Exploring Thread-Level Speculation in Software: The Effects of Memory Access Tracking Granularity

Spiros Papadimitriou   Todd C. Mowry

July 2001
CMU-CS-01-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Speculative execution is often the only way to overcome dataflow-imposed limitations and exploit parallelism when dependences can be discovered only at run-time. It also facilitates automatic parallelization of programs that exhibit complicated memory access patterns, which make complete compile-time dependence analysis either impossible or extremely complicated.
A number of approaches for coarse-grained data and control speculation have been proposed, mainly in hardware. A few software-only methods exist, but they usually rely on certain assumptions about control flow or memory access patterns. We investigate the possibility of supporting speculation in software, without making any such assumptions. The main motivation is the success of a number of software DSM systems. Our approach utilizes the virtual memory hardware to track memory operations. The goal is to provide the necessary support through a software library, instead of specialized hardware. We found that a low overhead mechanism that can track memory accesses at a small granularity is necessary, thus making the virtual memory mechanism unsuitable. In this paper we explore the various overheads and, in particular, those related to the granularity of memory access tracking.

Keywords: thread-level speculation, virtual memory, granularity

1 Introduction

A number of hardware-based schemes [SM98, ONH$^+$96, THA$^+$99, SBV95] for thread-level speculation (TLS) have been proposed in the past. A question that, to the best of our knowledge, has not been explicitly investigated in the past is how much added hardware support in necessary for TLS. Our initial goal was to essentially "simulate" our hardware-based scheme purely in software by utilizing the existing virtual memory mechanism, rather than adding new hardware. In the course of building our system, we gained some insight about the limitations imposed by existing hardware mechanisms, which have been designed for different purposes, as well as characteristics of programs that are good candidates for TLS.

After briefly explaining TLS and how it can be supported, we present the design and implementation of our system. Performance measurements are followed by the main discussion, which aims to explain the conflict mentioned above and provide a detailed overview of related work.

1.1 What is TLS?

Thread-level speculation (TLS) allows the compiler to automatically parallelize portions of code in the presence of statically ambiguous data dependences. This allows the dynamic extraction of parallelism available between whatever dependences actually exist at run-time. Under TLS, a program is broken into dynamic instruction sequences called *epochs*, which the compiler believes are likely to be independent. For example, the iterations of a loop might each be an epoch, if the compiler believes that cross-iteration dependences are unlikely. A software-managed epoch number is associated with each epoch. This specifies the original ordering of the epochs under sequential execution. The epochs are executed in parallel and the epoch numbers are used to detect whether data dependences have been violated. All speculative side-effects are buffered until they can be safely committed to memory. If a dependence violation is detected, execution is resumed using the correct data, after the appropriate recovery actions have been taken.

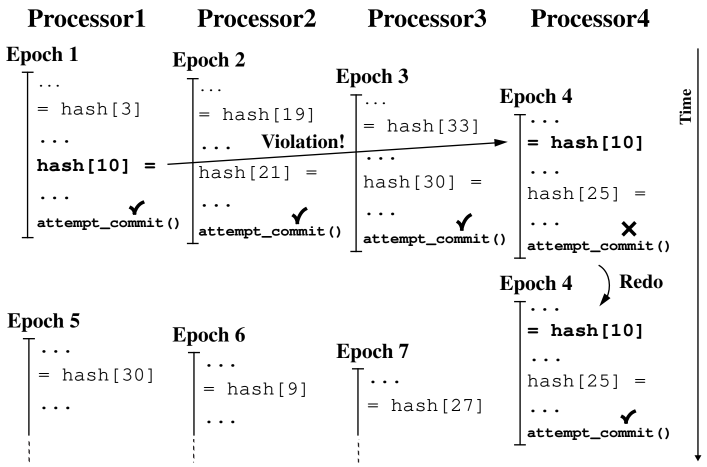
To illustrate how TLS works, consider the simple while loop in Figure 1, which accesses elements in a hash table. This loop cannot be statically parallelized due to possible data dependences through the hash table. While it is possible that a given iteration will depend on data produced by the preceding iteration, these dependences may in fact be infrequent if the hashing function is effective. Hence, a mechanism that could speculatively execute the loop iterations in parallel — while squashing and re-executing any iterations which do violate dependences — could potentially speed up this loop significantly, as illustrated above. Here, a read-after-write (RAW) data dependence violation is detected between epochs 1 and 4. Hence epoch 4 is squashed and restarted to produce the correct result. This example demonstrates the basic principles of TLS and can also be applied to regions of code other than loops.

1.2 Supporting TLS

Our main target architecture in the STAMPede project is a generic single-chip multiprocessor where each processor has its own primary data cache and all processors on the same chip physically share a secondary cache. The instruction set is extended with new instructions which enable software to manage TLS, to use the caches to buffer speculative state, and to extend the cache coherence scheme to detect data dependence violations. Readers interested in a more detailed description of this architecture can consult [SCM97, SM98, SCZM00].

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

(a) Pseudo-Code



(b) TLS Execution

Figure 1: Example of TLS execution.

Another option is to provide TLS support purely in software, without adding any new hardware support. An essential requirement we set from the beginning was to avoid imposing any further restrictions and requirements upon the compiler or the programmer. In fact, it would be highly desirable for both the hardware-based and software-only approaches to use the same high-level abstractions and support the same (or at least very similar) APIs.

A number of systems have provided the illusion of shared memory through software. Their success has been significant, but qualified—after all, there is still a market and active research on hardware shared memory as well as message passing architectures.

When implemented in hardware, shared memory also involves added support at the data cache level. The first software DSM systems exploited the virtual memory hardware and, in particular, the memory protection features. They employed essentially the same sequential consistency protocols [Lam79] used in bus-based hardware DSMs and simulated a "cache" with page-sized lines and latencies determined by the network used. However, the overheads were significant. In particular, because of the large block sizes, *false sharing* was a problem. A number of more recent systems [KCDZ94, Kel97] resort to methods (such as *diffing* and *lazy-release consistency*) that reduce these overheads significantly and, in most cases, yield a respectable speedup.

There is also another class of software-based DSMs [SFL$^+$94, SFH$^+$98, SGT96] that instrument all loads and stores into "shared" memory, instead of using the virtual memory hardware. Other hybrid techniques have also been explored, such as employing custom-designed memory boards to avoid the overhead of instrumenting memory instructions. Since designing a slightly modified memory board is considerably simpler than designing a new processor, this approach does make sense.

The two types of software-only DSMs (virtual memory vs. code instrumentation) have comparable performance. The main appeal of using the virtual memory mechanism is that the entire system can be implemented simply as a library, rather than via specialized executable editing tools, compiler extensions[1] or operating system extensions. Since our aim was to support TLS with minor or no modifications to the compiler infrastructure already under development for the hardware version, using the virtual memory mechanism seemed to be the best choice.

However, for the most part, our main discussion and analysis does not depend on this choice. Our observations are based on program behavior rather than our system's particular behavior. A different system architecture or different workloads only change where the trade-off points occur. We argue that with most workloads and reasonable assumptions about system overheads, it is very difficult to achieve performance gains.

1.3 Contributions

The contributions of this work are the following:

- We built a system for software TLS, which uses the virtual memory system. This derives from previous, successful work in software DSMs and incorporates a number of optimizations. We found that the virtual memory mechanism is unsuitable for speculation support, despite its success in software DSMs.

- We examine overhead factors in detail and how each of those affects performance. We present measurements that are derived from real programs and are largely system-independent. Based on these observations, we characterize a number of existing approaches and systems and argue about what is needed to successfully support TLS.

In particular, block size (i.e., the granularity at which memory access tagging is done) is a very important factor. The random memory access patterns of good candidate applications for speculation can easily cause false sharing problems. Besides needlessly increasing communication overheads, false sharing also causes false violations, which result in serialized execution. When using the virtual memory system, the block size is unavoidably large (equal to the size of a page). A very low overhead mechanism that can support relatively small block sizes is necessary to successfully support speculation. Another key issue is support for buffering and undoing memory operations. Although alternative software-based methods could potentially be used, it is not entirely clear that these would be sufficient. We believe that at least some minimal added hardware support for tracking and buffering memory operations is necessary.

2 Design and Implementation

In order to facilitate speculation, we use the virtual memory hardware to track speculative reads and writes. By setting the page protections appropriately and trapping to user-level code in response to access violations on shared pages, it is possible to track accesses to regions of shared memory. By exchanging messages between different threads, the local copies of the shared memory image in each processor can be brought up-to-date.

Our goal was not to add support for shared memory on cheap platforms that lack it, such as networks of workstations, but rather to enable TLS on existing systems. Therefore, we chose to build our

[1] In order to do automatic parallelization compiler/language extensions are still necessary. However, to use software TLS a programmer need only include the proper header files and libraries, rather than use special compiler directives or language extensions.

software TLS layer on top of a hardware shared-memory multiprocessor, such as the SGI Origin. The main advantage is the significantly reduced communication latency.

2.1 Software Interface

In this section we will give a high-level view of the abstraction that our system provides to the programmer and compiler. We will also try to justify our design choices. Table 1 defines our high-level primitives for software TLS and Figure 2 shows how all these are put together in an actual implementation of the example in Figure 1.

During initialization, one process is spawned for each processor in the system. After allocating initial shared memory regions, each process executes the system's main loop by calling `tls_start()`. This is responsible for fetching and executing epochs.

Shared memory is managed by calls to `tls_mem_alloc()` and `tls_mem_free()`, which are the equivalents of `malloc()` and `free()` for shared speculative memory. We will use the term *speculative memory* to distinguish the shared memory image provided by the software TLS layer from the actual shared memory used in its implementation. Since the allocation of a memory block is performed by one of the processors, some mechanism is needed to initially broadcast a pointer to the first shared block. This can be done with `tls_mem_send()` and `tls_mem_recv()`, which normally need to be called only once, during the initialization phase.

In order to have well-defined entry-points, the code for each epoch should be enclosed within a procedure[2]. An epoch can be created with `tls_epoch_new()`. Each epoch is represented by an opaque data structure and can be accessed from any processor (see section 2.2 for details). Forwarding buffers are associated with each epoch, both for data that are forwarded during epoch creation (forwarding frame), as well as during later stages of execution. A number of forwarding flags are also associated with each epoch. These are used for synchronization between the sender and recipient. All of the above need to be declared during the epoch creation stage, along with the epoch sequence number. Sequence numbers can be obtained and reserved with `tls_seq_get()`.

After an epoch has been created, it can be scheduled for execution with `tls_epoch_fork()`. This operation appends the epoch to the pending queue (see section 2.2).

Within each epoch procedure, all accesses to speculative memory have to be enclosed within calls to `tls_epoch_begin()` and `tls_epoch_end()`. These should be called once per epoch and perform the necessary initial and final work (respectively) necessary to track memory accesses. The latter also terminates the current epoch, returning execution to the main loop of the system.

The `tls_epoch_fwd_set()` and `tls_epoch_fwd_signal()` pair is used to forward values during epoch execution. The operations `tls_epoch_fwd_get()` and `tls_epoch_fwd_wait()` should be used at the receiving end, respectively. Note that, in the current implementation, the latter blocks execution until the desired value has been received.

The functions for setting violation and cancellation handlers, as well as canceling and killing an epoch are relatively straightforward. Epochs are never pre-empted, since this would impose a large overhead because, as we shall see, saving and restoring speculative state is very expensive. Checks for violation or cancellation are always performed when an epoch finishes. However, an early check can be

[2]The use of C procedures does not preclude access to variables from enclosing scopes (e.g., variables defined outside the body of a loop that needs to be speculatively parallelized). Data from enclosing scopes can either be copied through forwarding buffers as explained later, or accessed through a forwarded pointers to these data items (or possibly the entire stack frame). This will typically be handled automatically by the parallelizing compiler.

Operation	Description
`tls_init(argc, argv)`	Initialization of processes, shared-memory data structures and synchronization primitives.
`tls_shutdown()`	Cleanup
`tls_start()`	Start main epoch execution loop
`tls_mem_alloc(size)` `tls_mem_free(ptr)`	Shared memory management
`tls_mem_send(ptr, len)` `tls_mem_recv()`	Initialization of pointers to shared memory
`epoch = tls_epoch_new(seq, &proc, fwd_frame, fwd_frame_size, fwd_data_size, fwd_num_flags)`	Creation of new epoch data structure (contains forwarding buffers and flags).
`tls_epoch_fork(epoch)`	Insertion of epoch into pending queue
`tls_epoch_begin()`	Initialization of twin lists for current epoch
`tls_epoch_end()`	Move epoch into finished queue and jump to beginning of main epoch execution loop
`tls_epoch_fwd_set(epoch, ofs, data, len)`	Copy data into forwarding buffer of designated epoch
`tls_epoch_fwd_get(epoch, ofs, ptr, len)`	Copy data from forwarding buffer
`tls_epoch_fwd_wait(epoch, nflag)`	Wait for forwarding flag of designated epoch to be set (synchronization primitive)
`tls_epoch_fwd_signal(epoch, nflag)`	Set forwarding flag
`tls_epoch_cancel(epoch)`	Cancel epoch (invokes cancel handler)
`tls_epoch_kill(epoch)`	Destroy designated epoch and free all related data structures (blocking operation)
`tls_epoch_fail_handler(epoch, &hnd, arg)`	Set callback to invoke upon dependence violation
`tls_epoch_cancel_handler(epoch, &hnd, arg)`	Set callback to invoke upon cancellation
`tls_epoch_check_fail()`	Force early check for dependence violation
`tls_epoch_check_cancel()`	Force early check for cancellation
`tls_seq_get(nreserve)` `tls_seq_next(seq)`	Sequence number management

Table 1: Software-TLS operations.

forced, if necessary, by calling `tls_epoch_check_cancel()` or `tls_epoch_check_fail()`. These can be inserted by the compiler and/or programmer before starting highly time-consuming work.

2.2 Implementation

As we have already mentioned, our base platform for software TLS is a hardware shared-memory multiprocessor. The implementation essentially employs a work-queue model and is built upon a central data structure, which is located in shared memory.

The main component of this data structure is a set of four queues. At any point in time, all epochs currently in the system can be found in one of these queues. The epochs in each queue are ordered by sequence number. These queues contain epochs that are in various stages of execution:

- **Pending execution:** This queue contains epochs that have been scheduled for execution. Upon completion of an epoch, each processor fetches the next available epoch from this queue and begins executing it.
- **Executing:** This is simply a list of all epochs that are currently being executed by some processor in the system.
- **Finished execution:** As soon as an epoch completes execution, it is moved into this queue. Since epochs with smaller sequence numbers may still be executing, checking for dependence violations may not be immediately possible, hence the need for this queue. Upon completion of an epoch and before fetching the next, each processor eagerly performs all necessary dependence checks for as many epochs as possible from this queue. An epoch is checked if all preceding ones have been cleared to commit. If the checks fail, the appropriate violation handler is called and can decide whether to kill the epoch or re-schedule it for execution.
- **Clear to commit:** If the check for dependence violations is successful, an epoch is moved into this queue. An epoch needs to be kept in the system until all processors have committed the changes it has made to speculative memory. Before beginning execution of an epoch, each processor aggressively updates its state by traversing all epochs in this queue. A quick "garbage collection" check is executed during this traversal to destroy epochs whose state has been committed by all processors and which are no longer pointed to as "last seen" epochs, as explained later.

The work performed on each queue comprises the main loop of the system, which is started by `tls_start()`—the actual implementation uses long jumps. A global lock is used to ensure mutual exclusion during accesses to the queues.

Epochs can be explicitly destroyed with kill operations. Explicit cancellation is necessary for control speculation, as well as potentially useful for more complicated run-time optimizations. A kill operation blocks until all processors have finished executing their current epochs. Then, the state of each local image of the shared memory is restored to that of the epoch immediately preceding the one being killed. Finally, all epochs with larger sequence numbers are purged from the system. We chose this implementation for various reasons. First of all, we expect kills to be very infrequent. Also, doing otherwise would introduce a large amount of complexity in managing the queues and especially in the garbage collection function, which is frequently executed. Most importantly, we do not expect epochs logically succeeding a killed one to contain enough useful work to justify the extra overhead.

```
#include <tls.h>

typedef struct {
   int hash[N];
} shmem_t;
shmem_t *pmem;

typedef struct {
   int index1, index2;
} frame_t;

void f_iter (void *frm, int size, bool restart)
{
   frame_t *frame = (frame_t *)frm;
   int x, y; /* Local variables */

   if (!restart && continue_cond) {
      frame_t next_frame = {...};
      tls_epoch_seq_t seq = tls_seq_get(1);
      tls_epoch_t next_epoch = tls_epoch_new(seq, f_iter, &next_frame, sizeof(next_frame), 0, 0);
      tls_epoch_fork(next_epoch);
   }

   tls_epoch_begin();
   x = pmem->hash[frame->index1];
   ...
   pmem->hash[frame->index2] = y;
   ...
   tls_epoch_end();
}

static void initialize (void)
{
   pmem = (shmem_t *)tls_mem_alloc(sizeof(shmem_t));
   ...
}

static void start (void)
{
   frame_t frame = {...};
   tls_epoch_seq_t seq = tls_seq_get(1);
   tls_epoch_t *first_epoch = tls_epoch_new(seq, f_iter, &frame, sizeof(frame), 0, 0);
   tls_epoch_fork(first_epoch);
}

int main (int argc, char *argv[])
{
   tls_init(argc, argv);
   if (tls_pid == 0) {
      initialize();
      tls_mem_send(&pmem, sizeof(pmem));
      start();
   } else
      tls_mem_recv();

   tls_start();

   tls_shutdown();
   exit(0);
}
```

Figure 2: Putting it all together: the example in Figure 1 using our API.

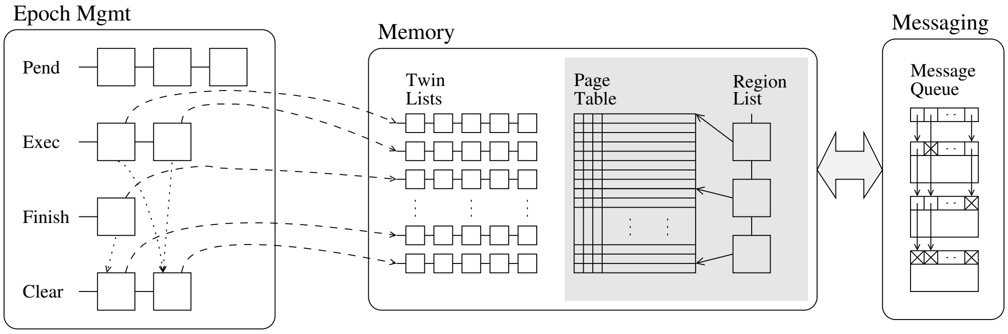


Figure 3: High-level view of software-only implementation. Shaded structures are *not* located in shared memory. The epoch management queues are lists of epoch structures, each of which points to a "last seen" epoch (once executing) and a list of page twins (once finished execution). Each processor maintains a private page table, which holds information about the private image of speculative memory. This information consists of read/write dirty bits and a pointer to a newly created page twin, if the page is write-dirty. These twins are gathered into a list when execution of an epoch completes. Pages of speculative memory are allocated in large chunks, or *regions*—creating a new region roughly corresponds to an `sbrk()`. The messaging subsystem is used to broadcast necessary updates, as well as during initialization and for any other exchange of information that do not fit in central epoch queues and linked structures. As shown in the figure, each message belongs to a number of message queues (linked lists), one per processor.

The epoch data structure (see Figure 4) is of central importance. This is opaque to the programmer/compiler, it is allocated by `tls_epoch_new()` and contained in the queues described above. Besides the basic information (such as sequence number, parent epoch sequence, executing processor, pointer to epoch procedure, etc.), the following information is associated with each epoch structure:

- **Forwarding frame:** This is simply a buffer containing a copy of the data forwarded to the epoch during creation; it is part of the epoch structure.

- **Forwarding buffer and flags:** This is space reserved for forwarding data after epoch creation. A flag is associated with each data item in the forwarding buffer. These flags are used for synchronization between sending and receiving epochs. All of the above are parts of the epoch structure and need to be pre-allocated during its creation.

- **Twin list:** This is a pointer to a list of copies of all pages written by an epoch during its execution. Whenever a page is first written, a clean copy of the page is stored. When an epoch finishes execution, a copy of the final contents of each page is also made. The twin list is also allocated in shared memory and thus accessible by all processors. All this information is used for dependence checking, as well as updating memory state, both during committing as well as undoing an epoch—a more detailed explanation will be provided later.

- **Last seen epoch:** This is simply a pointer to the epoch which was last committed on the processor that executed this epoch. This information is used when checking for dependence violations—only epochs after the "last seen" one need to be checked against.

<table>
<tr><td colspan="3">SeqNo</td><td colspan="3">PID</td></tr>
<tr><td colspan="3">ParentPID</td><td colspan="3">ParentStackPtr</td></tr>
<tr><td colspan="3">FailHndPtr</td><td colspan="3">FailHndArgPtr</td></tr>
<tr><td colspan="3">CancelHndPtr</td><td colspan="3">CancelHndArgPtr</td></tr>
<tr><td colspan="3">State</td><td colspan="3">RefCount</td></tr>
<tr><td colspan="6">PrevSeenPtr</td></tr>
<tr><td colspan="6">TwinListPtr</td></tr>
<tr><td colspan="2">FailFlag</td><td colspan="2">CancelFlag</td><td colspan="2">KillFlag</td></tr>
<tr><td colspan="6">Forwarding Buffers</td></tr>
<tr><td colspan="6">Forwarding Flags</td></tr>
</table>

Figure 4: Epoch data structure. The first and third portions are information set by the programmer or compiler. The latter (forwarding buffer and flags) is variable size, but the size has to be specified at epoch creation time. The middle portion is used by by the system during various epoch execution stages, as explained in the text.

The above discussion brings us to the memory subsystem, which is the part whose design is based upon software-DSM systems. Each processor keeps a private copy of the shared speculative memory. Thus, each epoch reads and modifies only the local copy of the processor upon which it executes. The virtual memory system is used to track these accesses. Initially, each page is both read- and write-protected. Upon the first read operation, a segmentation violation signal (SIGSEGV) occurs, which is trapped to user-level code. The signal handler sets a *read dirty* bit and the read protection is removed from the page, until the epoch completes execution. The first write operation also triggers a similar chain of events. Thus, for each epoch, only the first read and the first write accesses will cause an expensive signal handler invocation.

We have incorporated one more important idea from software-DSM systems. Practically all such systems implement *multiple-writer* protocols. Whenever a page is first written, a *twin* is created—i.e., a clean copy of its initial contents. By comparing the initial and final contents of the page, we can achieve two things. First, we can actually track writes at a word (or even byte) granularity and thus reduce *false sharing* problems. Second, if multiple processors (in our case, epochs) have modified a certain page, it is possible to merge these modifications. Software-DSM systems always use *diffs* to exchange page modifications between processors. This has the advantage of reducing the size of exchanged messages, and thus the communication overhead. However, since our TLS layer is built on top of a tightly-coupled hardware multiprocessor, the overhead of computing the diffs is usually greater than exchanging the entire "before" and "after" contents of a page through shared memory. Therefore, we do the equivalent of *diffing*[3] *only* when it is necessary to merge modifications, and not

[3] Diffing is performed at a word granularity by default. Smaller granularities impose a high overhead, mainly because

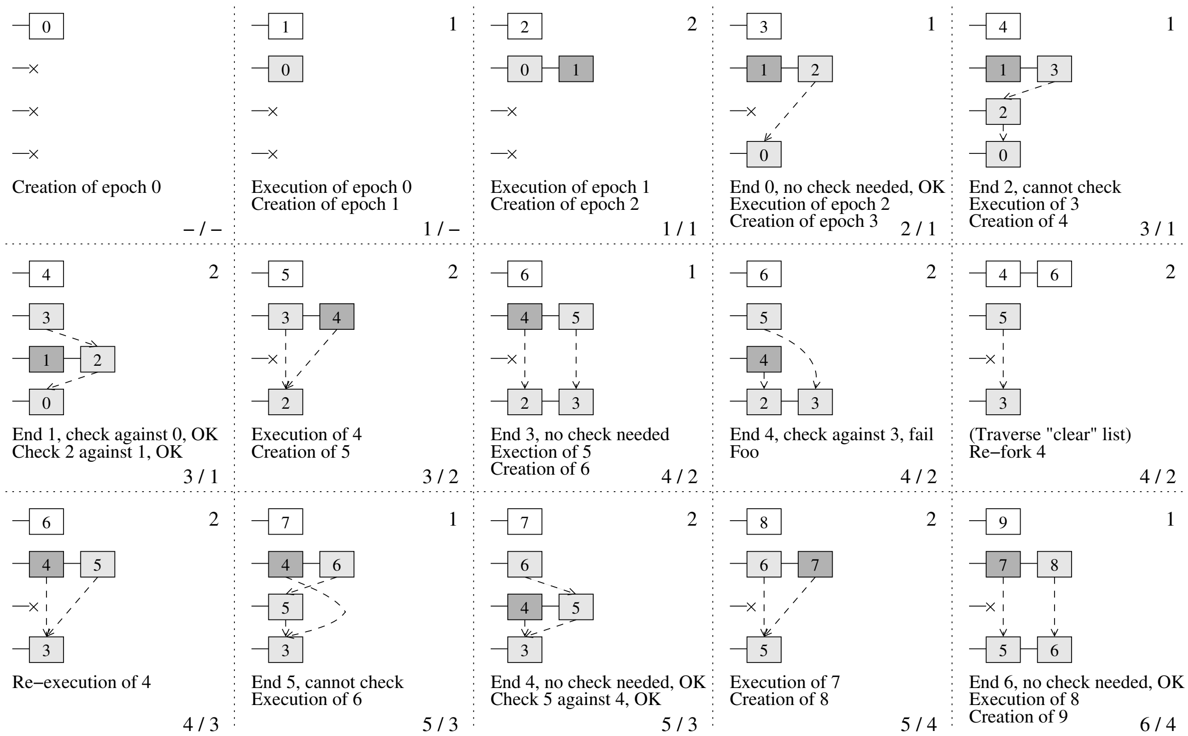


Figure 5: Central data structure during progress of execution, on a system with two processors. The "last seen" pointer is shown with a dashed line. The iteration number of the main loop for each processor is shown at the bottom right of each picture. The processor performing the actions listed in each picture is shown on the top right. Epochs executed on the first processor are lightly shaded, whereas epochs executed on the second processor are shaded dark.

when checking for dependence violations.

There are three possible types of dependences. Each of these poses different problems and is dealt with as follows:

- **Write-after-read (WAR):** Whenever an epoch begins execution, any modifications by epochs with larger sequence numbers are undone. This may be necessary if the epoch starting execution is one which was previously violated or canceled, since logically later epochs may, in the meantime, have been executed on the same processor. Since each epoch accesses the local copy of the speculative memory, WAR dependences cannot be violated.

- **Write-after-write (WAW):** The support for multiple-writers with on-demand diffing resolves such dependences: we just have to apply the write updates in the order dictated by epoch sequence numbers.

- **Read-after-write (RAW):** The good news is that this is the only type of dependence that may cause an epoch to fail. The bad news is that these dependences are difficult to identify correctly.

of inefficient memory accesses—in most modern processors, loads and stores are optimized for word-size accesses and accessing parts of a word often requires extra instructions (bit masking and shifting). Any granularity can be chosen during system startup.

> Since we rely solely on the virtual memory hardware, reads can only be tracked at a page granularity. Therefore, whenever a page is both written by an epoch and read by one with a smaller sequence number, we have to conservatively assume that there may have been a dependence violation, even though the sets of words read and words written may be disjoint. The reason for this is that we have no way of identifying the former. Of course, if we do not rely exclusively on the virtual memory mechanism, we can deal with this problem of false violations (which are caused by false sharing). For instance, Shasta [SGT96] uses compiler-inserted instructions to tag the precise address of each read. However, this imposes a significant overhead on each read operation, or at least those to speculative memory.

We have implemented our own equivalents of `malloc()` and `free()` for managing the shared speculative memory. The speculative memory heap consists of a list of regions. New regions are allocated whenever necessary. A relatively simple message-passing subsystem is used for exchanging the information necessary to coordinate these operations. The implementation is based on a shared message queue and supports "broadcasting" (each message may belong to as many chains as the processors in the system), message type tags and both blocking and non-blocking send/receive operations; the design was meant to be general enough to support our system's needs as it evolved. This message passing subsystem is also used during initialization to broadcast any necessary information, as well as for the implementation of the `tls_mem_send()`/`tls_mem_recv()` pair of functions. These are normally used during the initialization phase to broadcast the address of the initial block of shared memory.

3 Performance Evaluation

The first part of this section describes the speedups achieved with purely synthetic workloads. This serves a dual purpose: to show that our system does indeed achieve speedups under certain conditions, and to give a rough idea of what these conditions are. The second part describes our observations from a real program and provides a starting point for our analysis. As we shall see, these observations paint a very different picture.

3.1 Ideal Conditions

We performed measurements with a synthetic benchmark, to estimate performance under "ideal" conditions. The benchmark consists of a simple loop that accesses an array of page-sized elements (see Figure 6). The array has as many elements as there are CPUs available in the system and each loop iteration modifies exactly one word in array element $i \mod N$ (where i is the iteration number and N is the number of processors) and then idles for a certain amount of time.

The results from this simple benchmark are shown in Figure 7 and provide an indication about the overheads associated with page diffing and copying in our implementation. As we see, given sufficiently large units of work (in terms of *time* required to complete them), we achieve good speedup. In itself, this is a reasonable requirement. However, as we shall soon see, the overheads increase linearly with the size of work units in terms of memory *space* they touch. For real programs, this linear increase involves a very large multiplicative constant. Reducing this requires some form of added hardware support.

```
#define I_DEP 1

int a[PAGE_SIZE * NUM_ELT];

void loop (void)
{
    int i;
    for (i = 0; i < NUM_ITER; i++) {
        int ai = i % NUM_ELT;
        if (ai == 0)
            write a[I_DEP * PAGE_SIZE];
        read a[ai * PAGE_SIZE];
        busy loop;
    }
}
```

Figure 6: Synthetic workload.

3.2 The Real World

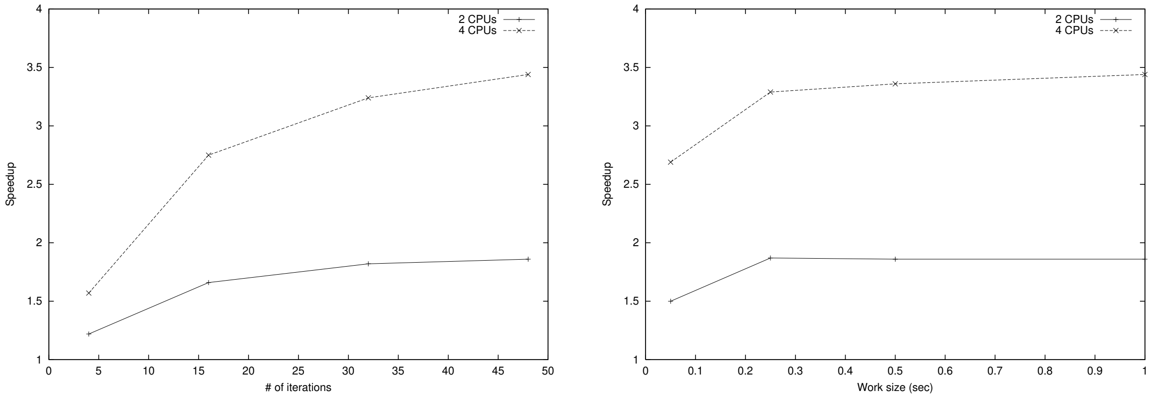
We use bucket sort (see Figure 8) as a starting point for our analysis of overheads incurred when running real applications. We implemented a standard version and chose the following parameters:

- Input: an array of 512K integers
  This takes takes up a total of $2\,\mathrm{Mb}$, or 512 pages on the Intel IA32 architecture[4].
- Key-set size: the values of the integer elements were in the range of 0 to 1023.
  Thus, the frequency count array fits in a single 4K page.
- Epoch size: In both counting and sorting phases each epoch goes through 2K integers.
  In other words, each epoch executes 2K iterations of the counting (read input array and update frequency counts) and sorting (write output array) loops. This amounts to unrolling 2K iterations from each of the loops in Figure 8. Therefore, there is a total of 512 epochs, 256 for each phase.

As noted above, the frequency count array fits in a single page. This results in *fully serialized* execution of the counting phase, since all epochs modify the exact same page. To avoid this, we padded each array element to an entire page. This introduces a factor of 1024 space overhead, as well as a time overhead, since for each modified element we need to go through and copy the entire page to other CPUs). The input and output arrays were left unpadded, since padding would introduce an extreme overhead.

During the sorting phase, we observed that each epoch touches on average 467 pages (minimum and maximum were 456 and 477 pages). In other words, practically all pages are touched by all epochs! As we will see, this is because of the page-granularity limitation imposed by the virtual memory mechanism and is not specific to the parameters or even the application. Since we need to compare the old and new contents of each modified page in order to perform diffing, we end up going through $467\times\mathbf{2}\times4\,\mathrm{Kb} \approx 3.65\,\mathrm{Mb}$ per $8\,\mathrm{Kb}$ work! This introduces a factor of 463 $(= 3.65\,\mathrm{Mb}/8\,\mathrm{Kb})$ overhead factor. Indeed, the actual execution times are about three orders of magnitude larger—approximately 5 minutes versus 2 seconds!

[4]Pages on the SGI machines are double in size, which makes things even worse.



(a) Speedup vs. running time

(b) Speedup vs. work size

Figure 7: Speedups achieved with synthetic loads.

Because of all the page copies maintained, memory utilization becomes a problem: although the dataset size is 2 Mb as noted above, the actual address space for each process was about 21 Mb. This degrades the memory system's performance and is a significant problem.

4 Overhead Analysis

In this section, we will describe all sources of overhead we observed, and discuss their causes. From these follow potential remedies, most of which have already been attempted, at least in some form. Thus, we will also try to place existing work on software TLS in this context.

4.1 False Sharing

One problem that hurts performance is *false sharing*. This generally occurs because certain information about memory accesses is typically not kept at the finest granularity (i.e., per byte or word). Usually such information is kept for each *block* (e.g., cache line or virtual memory page). Therefore, accesses to different elements of a block cannot be distinguished and this may lead to detection of false dependences. The problem of false sharing is aggravated by large block sizes. The graphs in Figure 9 and Figure 10 show exactly how the increase in block size leads to an increase in false conflicts.

In our case, diffing solves the problem for write accesses on the same page. In theory, we can obtain information about writes on a page at whatever granularity is desirable (although comparisons of quantities smaller than a word incur a significant performance penalty). This information, however, comes at a price. We will explain this in greater detail later, but the main issue is that, sooner or later, we need to go through the *entire* old and new images of the page and compare every single word. The sorting phase graphs (right column) in Figure 9 and Figure 10 show how often the diffing mechanism is triggered as block sizes increase. These graphs show what fraction of epochs suffer a WAW conflict with a previous one during the sorting phase. The actual number of conflicts is slightly higher if we

```
int a[NUM_ELT]; /* Input array */
int as[NUM_ELT]; /* Output, sorted array */
int f[NUM_KEYS]; /* Key count array */

void sort (void)
{
    int i;

    /* Counting phase */
    bzero(f, sizeof(f));
    for (i = 0; i < NUM_ELT; i++) /* Loop 1 */
        ++f[a[i]];
    /* Accumulation -- part of last counting epoch */
    for (i = 1; i < NUM_KEYS; i++)
        f[i] += f[i-1];

    /* Sorting phase */
    for (i = 0; i < NUM_ELT; i++) { /* Loop 2 */
        --f[a[i]];
        as[f[a[i]] = a[i];
    }
}
```

Figure 8: Bucket sort.

also take into account WAR violations.

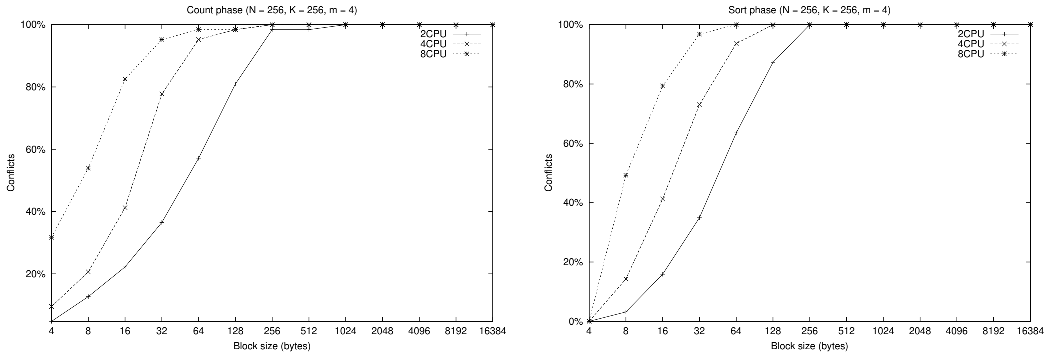
4.2 False Violations

Besides the added overhead of performing costly *diffing* operations, false sharing leads to another very serious problem, that of *false violations*. In traditional DSM systems, false sharing hurts performance by unnecessarily increasing data movement overheads. However, when we add speculation support, it may also hurt parallelism by causing false violations, which occur when two *speculatively* executing epochs touch different words within the same block.

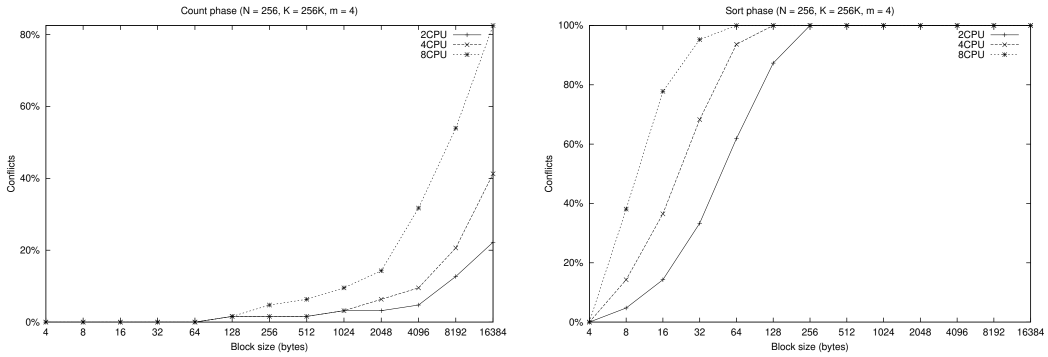
In the case of reads, if we rely on the virtual memory mechanism, we really cannot obtain information at a finer granularity than an entire page. Thus, we have to assume a violation has occurred. The counting phase graphs (left column) in Figure 9 and Figure 10 show how often epochs have to be canceled because of false WAR conflicts.

Even if it were possible to unprotect a page just for a single read to go through (which is not possible, at least with user-level code), invoking a signal handler upon every read entails enormous overheads. An alternative would be to instrument all read instructions [SGT96, SFL$^+$94]. However, in this case we have to give up the idea of avoiding compiler extensions,[5] special executable-editing tools or, possibly, specialized hardware such as the custom memory boards in [SFL$^+$94] for enhanced performance.

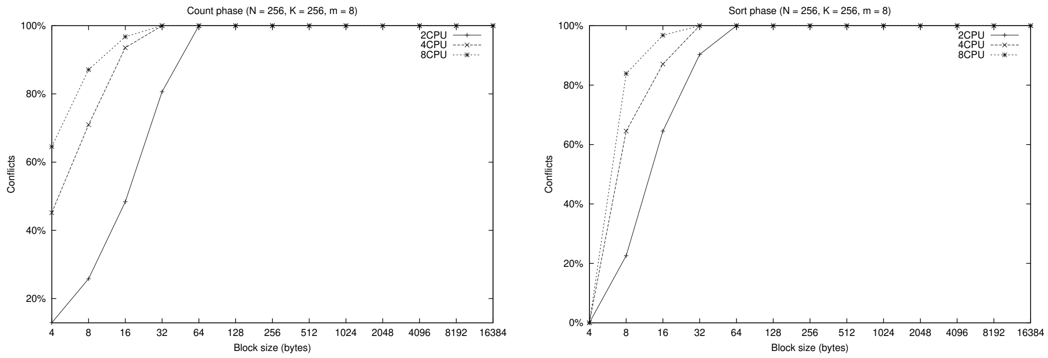
[5]Modifications to the compiler are necessary for automatic parallelization. However, compiler extensions for load/store annotation may have to be placed in the code-generation phase if we want them to be highly efficient. Furthermore, some extensions (either to the language or as pragmas) will have to be present for a programmer to use the system directly. Always requiring the presence of modified compiler in every platform in this way may be somewhat burdensome.



(a) Initial parameters

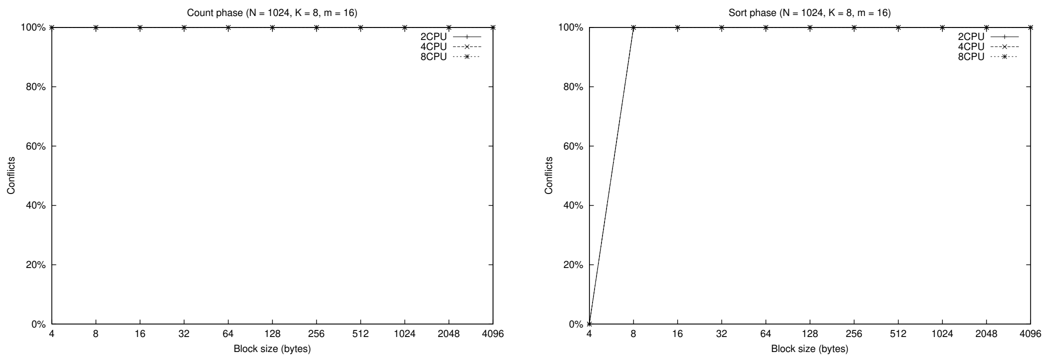


(b) Increased key set size

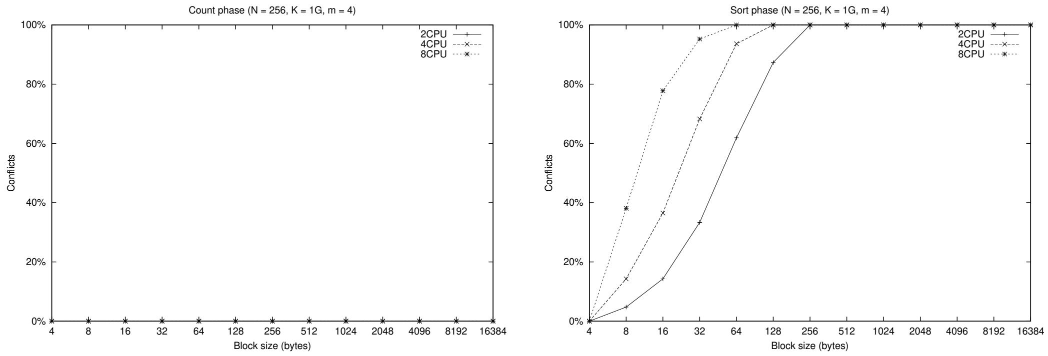


(c) Increased epoch size (unrolling)

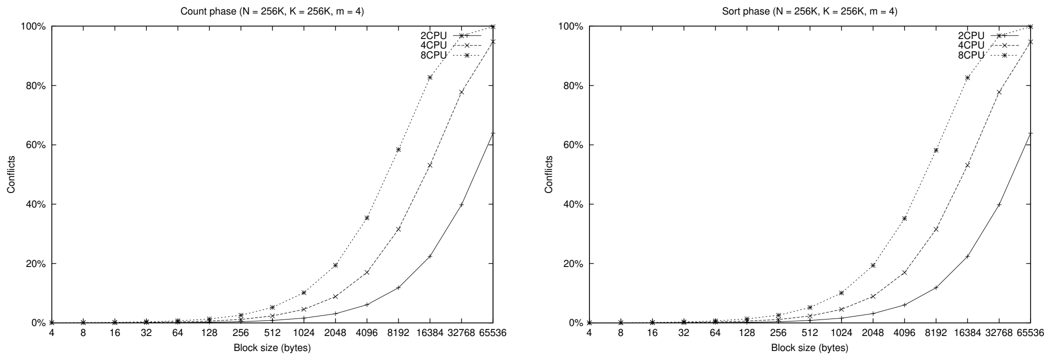
Figure 9: Epoch conflicts versus block size for varying array sizes (N), key set sizes (K) and epoch sizes (unroll factor u), as well as dependence window sizes w (which correspond to number of CPUs with round-robin execution). We only consider RAW and WAW dependences, not WAR. In the counting phase, all dependences occur through the key count array. Each element is read, incremented and written, thus all dependences are WAR and lead to epoch cancellation. For the sorting phase, we show only conflicts that arise through the sorted array, not the key count array. The sorted array is only written, thus all dependences are WAW and trigger the diffing mechanism. The vertical axis shows the percentage of epochs that have a RAW or WAW conflict with any of the $w - 1$ previous epochs.



(a) Relative parameters closer to our software TLS requirements



(b) Key set size much larger than number of elements



(c) Much smaller unrolling with respect to number of elements

Figure 10: Epoch conflicts versus block size with realistic and extreme choices of parameters. Once again, we consider only WAR and WAW conflicts and show only conflicts through the sorted output array in the sorting phase. For (b), we had to use a sparse representation for the key count array, but we show what the behavior would be if it were not (i.e., was implemented using a 4Gb array). The parameters in (a) were chosen based on the observation that the software TLS system needs a large unrolling factor (to cover the epoch setup overhead) and, consequently, a large array—necessarily much larger than the key set size.

One might argue that the problems caused by false sharing could be alleviated by developing smart memory placement algorithms in the compiler. However, the main goal of TLS is to enable automatic parallelization of programs for which data dependence analysis and, consequently, data partitioning to avoid them is very difficult or impossible to do at compile-time.

As mentioned before, in our bucket sorting benchmark, the key-count array for 1024 keys fits entirely in one page. Thus, the execution of all epochs in the key counting phase is fully serialized. A simple and straightforward way to avoid this is to pad each array element to occupy an entire page. This is wasteful in terms of memory usage and may cause further performance degradation due to cache misses and conflicts that would otherwise not occur. It also creates an artificial overhead. Whenever a single array element is modified, we need to examine the entire page to determine what was changed, or at best copy the entire page to each of the other processors. We will further discuss the problem of copying in the next section.

4.3 Maintaining Local Images

The main overhead of our system is in maintaining local images of the shared memory that may be speculatively accessed (which we call "shared speculative memory"). It is interesting to note here that the few software-only systems that support at least some form of speculation try to avoid privatization as much as possible.

As we have already described, each processor uses a local copy for all memory accesses. This copy must be updated on each processor. Interesting candidate applications for TLS typically exhibit a pattern of memory accesses that are not localized. Programs with localized memory access patterns can often be parallelized either at compile time or by using simpler techniques.

In the following paragraphs we will describe the two main aspects of the problem. It is important to note that this is not specific to the virtual memory approach, although large block sizes make things much worse.

4.3.1 Effects of Large Block Sizes

During program execution, information about memory accesses is kept on a per-block basis. When using the virtual memory system, blocks are entire pages. Whenever a single memory location within a block is accessed, *all* the locations in the same block are "tainted." If multiple processors have accessed the same block, then we must perform an expensive *diffing* operation. Otherwise, we can at best copy the entire block verbatim into each processor's local image. In both cases, however, we have a high overhead factor.

More precisely, if the block size is N_{blk} bytes and, on average, N_{acc} out of these are accessed by an epoch, then the *overhead factor* is *at least* $f_{min} = N_{blk}/N_{acc}$.

Let us assume a page size[6] $N_{blk} = 4096$. For instance, during the key counting phase of bucket sort, we have $N_{acc} = 4$ because of the padding to avoid false sharing. Thus, the overhead factor is $f_{min} = 1024$. During the sorting phase, $N_{acc} = 17.6$ in our setup and $f_{min} \approx 233$. However, the actual factor is closer to $f = 463$ (i.e., about $2f_{min}$), because of the expensive *diffing* operation.

Also, a large block size significantly increases the chance that the same block will be modified by more than one processor. The first option is to conservatively assume that there has been a dependence

[6]This is the page size that Linux/i386 uses, which is relatively small. Irix on the SGI Origin uses a page size of 16 Kb, which makes things much worse.

violation, but this results in serialized execution of most epochs. If we want to avoid this, we have to compute diffs and pay a large performance penalty.

The above behavior is indeed exhibited by bucket sort. In fact, in the sorting phase the situation is even worse. Not only is each block touched by more than one processor, but each processor by itself practically accesses all the blocks in shared speculative memory! Figure 11 shows in detail how the fraction of total blocks touched by each epoch increases with block sizes.

Aside from the copying/diffing overhead that this entails, it also makes things even worse because it causes too many protection faults per epoch. It also makes a "lazy diffing" (i.e., postpone diffing until the block is actually accessed by another processor) approach infeasible, since almost all blocks are always needed by each epoch. It is important to note that this is not a characteristic of the chosen parameters (see Figure 9 and 11) or even, to a large extent, of the chosen benchmark (see Figure 12). A smaller block size would reduce or even eliminate[7] this problem.

One solution might be to try making the overhead factor $f \approx 1$. One approach would be to pick a small enough (i.e., $N_{blk} \approx N_{acc}$), *fixed* block size. However, if we want to use the virtual memory mechanism, then it is impossible in practice to reduce the block size, as explained in section 4.1 and 4.2. An alternative is to modify memory instructions. Software DSMs such as Blizzard [SFL+94] and Shasta [SGT96] employ executable editing to instrument loads and stores. However, this still incurs a penalty for *each* read and write to shared memory. Blizzard also resorted to custom memory boards to improve performance.

Clearly, the optimal approach would be to use a variable "block" size, always equal to each object. This technique was essentially used in Orca [BKT92], which is a parallel, object-oriented programming language. The compiler and runtime environment keep track of accesses to each object on each processor in the system. In general, each memory object must be identified at the proper granularity and have read/write dirty bits that are properly updated whenever it is accessed. In software TLS systems this technique is not used; thread pipelining [TY96] is probably the one that comes closest (see also section 5.1).

4.3.2 Copying Overhead

Even if we reduce the overhead factor to $f = 1$, this does not mean that the overhead has been eliminated, but rather that it has been limited. When an epoch is committed, we eventually must perform intra-processor copying in order to update each local image of shared memory. If $f = 1$, then we are copying exactly the words that have been modified and no more.

As a side note, it is not clear how to do fast (i.e., essentially constant-time, if this scheme is to succeed) random tagging of a block (as read and/or write dirty), while also being able to retrieve the list of tagged blocks in time proportional to the size of this list and not the size of the shared region.

It would be desirable to avoid copying altogether, by using actual shared memory. However, threads running speculatively should not modify the same data without some control. One approach would be to use wait/wakeup synchronization flags, rather than read/write-dirty bits per block or object. This is a method for *runtime serialization* of *potentially* dependent memory accesses, rather than *speculation*. Such an approach is used in thread pipelining [KL98], which is further explained in section 5.1. The compiler must *statically* identify where dependences may occur and insert the appropriate instructions to tag store addresses at runtime and perform the necessary checks and synchronization between threads. Addresses are tagged in a stage separate from the main thread body. Thus, execution

[7] For instance, if blocks were 4 bytes large, i.e. exactly the size of a single array element.

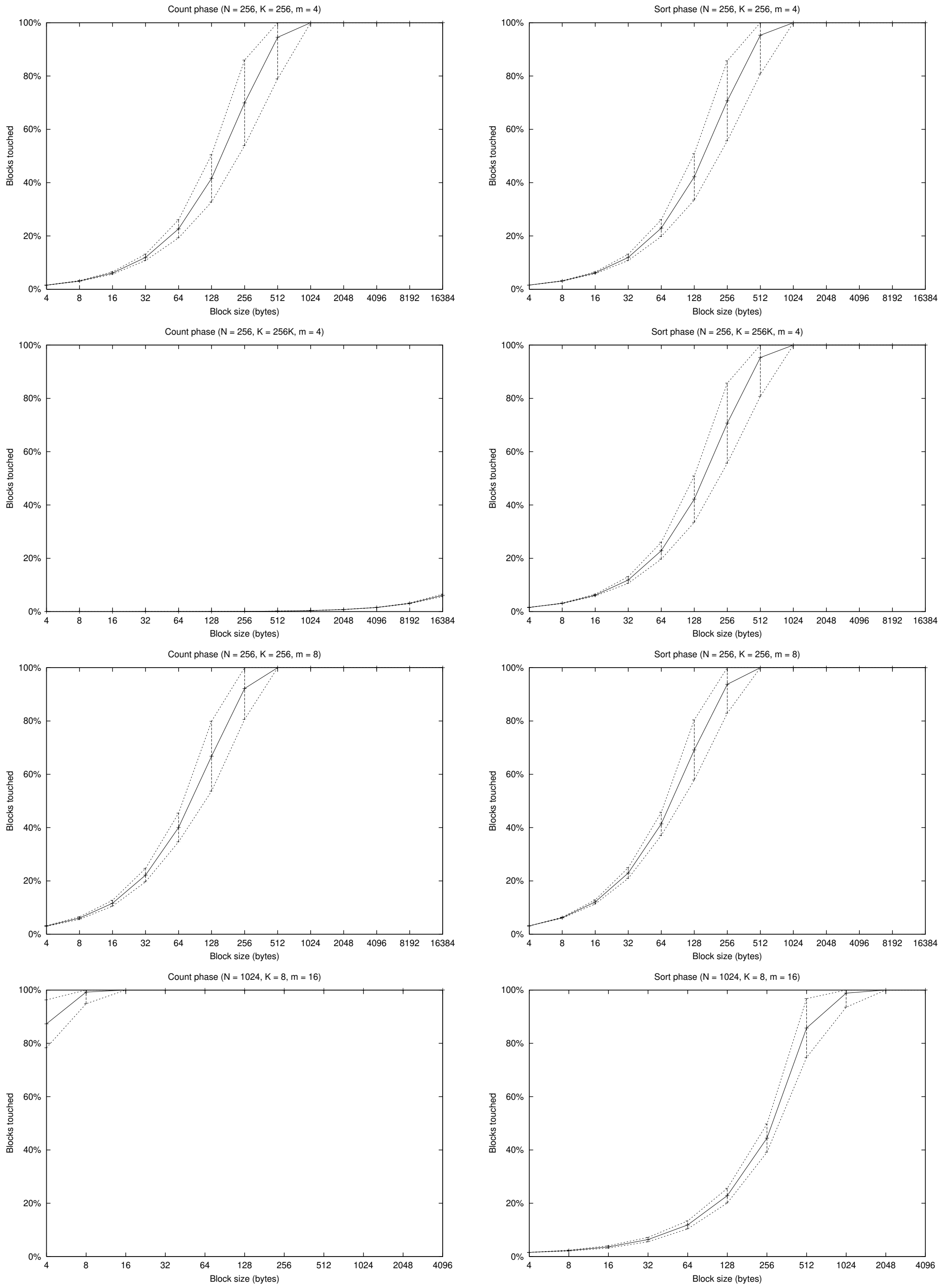


Figure 11: Fraction of blocks touched per epoch for various block sizes. We show he average over all epochs and the standard error. Once again, for the sorting phase we consider only the sorted output array, not the key count array.

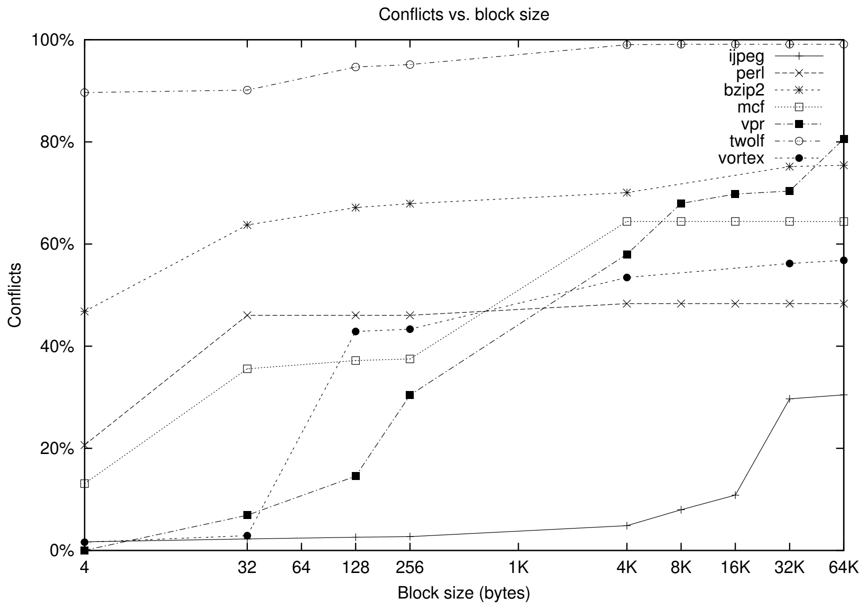


Figure 12: Conflicts versus block size for certain SPECint95 and SPEC CINT2000 benchmarks. If we make the simple and optimistic assumption of linear speedup in proportion to the conflicts (i.e., $t_{par} = ct_{ser} + (1 - c)t_{ser}/N_{CPU}$), then an increase in conflicts from 2%–4% to 30%–40% results in a factor of 1.8–2 slowdown with 4 CPUs. The results were obtained using the benchmarks and dependence model from [SCZM00]. Although the our software TLS system was designed for different applications with coarser-grain epochs in mind, these data provide further evidence about the behavior of programs from standard benchmarks that have been shown to benefit from TLS.

may be serialized, even when dependences do not actually exist at runtime. Also, tagging and tag checking must occur at a granularity greater than per-word, in order to limit overheads.

5 Related Work

In the past there has been a significant amount of work on TLS (although the term is not consistently used), both in hardware as well as software. The main focus has been on hardware-based schemes, but a few software schemes have been developed. However, the latter have not been shown to be generally applicable and effective. In the next sections we provide an overview of various systems.

5.1 Software Approaches

The earliest work on software-based speculation involves run-time parallelization of loops that iterate over arrays (i.e., fixed-size, discrete sets of variables). These techniques are helpful when the array access pattern does not allow compile-time parallelization. Essentially all techniques in this class involve first doing a “quick” execution of the loop, in order to compute the array access pattern by performing the least possible work. This requires some extra storage space, whose size is of the same order as the array size. The basic idea is to keep track at which iteration each array element is accessed.

Based on the access pattern, there are several possible actions [RP95]. The simplest technique is to simply execute the iterations in parallel, if and only if there are no cross-iteration dependences at all. This can be improved upon by using *privatization* to eliminate certain dependences. For instance, write-after-read dependences can be handled by writing first into private storage and copying the values into the actual array after parallel execution has finished[8].

The above techniques have also been extended to exploit any available parallelism. Based on the access pattern, it is possible to discover which iterations are mutually independent of each other. All mutually independent *consecutive* iterations are grouped together. All iterations in the same group (or *wavefront*) are executed in the same parallel phase.

The main disadvantage of the above class of techniques is that they are applicable only to arrays. They cannot be extended to more complicated, pointer-based structures, which are the most common ones in non-scientific applications.

A more recent approach is that of *thread pipelining* [KL98]. It is inspired by the superthreaded architecture [THA$^+$99] (see also section 5.2). Each thread (roughly corresponding to epochs in our scheme) is divided into a number of pipeline stages:

- Continuation stage: This computes all variables necessary to fork the next thread.
- Target-store address generation (TSAG) stage: Computes the addresses of write operations upon which other threads *may* be data dependent. These addresses are forwarded to memory buffers of successor threads.
- Computation stage: This performs the main bulk of the work. Any reads from target store addresses of previous threads are synchronized via a flag allocated for each such address.
- Write-back stage: During this stage, the thread writes any privatized data back into the shared memory.

The above mechanism is used to support data speculation. Control speculation can be supported by speculatively forking threads. These threads write in private memory and transfer the results into shared memory only when they complete and speculation has proved successful.

By limiting privatization as much as possible, the overhead is small enough to achieve speedups. This approach relies on the compiler to identify potential sources of dependences and insert the appropriate instructions in the TSAG and computation stages. Thread pipelining is has been proven successful in a number of benchmarks and is an interesting approach. However, its performance when there is significant imbalance among the various stages or when there is a large number of potential dependence sources has not been thoroughly investigated.

Finally, recent work [Dev00] explores ways to exploit *stride predictability*. The main idea of this approach is to detect at runtime if memory accesses indeed follow a predictable pattern, by executing the first few iterations and examining the addresses of memory instructions. If the addresses differ by a constant amount, then *all* future accesses are assumed to be stride-predictable. If the initial addresses and strides are such that there are no conflicts[9], then the code is executed in parallel. This approach supports *data* speculation. One of its main strengths is that it completely avoids privatization (although some copying needs to be done to restore original values upon failed speculation). However, as the

[8] Only those array locations that are the source of such dependences need to be privatized.

[9] This requires computing the GCD of the strides. Even this turns out to be an expensive operation to perform *at runtime* and often eliminates any performance gains. Thus, an approximate solution is used in practice.

authors admit, in many cases results at least as good can be obtained by compile-time parallelization methods. There is no conclusive proof yet whether this approach is useful in speculation on arbitrary pointer-based structures in real applications.

5.2 Hardware Approaches

The Multiscalar architecture [SBV95] was the first complete work in supporting fine-grained thread-level parallelism. The compiler partitions the program into *tasks*, each task being a partial walk of the control-flow graph (e.g., a complete loop iteration). Each task is assigned to one of multiple processing units. The assignment is done in a cyclical queue fashion and tasks are retired in order. Processing units communicate via a ring, in order to detect data dependence violations in registers and/or memory, as well as forward values to tasks later in the queue.

Our group's STAMPede project has been developing hardware support for TLS [SCM97, SM98, SCZM00]. This work was the main source of inspiration for our software-based approach. The main focus has been on a generic single-chip multiprocessor, where each processor has its own primary data cache and all processors on the same chip physically share a secondary cache. We have extended the instruction set to provide new instructions which enable software to manage TLS, to use the caches to buffer speculative state, and to extend the cache coherence scheme to detect data dependence violations. This approach has been successful in a wide variety of applications; a more detailed description can be found in [SM98].

The Hydra [HWO98, ONH+96] single-chip multiprocessor uses an approach that is very similar to ours. The main difference is that it relies on more support from the hardware. This achieves lower software overheads, at the expense of more hardware. It is not clear that the chosen trade-off provides any significant performance benefits.

Recently, the Multiscalar group has introduced the *speculative versioning cache* [GVSS98]. This design is also based on a modified cache coherence scheme, similar to snooping bus-based cache protocols.

The superthreaded architecture [THA+99] is also another design for thread-level speculation. This was the inspiration for the thread-pipelining model (described in section 5.1), which employs essentially the same approach.

5.3 Hybrid Approaches

It is not entirely clear what "hybrid" means, since most systems need some form of software support. We use the term to signify systems that rely *equally* on software and at least some special hardware. For instance, an incarnation of Blizzard [SFL+94] (see also section 5.4) which uses custom memory boards instead of code instrumentation may be considered a hybrid system.

Among systems for thread-level speculation, the one that probably best fits this category is the one described in [ZRT98]. This is essentially an extension of the work in [RP95]. However, instead of checking whether speculation was successful in software after the entire loop has been executed, this system relies on extensions to the cache coherence scheme to detect violations as early as possible. If there is a dependence violation, then software restores memory contents and re-executes the loop sequentially. The scope of this system is relatively more limited than other hardware approaches, although the original design has been recently extended [ZRT99].

5.4 Software DSMs

One of the first successful software DSM systems is Treadmarks [KCDZ94]—and its offspring, CVM [Kel97]. Treadmarks uses the virtual memory hardware mechanism and employs a lazy release consistency model as well as a multiple-writer protocol with diffing to achieve good performance. Shasta [SGT96] and Blizzard [SFL+94] were developed later. They rely on code instrumentation instead of the virtual memory hardware. Sirocco [SFH+98] investigates the potential of using cheap hardware shared-memory nodes (typically bus-based, with 2-4 CPUs each) in order to build a larger scale software DSM.

6 Discussion

We have described our approach for supporting thread-level speculation (TLS) purely in software, using the virtual memory mechanism. This work derives from previous, successful work in software DSM systems. However, we found that this success does not translate to software TLS.

The unavoidably large block sizes cause most serious problems. When using the virtual memory mechanism, the block size is equal to the size of a page. Typical page sizes are 4–16 Kb, although permissible ranges are usually 4–64 Kb and pages larger than 16 Kb are becoming more common with newer, 64-bit architectures (for instance, Intel IA64 and AMD x86-64) and larger address spaces. This poses no problems to the primary goal of virtual memory, particularly since the availability of large physical memories reduces the impact of internal fragmentation problems. However, such large page sizes have very serious consequences for speculation support.

The random memory access patterns of good candidate applications for speculation can easily cause false sharing problems. Besides needlessly increasing communication overheads, false sharing also causes false violations, which result in serialized execution.

Besides the overheads related to block size, the need for privatization is the other factor that significantly affects performance. In fact, speculation support also requires the need to keep multiple versions of memory contents, in order to undo writes when speculation fails. This can be avoided to some extent in *special* forms of speculation. However, some form of buffering of memory operations has to be done eventually, since we may have to undo them.

The issues raised above (i.e., false sharing/violations, privatization and undo) are the two main reasons that the techniques from software DSMs are not as successful in software TLS. As we have already mentioned, the main goal of TLS is to enable automatic parallelization of programs for which data dependence analysis and, consequently, data partitioning is very difficult or impossible to do at compile-time. Therefore, applications that can benefit from TLS typically exhibit random memory access patterns. In parallel applications that run on DSMs, a significant effort has been already placed in partitioning the data among processors. This results in better locality. Furthermore, software DSMs do not have to deal with buffering multiple versions of memory contents and undos.

There are alternative approaches that do not rely on the virtual memory system. We must clarify that we are mostly concerned with general support for TLS and not other limited forms of speculation. There is significant evidence that such support provides large benefits, especially for non-scientific applications [SM98, HWO98] that use irregular, pointer-based structures.

The most promising alternative seems to be based on instrumenting memory instructions. This technique can support relatively small block sizes, in the range of 128–256 bytes. These may be sufficient for some applications, but for others smaller block sizes may be necessary (see Figure 12).

However, there is a trade-off between small block sizes and fast tagging. Ideally, we would like tagging to cost only a few instructions, while lookup of dirty block sets take time directly proportional to the size of the sets and not to the size of speculative shared memory. This is a difficult goal to achieve. Furthermore, this approach still does not solve the problem of maintaining local images. Thus, even though instrumentation-based approaches have a much better chance to succeed than VM-based approaches, it is not entirely clear that they would be sufficient.

Finally, there is the option of added hardware support. There is already evidence that such an approach works well. Based on our observations, the minimum necessary extra support should allow tracking memory references at a small granularity and also allow fast retrieval of dirty sets. Furthermore, maintenance of multiple versions of memory location contents and fast undo of write operations can be done efficiently in hardware. This is necessary to avoid the problems associated with privatization. The most natural location for adding this support is the cache subsystem. In fact, most current approaches rely on modified cache coherence protocols. Of course, these approaches are sensitive to cache line size, but the effects are much less pronounced at those block size ranges. Unlike increasing page size, increasing cache line size also has negative effects on overall performance due to an increase in conflict misses. Therefore, there isn't much reason to question the survival of hardware-based TLS mechanisms in the future.

Acknowledgments

We would like to thank J. Gregory Steffan for his detailed feedback during the preparation of this manuscript, Chris Colohan for all his help with the simulation tools and Christos Faloutsos for his suggestions about the presentation of the material.

References

[BKT92] Henry E. Bal, M. Frans Kaashoek, and Andrew S. Tannenbaum. Orca: A language for parallel programming on distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[Dev00] Srikrishna Devabhaktuni. Softspec: Software-based speculative parallelism via stride prediction. Master's thesis, MIT/LCS, April 2000.

[GVSS98] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[HWO98] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *The 1994 Winter USENIX Conference*, 1994.

[Kel97] Pete Keleher. CVM: The Coherent Virtual Machine. Technical report, Department of Computer Science, University of Maryland, November 1997.

[KL98] Iffat H. Kazi and David J. Lija. Coarse-grained speculative execution in shared-memory multiprocesors. In *Proceedings of the International Confenrece on Supercomputing*, pages 93–100, July 1998.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Computer*, 28(9):690–691, September 1979.

[ONH+96] Kunle Olukotun, Basem Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[RP95] Laurence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 218–232, June 1995.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[SCM97] J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Architectural support for thread-level data speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[SCZM00] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 1–12, June 2000.

[SFH+98] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, and David A. Wood. Sirocco: Cost-effective fine-grain distributed shared memory. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 40–49, October 1998.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–252, October 1996.

[SM98] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[THA+99] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. The Superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9), September 1999. Special Issue on Multithreaded Architectures.

[TY96] Jenn-Yuan Tsai and Pen-Chung Yew. The Superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, October 1996.

[ZRT98] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[ZRT99] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative reduction parallelization and optimization in DSM multiprocessors. In *Proceedings of the 1st Workshop on Parallel Computing for Irregular Applications*, January 1999. Held in conjunction with HPCA-5.