SHARED-CACHE SUPPORT FOR TLS

Until this point we have focused on the implementation and performance of TLS on a chip-multiprocessor where each processor has its own private data cache. In this section we describe and evaluate support for multiple speculative contexts within a single cache, which is important for three reasons. First, support for multiple speculative contexts allows us to implement TLS with *simultaneous multithreading* (SMT) [Tullsen et al. 1995] and other shared-cache multithreaded architectures. Second, we can use multiple speculative contexts to allow a single processor to switch to a new epoch when the current epoch is suspended (e.g., when waiting for the homefree token). Finally, we may want to maintain speculative state across OS-level context switches so that we can support TLS in a multiprogramming environment.[7]

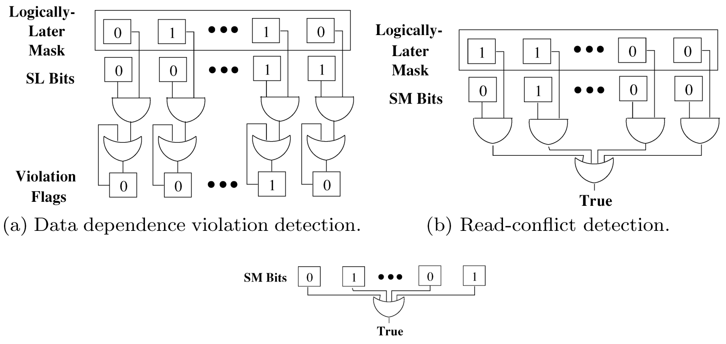
We begin by describing how our implementation of speculative state from Section 3.5 can be extended to support multiple speculative contexts. We next evaluate this support and then explore ways to avoid failed speculation due to conflicts in the shared cache.

6.1 Implementation

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

Figure 21 shows how we can support TLS in a shared cache by implementing multiple speculative contexts. The exclusive (*Ex*), dirty (*Di*), and valid (*Va*) bits for each cache line are shared between all speculative contexts, but each

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



(a) Data dependence violation detection. (b) Read-conflict detection.

(c) Write-conflict detection.

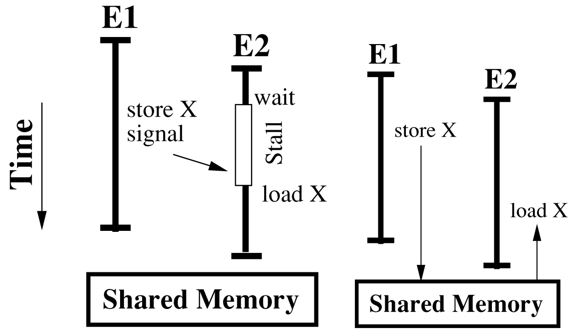
Fig. 22. Support for efficient epoch number comparison.

speculative context has its own speculatively-loaded (*SL*) and speculatively-modified (*SM*) bits. This state allows us to track which epochs have speculatively loaded or modified any cache line, and allows us to track data dependences between epochs as well as check for conflicts. As we add more state to the cache, it is of course important to consider layout factors and to ensure that the basic performance of the cache is not adversely affected. For example, this will limit the number of speculative contexts that can be supported in a single cache.

Since epoch contexts that share a cache are implemented in a common structure (as opposed to the distributed implementation for private caches), it is wasteful to frequently recompute their relative ordering by comparing epoch numbers on every memory reference. Instead, we can *precompute* and store the relative ordering between all active epochs that share the cache. A convenient method of storing this information is in a *logically-later mask*. Each speculative context maintains its own mask, and within each mask there is a bit per speculative context. For the logically-later mask owned by a context executing epoch `i`, each bit of the mask is set if the corresponding speculative context is currently executing an epoch that is *logically-later* than epoch `i`. Hence this mask must be updated whenever an epoch is spawned or completes.

As shown in Figure 22(a), we can use the *logically-later mask* to detect data dependence violations. If the active epoch stores to a location, then any logically-later epoch that has already speculatively loaded that same location has committed a violation. We can detect a violation by taking the bit-wise `AND` of the logically-later mask with the *SL* bits for the appropriate cache line and `OR`'ing the result with the violation flag for each epoch.

In addition to detection of data dependence violations, we also need to track read and write conflicts in the shared cache. A conflict occurs when two epochs access the same cache line in an incompatible way, and is resolved by squashing the logically-later epoch (a *conflict violation*). For our initial shared-cache implementation, the following two access patterns are incompatible (these are formalized in the description of our speculative coherence scheme [Steffan et al. 1997]).



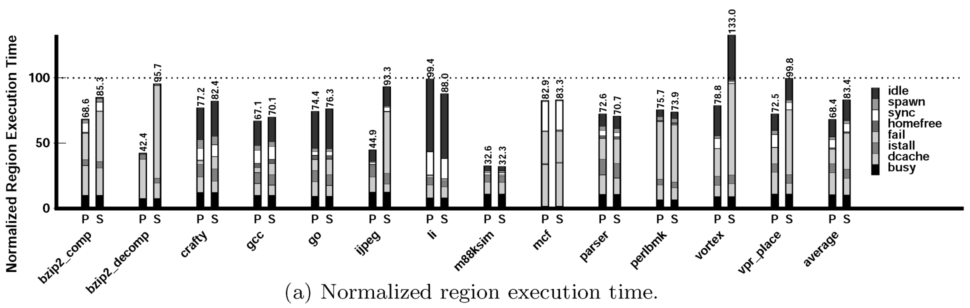
(a) Explicit forwarding. (b) Implicit forwarding.

Fig. 23. Explicit vs implicit forwarding.

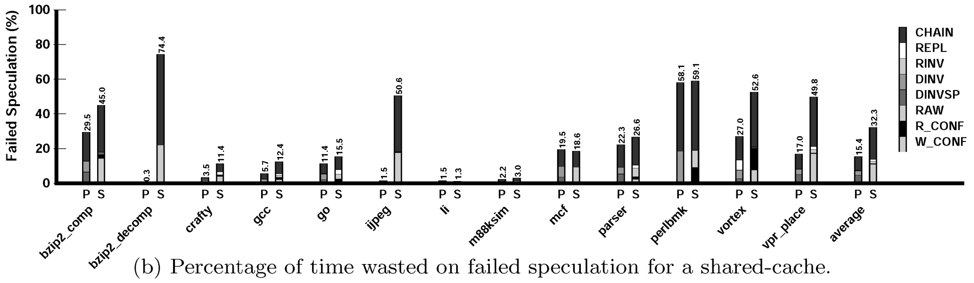
(1) If an epoch speculatively modifies a cache line, only that epoch or a logically-later epoch may read that cache line afterwards. If a logically-earlier epoch attempts to read the cache line, a *read-conflict violation* results.
(2) Only one epoch may speculatively modify a given cache line. If an epoch attempts to speculatively modify a cache line that has already been speculatively modified by a different epoch, a *write-conflict* violation results.

We can use the logically-later masks to determine whether any load or store will result in a read or write-conflict violation, as illustrated in Figures 22(b) and 22(c). Recall that a read-conflict miss occurs when the active epoch attempts to execute a speculative load but a logically-later epoch has already modified that same cache line. This condition may be checked by taking the AND of the logically-later mask with the *speculatively-modified* (*SM*) bits for the appropriate cache line and checking the OR of the results. A write-conflict miss occurs when the active epoch executes a speculative store and any other epoch has already speculatively stored that cache line. We can check for this case by taking the OR of the speculatively-modified (*SM*) bits (excluding the bit belonging to the epoch in question) as shown in Figure 22(b), and proceed by squashing the logically-later epoch.

The data dependence tracking implemented by our shared-cache support for TLS differs from the private-cache support described in Section 3.5 in the following two ways. First, our shared-cache support allows us to *implicitly forward* speculative modifications between two properly ordered epochs. In Figure 23, we differentiate between explicit and implicit forwarding. With explicit forwarding, as is supported in our private-cache scheme, the compiler inserts explicit `wait` and `signal` primitives, which communicate a value between epochs through the *forwarding frame*. In contrast, *implicit forwarding* (which is not supported in our private-cache scheme) allows a value to be communicated between a store from one epoch and a load from a logically-later epoch that happen to execute in order. Our private-cache design supports only explicit forwarding because its distributed nature makes implicit forwarding extremely



(a) Normalized region execution time.



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

Fig. 24. Region performance on both private-cache and shared-cache architectures. *P* is speculatively executed on a 4-processor CMP with private caches, and *S* is speculatively executed on a 4-processor CMP with a shared cache.

difficult to implement.[8] However, implicit forwarding is trivial to support in a shared cache: we simply allow an epoch to speculatively load from a cache line that has been speculatively-modified by a logically-earlier epoch. Note that if the logically-earlier epoch then speculatively modifies that cache line again, a write-conflict violation will result. Since support for implicit forwarding is trivial to implement in a shared cache, we include implicit forwarding in our baseline design.

The second major difference between shared and private-cache architectures is that we cannot easily allow two epochs to modify the same cache line—in this respect, our basic shared-cache design is less aggressive than our private-cache design, which has such support for *multiple writers* (see Section 5.3.2). We will describe how our baseline shared-cache design can be extended to support multiple writers in Section 6.3.

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

6.2 Evaluation of Shared-Cache Support

We begin our evaluation by comparing the performance of both private-cache and shared-cache support for TLS, as shown in Figure 24(a). *P* shows speculative execution on a 4-processor CMP with private caches, and *S* shows

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

speculative execution on a 4-processor CMP with a shared first-level data cache. To facilitate comparison, the shared cache is the same size and associativity as one of the private caches (32kB, 2-way set associative). For 7 of the 13 applications, the performance with a shared cache is similar to the performance with private caches, while for LI, performance with a shared cache is somewhat improved; the remaining 6 applications (BZIP2_COMP, BZIP2_DECOMP, IJPEG, VORTEX, and VPR_PLACE) perform significantly worse with a shared cache due to increased failed speculation. On average, the shared-cache implementation performs 22% worse than the private-cache version, despite the potential benefits of increased data and instruction cache locality.

Figure 24(b) shows the percentage of time wasted on failed speculation for each application, broken down into the reasons why speculation failed. The first three segments are common between private and shared cache architectures: *CHAIN* violations represent time spent on epochs that were squashed because a logically-earlier epoch was previously squashed, while (*REPL* and *RINV*) represent violations caused by replacement in either the first-level data caches or the shared unified cache, respectively.[9] The next two segments (*DINV* and *DINVSP*) are for private-cache architectures only. Recall that a *DINV* violation occurs when an epoch commits and flushes its ORB, generates a read-exclusive request, and invalidates a speculative cache line belonging to a logically-later epoch, while a *DINVSP* violation is caused by a *speculative* invalidation, which is sent before an epoch commits.

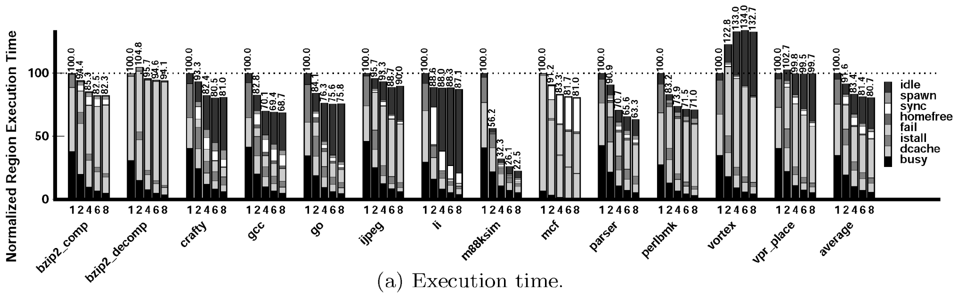
The remaining three segments represent violations raised by the new shared-cache TLS mechanisms: the *RAW* segment represents read-after-write data dependence violations, and the *R_CONF* and *W_CONF* segments represent read and write conflicts respectively. It is apparent that write conflicts (*W_CONF*) account for the vast majority of the failed speculation for the shared-cache TLS support. Furthermore, even though the set-associative first-level data cache is only 2-way set-associative, it is encouraging that CRAFTY and GO are the only two applications for which replacement (*REPL*) is a significant component of time lost to failed speculation.

For half the applications, our baseline shared-cache hardware support is sufficient to maintain the performance of private caches; however, for the remaining applications the impact of conflict violations is severe. We will investigate ways to tolerate these conflicts later in Section 6.3.

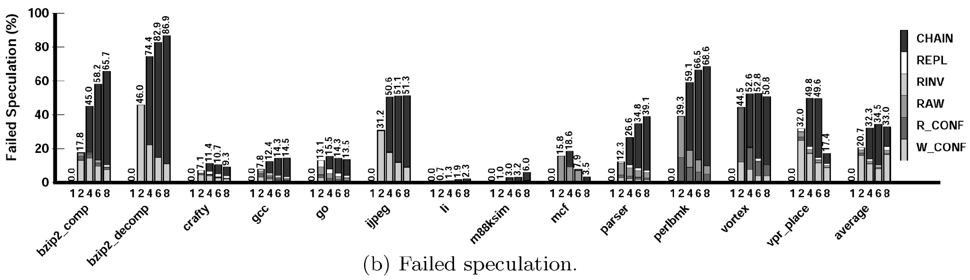
6.2.1 *Scaling Within a Chip.*

Next we examine how a shared-cache architecture can scale (within a chip) by varying the number of processors from 2 to 8, as shown in Figure 25. The observed scaling behavior is similar to that of the private cache architecture (evaluated in Section 5.1), except for BZIP2_COMP, BZIP_DECOMP, IJPEG, VORTEX, and VPR_PLACE—for these applications failed speculation prevents scaling for the shared-cache architecture. To investigate further, Figure 25(b) shows the percentage of time wasted on failed speculation for the

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



(a) Execution time.



(b) Failed speculation.

Fig. 25. Varying the number of processors for the shared-cache architecture (1, 2, 4, 6, 8).

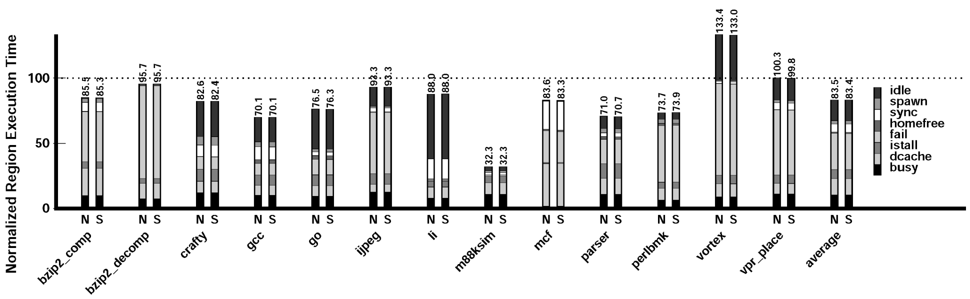


Fig. 26. Benefits of implicit forwarding in a shared cache. *N* does not support implicit forwarding, while *S* (our shared-cache baseline) does.

shared-cache design. Again, we observe that write conflicts (*W_CONF*) are the main cause of failed speculation for most applications, except for PERLBMK and VORTEX, which suffer from read conflicts as well (*R_CONF*)—we will further investigate conflicts in Section 6.3.

6.2.2 *Impact of Implicit Forwarding.* For shared-cache designs, providing support for implicit forwarding is relatively straightforward since the speculative state is implemented in a common structure (as opposed to the distributed implementation for private caches). However, it is interesting to quantify the benefits of such support. In the shared-cache experiments in Figure 26, the *N* experiment does not include support for implicit forwarding while the *S* experiment (our shared-cache baseline) does. It is apparent that support for implicit forwarding does not have a significant impact on performance other than for MCF, which improves slightly due to a decrease in failed speculation. The main

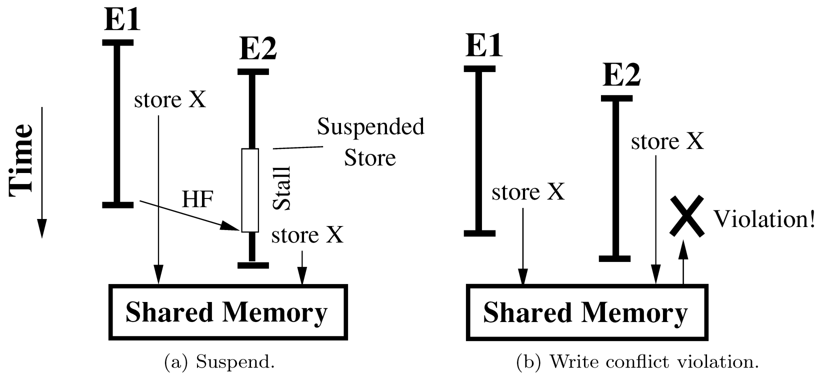


Fig. 27. Two epochs that store the same cache line. In (a), suspension of epoch 2 allows it to proceed later. In (b), suspension cannot help, and epoch 2 is violated due to the write conflict.

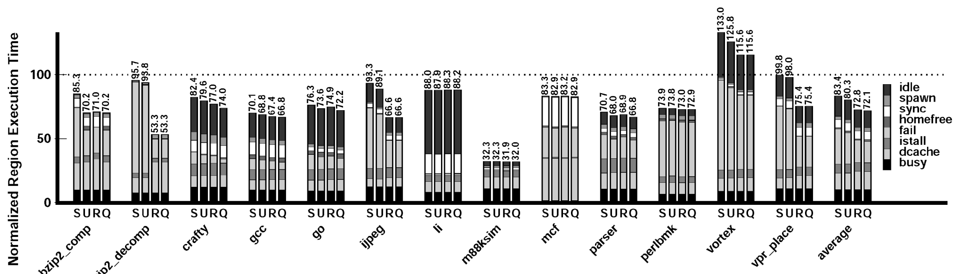
reason that implicit forwarding does not have a large performance impact is because the compiler has already explicitly synchronized local scalars. Also, since our speculative regions were selected for having good performance on hardware without support for implicit forwarding, they may be biased to not require such support. However, our results show that decent performance benefits can be obtained without such support.

We have shown that support for TLS in a shared data cache architecture is straightforward to implement, and that performance for most applications is comparable to that of a private-cache architecture. However, we also observed that an increase in failed speculation due to read and write conflicts can negate the potential performance improvement from the increased locality of the shared cache architecture (as compared with the private-cache architecture); hence we next investigate ways to tolerate these conflicts.

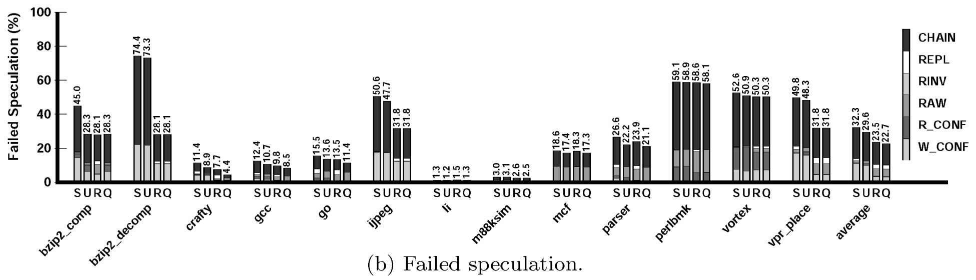
6.3 Tolerating Read and Write Conflicts

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

6.3.1 *Suspending Epochs.* Rather than handling read or write conflicts by squashing the logically-later epoch, we can instead *suspend* that epoch until it becomes homefree. Only in certain cases can an epoch be suspended. For example, consider two epochs that both attempt to write to the same cache line, as shown in Figure 27. In Figure 27(a) *epoch 1* (E1) writes first, then *epoch 2* (E2) is suspended when it attempts to write until it is passed the homefree token, at which point it can proceed. In contrast, in Figure 27(b) *epoch 2* writes first, and when *epoch 1* writes, a write conflict is triggered and *epoch 2* is squashed. There are two requirements to avoid deadlock when suspending an



(a) Execution time.



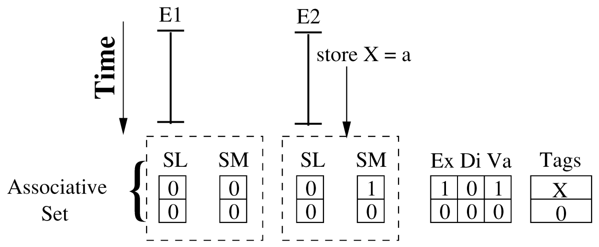
(b) Failed speculation.

Fig. 28. Impact of suspending violations for replacement and conflicts. *S* is the baseline 4-processor shared-cache architecture, *U* builds on *S* by tolerating conflicts and replacement through suspension of the logically-later epoch, *R* builds on *S* by tolerating conflicts through replication, and *Q* supports both suspension and replication.

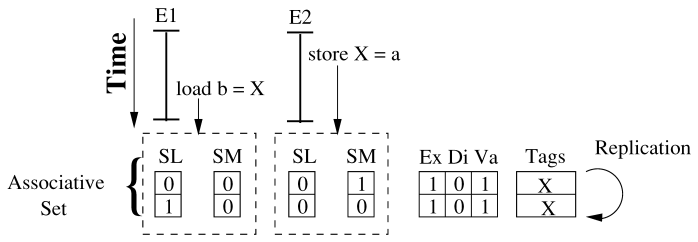
epoch: (i) that exactly one epoch is always homefree, and (ii) any suspended epoch that receives the homefree token is unsuspended at that point.

In Figure 28, we evaluate performance when both replacement and certain read/write conflicts cause the logically-later epoch to be suspended rather than squashed. We show performance on our shared-cache baseline architecture (*S*), and an augmented baseline where epochs are suspended rather than squashed whenever possible (*U*). Suspension eliminates a significant amount of failed speculation due to replacement and conflicts for several applications, including BZIP2_COMP, IJPEG, PARSER, and VORTEX. Overall, this support is worthwhile and also straightforward to implement, and provides a 3.7% performance improvement. However, it only eliminates some of the problem of read/write conflicts.

6.3.2 *Cache Line Replication.* Another technique for tolerating read and write conflicts is cache line replication. Rather than squashing the conflicting epoch, the epoch can proceed by replicating the appropriate cache line: if the cache line is not yet speculatively-modified, then it may be copied directly; if the cache line is speculatively-modified, then the replicated copy is obtained from the external memory system. Once replicated, both copies of the cache line are kept in the same associative set of the shared cache. The owner of a given cache line can be determined by checking the *SM* and *SL* bits—in other words, the *SM* and *SL* bits are considered part of the tag match. If all entries in an associative set are consumed, then replication fails and the logically-latest epoch owning a cache line in that set is suspended or squashed.



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



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

Fig. 29. Example of cache line replication.

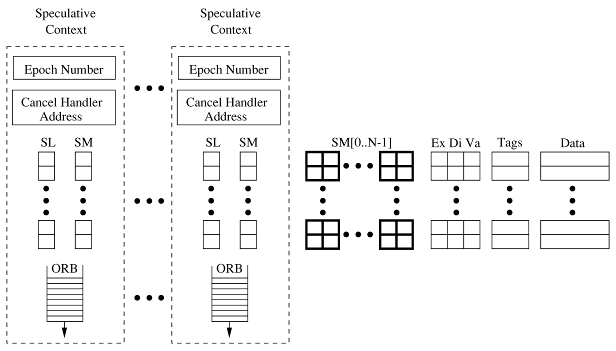


Fig. 30. Hardware support for multiple writers in a shared cache.

Figure 29 shows an example of cache line replication. In the figure, only the speculative state of the appropriate associative set is shown. In Figure 29(a), *epoch 2* (E2) executes a speculative store to location X, and the *SM* bit for that cache line is set. Next, in Figure 29(b), *epoch 1* executes a speculative load from the same location (X), resulting in a read conflict (since *epoch 1* must not read the speculative modifications of *epoch 2*). Rather than squashing *epoch 2*, we can use the available entry in the associative set to store a replicated copy of location X. Once replication is supported, we can also extend our implementation to support multiple writers by adding *fine-grain SM* bits [Steffan et al. 2000]—as shown in Figure 30, only one group of fine-grain SM bits is necessary per cache line since only one epoch may modify a given cache line.

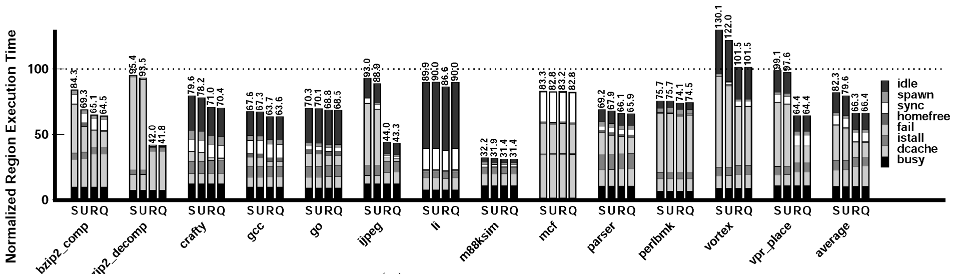
For replication to function correctly, two other issues must be dealt with. First, when an epoch performs a store, the store value must be propagated to replicated copies that belong to logically-later epochs—this prevents potentially expensive merging operations at commit time. Second, when an epoch commits, any cache line that it has speculatively modified transitions to the *dirty* state; if such a cache line was itself a replica, then the original cache line must be invalidated since it has been made obsolete by the dirty cache line. One efficient way to track such cache lines would be to keep a list of their cache tags, in a manner similar to the ORB mechanism (described in Section 3.4.2).

In Figure 28, the *R* experiment builds on the baseline *S* with support for cache line replication. This support has a significant positive impact on the performance of BZIP2_COMP, IJPEG, VORTEX, and VPR_PLACE. On average, replication provides a 12.7% improvement over basic shared-cache support, and a 9.3% improvement over suspension. In all of these cases the amount of failed speculation has been significantly reduced. From Figure 28(b), which shows the percentage of execution time wasted on failed speculation and the corresponding breakdown, we see that replication does increase tolerance of write conflicts (*W_CONF*) and read conflicts (*R_CONF*). Reducing the occurrence of write conflicts merely exposes more true data dependence violations (*RAW*) for VPR_PLACE. Although this support is relatively costly to implement (since we need to be able to store multiple versions of the same cache line in a single associative set), the benefits are significant.

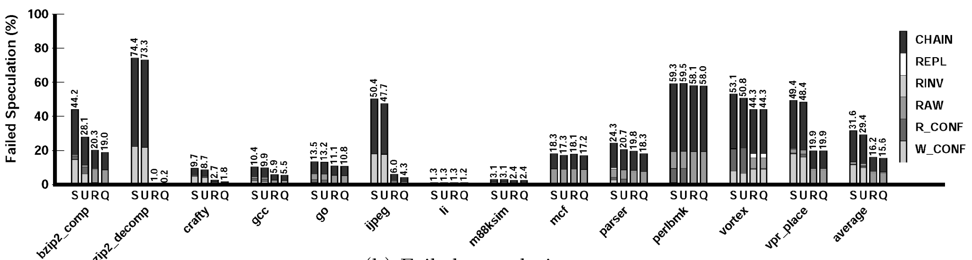
6.3.3 *Combining Suspension and Replication.* In the *Q* experiment in Figure 28, we evaluate the combination of both the suspension of epochs and cache line replication. Although replication (*R*) is more effective than suspension (*U*) for most benchmarks, compared with either technique in isolation we observe that the combination of the two techniques (*Q*) captures the best performance of either in nearly every case. Furthermore, for 5 applications (CRAFTY, GCC, GO, PARSER, and PERLBMK), this combination is complementary, achieving better performance than either technique alone. Overall, the combination of both techniques gives an average improvement of 13.5% over the basic shared-cache scheme.

6.4 Impact of Increasing Associativity

Increased associativity is usually desirable for shared-cache architectures, although there is a point where the increase in hit latency negates further benefit. Hence we want to ensure that our scheme for supporting TLS in a shared cache can also benefit from increased associativity—in particular, whether support for cache line replication can capitalize on the increased opportunity for storing replicated copies. In Figure 31 we repeat the experiments for suspension and replication (shown in Figure 28) for a shared cache with an associativity of 4 ways (as opposed to 2 ways, as used until this point): we maintain the original hit latency, and re-evaluate the sequential execution (to which all experiments are normalized) on a 4-way set-associative cache as well. Comparing with Figure 28, we observe that the performance of the baseline (*S*) with increased associativity is improved in most cases. Suspension and replacement are even more effective at tolerating conflicts and reducing failed speculation for most applications. With support for both suspension and replication (*Q*),



(a) Execution time.



(b) Failed speculation.

Fig. 31. Impact of suspending violations for replacement and conflicts when the shared data cache is 4-way set-associative (as opposed to 2-ways). *S* is the baseline 4-processor shared-cache architecture, *U* builds on *S* by tolerating conflicts and replacement through suspension of the logically-later epoch, *R* builds on *S* by tolerating conflicts through replication, and *Q* supports both suspension and replication.

failed speculation due to read/write conflicts is nearly or entirely eliminated for BZIP2_COMP, BZIP2_DECOMP, IJPEG, PERLBMK, VORTEX, and VPR_PLACE. These results confirm that increased associativity does not eliminate the need for suspension and replication, but instead further enables those techniques to improve the performance of shared-cache TLS.

7. CONCLUSIONS

Architectures that naturally support multithreading—such as chip-multiprocessors and simultaneously-multithreaded processors—have become increasingly commonplace over the past decade, and this trend will likely continue in the near future. However, only workloads composed of parallel threads can take advantage of such processors. We propose support for *Thread-Level Speculation* (TLS) that is simple and efficient, and can scale to a wide range of multithreaded architectures, while empowering the compiler to optimistically create parallel threads despite uncertainty as to whether those threads are actually independent. This claim is validated through a detailed evaluation of SPEC integer benchmarks, generated by a feedback-directed, fully-automated compiler, and measured on realistic, simulated hardware.

We have introduced a speculative cache coherence scheme that allows the compiler to automatically parallelize general purpose applications and to exploit moderate numbers of processors on a single chip. Our approach extends the architecture of a generic chip-multiprocessor without adding any large or centralized TLS-specific structures, and without hindering the performance of non-speculative workloads. Of 13 benchmark applications studied, our baseline architecture and coherence scheme improves program performance for two applications by 86% and 56%, for four other applications by more than 8%, and provides more modest improvements for six other applications—an average of 16% program speedup was achieved across all applications. A deep analysis of our scheme shows that our implementation of TLS support is efficient, and that our mechanisms for supporting speculation are not a bottleneck.

A closer look at our hardware support and speculative coherence scheme resulted in many important observations. We found that support for multiple writers is necessary for good performance for most general-purpose applications studied. Analyzing the sensitivity of our scheme to various architectural parameters, we found an expensive interprocessor communication mechanism to be unnecessary so long as a less expensive mechanism with latency on the order of 20–30 cycles can be implemented. Varying the sizes of the data caches demonstrated that 8kB caches are insufficient, although 64kB caches do not offer a significant improvement over 32kB caches. We also explored alternative designs for several aspects of our TLS hardware support.

Our evaluation of support for TLS in shared-cache architectures showed that performance is similar to that of private-cache architectures, since the increased cache locality of a shared-cache architecture is balanced with an increase in failed speculation due to conflicts. We also showed that two techniques for tolerating read and write conflicts—suspending conflicting epochs and replicating cache lines—can significantly lower the amount of failed speculation due to these conflicts.

Our approach to TLS empowers the compiler to optimize the performance of speculative threads, and frees the hardware from the burden of centralized structures and tightly-coupled connections. Our hardware support for TLS is unique because it scales seamlessly within chip boundaries and provides a framework for scaling beyond a chip—allowing this single unified design to apply to a wide variety of multithreaded processors and larger systems that use those processors as building blocks.

ACKNOWLEDGMENTS

We thank the reviewers and editor for their insightful comments.

REFERENCES

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

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

Akkary, H. and Driscoll, M. 1998. A dynamic multithreading processor. In *MICRO-31*.

Breach, S. E., Vijaykumar, T. N., Gopal, S., Smith, J. E., and Sohi, G. S. 1996. Data memory alternatives for multiscalar processors. Tech. Rep. CS-TR-1997-1344, Computer Sciences Department, University of Wisconsin-Madison.

Breach, S. E., Vijaykumar, T. N., and Sohi, G. S. 1994. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*. 181–190.

CINTRA, M. AND LLANOS, D. R. 2003. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, 13–24.

CINTRA, M., MARTÍNEZ, J. F., AND TORRELLAS, J. 2000. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of ISCA 27*.

CINTRA, M. AND TORRELLAS, J. 2002. Learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th HPCA*.

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

FARRENS, M., TYSON, G., AND PLESZKUN, A. 1994. A study of single-chip processor/cache organizations for large number of transistors. In *Proceedings of ISCA 21*. pp. 338–347.

FRANK, M., MORITZ, C., GREENWALD, B., AMARASINGHE, S., AND AGARWAL, A. 1999. Suds: Primitive mechanisms for memory dependence speculation. Tech. Rep. MIT/LCS Technical Memo LCS-TM-591. January.

FRANKLIN, M. AND SOHI, G. S. 1996. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput. 45*, 5 (May).

GARZARAN, M. J., PRVULOVIC, M., LLABERIA, J. M., VINALS, V., RAUCHWERGER, L., AND TORRELLAS, J. 2003. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*.

GOLDSTEIN, S. C., SCHAUSER, K. E., AND CULLER, D. E. 1996. Lazy threads: Implementing a fast parallel call. *J. Para. Distrib. Comput. 37*, 1 (Aug.), 5–20.

GOPAL, S., VIJAYKUMAR, T., SMITH, J., AND SOHI, G. 1998. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*.

GUPTA, M. AND NIM, R. 1998. Techniques for speculative run-time parallelization of loops. In *Proceedings of Supercomputing 1998*.

HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. 1998. Data speculation support for a chip multiprocessor. In *Proceedings of ASPLOS-VIII*.

KAHLE, J. 1999. Power4: A Dual-CPU processor chip. *Microprocessor Forum '99*.

KNOOP, J. AND RUTHING, O. 1992. Lazy code motion. In *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*.

KRISHNAN, V. AND TORRELLAS, J. 1999a. A chip multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput. Special Issue on Multithreaded Architecture*.

KRISHNAN, V. AND TORRELLAS, J. 1999b. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

LAUDON, J. AND LENOSKI, D. 1997. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th ISCA*. 241–251.

MARCUELLO, P. AND GONZÁLEZ, A. 2002. Thread-spawning scheme for speculative multithreading. In *Proceedings of the 8th HPCA*.

MARCUELLO, P. AND GONZLEZ, A. 1999. Clustered speculative multithreaded processors. In *Proceedings of the ACM International Conference on Supercomputing*.

MOSHOVOS, A. I., BREACH, S. E., VIJAYKUMAR, T., AND SOHI, G. S. 1997. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th ISCA*.

OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. 1996. The Case for a Single-Chip Multiprocessor. In *Proceedings of ASPLOS-VII*.

OOI, C. L., KIM, S. W., PARK, I., EIGENMANN, R., FALSAFI, B., AND VIJAYKUMAR, T. N. 2001. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the International Conference on Supercomputing*.

OPLINGER, J., HEINE, D., AND LAM, M. S. 1999. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*.

PALACHARLA, S., JOUPPI, N. P., AND SMITH, J. E. 1996. Quantifying the complexity of superscalar processors. Tech. Rep. CS-TR-1996-1328, University of Wisconsin-Madison.

PARK, I., FALSAFI, B., AND VIJAYKUMAR, T. N. 2003. Implicitly-multithreaded processors. In *Proceedings of the 30th annual international symposium on Computer architecture*.

Prabhu, M. and Olukotun, K. 2003. Using thread-level speculation to simplify manual parallelization. In *Principles and Practices of Parallel Programming*.

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

Rauchwerger, L. and Padua, D. 1995. The LRPD Test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of PLDI '95*. 218–232.

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

Roth, A. and Sohi, G. 2001. Speculative data-driven multithreading. In *7th International Symposium on High Performance Computer Architecture (HPCA-7)*. 20–24.

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

Sohi, G. S., Breach, S., and Vijaykumar, T. N. 1995. Multiscalar processors. In *Proceedings of ISCA 22*. 414–425.

SPEC. 2000. The SPEC Benchmark Suite. Tech. rep., Standard Performance Evaluation Corporation. http://www.spechbench.org.

Steffan, J. G. 2003. Hardware Support for Thread-Level Speculation. Ph.D. thesis, Carnegie Mellon University. Tech. Rep. CMU-CS-03-122.

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

Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. 2002. Improving value communication for thread-level speculation. In *Proceedings of the 8th HPCA*.

Steffan, J. G., Colohan, C. B., Zhaia, A., and Mowry, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of ISCA 27*.

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

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

Tullsen, D. M., Eggers, S. J., and Levy, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of ISCA 22*. 392–403.

Veenstra, J. 2000. `MINT+` mips emulator. Personal communication.

Vijaykumar, T. 1998. Compiling for the multiscalar architecture. Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison.

Yeager, K. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro*.

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

Zhaia, A., Colohan, C. B., Steffan, J. G., and Mowry, T. C. 2004. Compiler optimization of memory-resident value communication between speculative threads. In *Proceedings of the International Symposium on Code Generation and Optimization*.

Zhang, Y., Rauchwerger, L., and Torrellas, J. 1998. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*.

Zhang, Y., Rauchwerger, L., and Torrellas, J. 1999. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Fifth International Symposium on High-Performance Computer Architecture (HPCA)*. 135–141.

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

Zilles, C. and Sohi, G. 2002. Master/slave speculative parallelization. In *35th International Symposium on Microarchitecture (MICRO-35)*. 18–22.

Received June 2003; revised December 2004; accepted January 2005