Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor

Dean M. Tullsen
Dept. of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
`tullsen@cs.ucsd.edu`

Jack L. Lo, Susan J. Eggers, Henry M. Levy
Dept. of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
{`jlo,eggers,levy`}`@cs.washington.edu`

UCSD CSE Technical Report CS98-587
UW CSE Technical Report UW-CSE-98-06-02

June 1998

Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor

Abstract

Existing multiprocessor synchronization mechanisms are relatively heavyweight, due in part to the level of the memory hierarchy (typically main memory) at which threads must synchronize. Multithreaded processors, on the other hand, have the potential to significantly reduce synchronization cost, because threads share the processor simultaneously and can synchronize using processor-internal state.

This paper proposes and evaluates new synchronization schemes for a simultaneous multithreaded processor. We present a scalable mechanism that permits threads to cheaply synchronize within the processor, with blocked threads consuming no processor resources. The basic mechanism, blocking *acquire* and *release*, is a hardware implementation of traditional software synchronization abstractions, implemented with a thread-shared hardware *lock box*.

This study shows that a multithreading processor with efficient synchronization enables parallelization at a granularity an order of magnitude finer than required on conventional parallel machines with memory-based synchronization. When combined with *lock release prediction*, which reduces the overhead of restarting a blocked thread, acquire-release gains an additional improvement of 40%. Overall, we show that these substantial improvements in synchronization cost enable parallelization of code that could not be effectively parallelized using traditional synchronization; performance of these codes showed up to a two-fold improvement over single-threaded versions of the previously unparallelizable code.

1 Introduction

The performance and style of a multiprocessor's synchronization mechanisms determine to a large extent the granularity of parallelism that can be exploited on that machine. For this reason, substantial effort has gone into studying synchronization techniques for shared-memory multiprocessors [14, 4], with the objective of supporting fine-grained parallelism. Despite this work, synchronization and communication are still relatively costly on contemporary multiprocessors; this high cost is inherent in the hardware levels at which inter-thread synchronization and communication must occur. As a result, compilers and programmers must decompose parallel applications in a relatively coarse-grained way in order to reduce synchronization overhead.

This paper examines *fine-grained* synchronization on a simultaneous multithreaded (SMT) processor — a processor in which the CPU can issue instructions from multiple threads in a single cycle [19, 18]. Multithreaded processors, such as SMT, provide an opportunity to greatly *decrease* synchronization cost, because the communicating threads are *internal* to a single processor. While previous work has shown the benefits of SMT on parallel workloads [6, 13], those studies relied on traditional synchronization mechanisms, ignoring the potential advantages (and problems) of synchronizing in an SMT CPU.

A simultaneous multithreaded processor differs from a conventional multiprocessor in several crucial ways that influence the design of SMT synchronization:

- Threads on an SMT processor compete for all fetch and execution resources each cycle. Synchronization mechanisms (e.g., spin locks) that consume *any* shared resources without making progress, even for a few cycles, can impede other threads.

- Data shared by threads is held closer to the processor, e.g., in the L1 cache or perhaps even in the internal store buffer; therefore, communication is dramatically faster between threads. Synchronization must experience a similar increase in performance to avoid becoming the bottleneck.

- Hardware thread contexts on an SMT processor share physical registers (e.g., renaming registers) and functional units. This opens the possibility of communicating synchronization and/or data much more effectively than through memory.

An SMT CPU can greatly reduce the cost of synchronization, permitting significantly finer-grained parallelism than is possible on conventional MPs. However, inappropriate choice of synchronization mechanisms can cause harmful contention for heavily shared resources on an SMT processor.

In this paper we present a new mechanism for fine-grained synchronization in SMT processors and compare the performance of that mechanism with traditional synchronization techniques. Our results show an order of magnitude improvement in the granularity of parallelism available, relative to conventional shared-memory multiprocessors. We also describe and evaluate optimizations that reduce the synchronization cost even further, such as speculatively restarting blocked threads through lock-release prediction. The optimizations increase the synchronization efficiency by another 50%. We demonstrate that this fine-grained synchronization is sufficiently lightweight that it permits parallelization of new codes that could not previously be parallelized. Several parallel codes that slow down (relative to uniprocessor execution time) by a factor of 10 or more on a traditional parallel processor experience speedups in excess of 2 with an SMT processor and efficient synchronization.

The paper is organized as follows. The next section describes existing synchronization mechanisms and presents our new mechanism for synchronizing in an SMT processor. Section 3 describes the methodology for our simulation-based experimental studies. Section 4 defines a metric for synchronization efficiency and uses the metric to evaluate several synchronization mechanisms. Section 5 proposes and evaluates optimizations to the baseline SMT synchronization. Section 6 looks at specific examples of loops which only become worthwhile to parallelize with the fast synchronization available on a multithreaded processor. We show that in these cases the speed of the synchronization mechanism becomes critical to performance. Section 7 discusses related work, and Section 8 concludes.

2 Synchronization Mechanisms

In this section, we begin with a brief description of existing synchronization mechanisms. We then present our goals for synchronization in SMT processors and describe the new mechanism that we evaluate in this paper, and how it meets the goals. We also discuss the shortcomings of existing schemes (as candidates for SMT synchronization) relative to these goals.

2.1 Review of Existing Synchronization Schemes

A number of different synchronization mechanisms exist in commercial or research multiprocessors, both conventional and multithreaded. Most common are *Spin Locks*, such as `test-and-set`. While logically `test-and-set` modifies memory, optimizations typically allow the spinning to take place in the local cache to reduce bus traffic [3, 14]. More recently, *Lock-Free synchronization* has been widely studied [9, 11] and is included in modern instruction sets, e.g., the DEC Alpha's load-locked (`ldl_l`) and store-conditional (`stl_c`) [17] (collectively, LL-SC). Rather than achieve mutual exclusion by preventing multiple threads from entering the critical section, lock-free synchronization prevents more than one thread from successfully writing data and exiting the critical section. A store-conditional only completes successfully (returning a success value in a register) if no store or `stl_c` to the same address from another thread occurred between the `ldl_l` and the `stl_c`. Otherwise, software retries the critical section. Lock-free synchronization is effective when it succeeds, because the synchronization is embodied in the memory access itself — the only extra overhead is the check for failure.

The multithreaded processors on the Tera [2] and Alewife [1] machines rely on *full/empty (F/E) bits* associated with each memory block. F/E bits allow memory access and lock acquisition with a single instruction, such as `read_if_full/set_empty`, where the full/empty bit acts as the lock, and the data is returned only if the lock succeeds. If it fails, the access stalls at the memory or is returned to the CPU to be retried.

Another multithreaded processor, the M-Machine [8, 10], attaches full/empty bits to registers. In the M-Machine, multiple threads execute on each of several clusters. Each cluster has a unique register file with full/empty bits attached to all registers. Synchronization among threads on different clusters or even within the same cluster can be achieved by a cluster-local thread explicitly setting a register to empty, after which a write to the register by another thread will succeed, setting it to full. A read of that register by the local thread will not succeed until the register has been filled. Keckler et al. [10] provide a good description of these mechanisms in a study with similar goals to ours. They show that register-based communication mechanisms enable finer-granularity threading than memory-based techniques; by exploiting fine-grain threads, a three-cluster MAP chip uses parallel procedure calls to achieve speedups up to 1.6 over sequential applications running on a single cluster.

We evaluate similar register-based communication mechanisms to theirs; however, we do not consider these to be sufficient in themselves for SMT synchronization. Several differences between the M-machine and SMT result in the different directions taken by our two studies: (1) the M-machine is a message-passing MP, so its synchronization mechanisms do not have to scale to a shared-memory MP; (2) no single MAP execution unit is shared by all threads, so the M-Machine cannot use execution-unit-based synchronization (e.g., the lock-box mechanism we will describe); and (3) the M-machine processor is itself a multiprocessor, so Keckler et al.'s study focuses on enabling speedup through synchronization between processors (clusters), running applications that are parallelizable on traditional MPs. In contrast, our focus is on obtaining speedup on an SMT *uniprocessor*, running applications traditionally considered to be serial.

2.2 Goals for SMT Synchronization

This section identifies the desired goals for SMT synchronization. These goals are motivated by the special properties of an SMT processor, as described in Section 1. Given these properties, synchronization on an SMT processor should be:

1. *High Performance.* On an SMT, even traditional synchronization will be faster than on traditional multiprocessors, because the locks can stay in the lowest levels of the cache hierarchy. We wish to take full advantage of SMT's thread-shared resources to make it as fast as possible. High performance implies both high throughput and low latency.

2. *Resource-conservative.* Most parallel machines have spin locks of some type, which require repeated reads or writes. Lock-free synchronization also includes a potential repeated retry of a lock. On SMPs, Anderson, et al. [3] favor queueing locks and `test-and-test-and-set` over `test-and-set` locks; in that situation, it is acceptable for a blocked process to waste processor-private resources (e.g., local cpu, local cache), but not shared resources (e.g., bus and memory bandwidth). In SMT the same principle applies, except that virtually *everything* in the processor is shared; therefore, to be resource-conservative, stalled threads must use *no* processor resources.

3. *Deadlock-free.* We must avoid introducing new forms of deadlock. SMT shares the instruction scheduling unit among all executing threads and could deadlock if a blocked thread fills the instruction queue, preventing the releasing instruction (from another thread) from entering the processor.

4. *Scalable.* We expect that SMT processors will exist in both multiprocessor (multi-SMT) and uniprocessor configurations. To simplify programming, the same primitives should be usable to synchronize threads on different processors and threads on the same processor. However, the *performance* of the mechanisms may not be the same in both cases, because this would preclude taking advantage of fast intra-SMT synchronization when we can.

5. *Easy to build.* Work on simultaneous multithreading has been aimed at mainstream processors. One implication is that we would prefer to allow standard memory designs and commodity memory parts.

None of the existing synchronization mechanisms presented in Section 2.1 meets all of these goals when used in the context of SMT. For example, spin locks are not resource-conservative and can badly degrade the performance of non-spinning threads on an SMT processor (for this reason, we do not simulate spin locks in our experiments). Similarly, lock-free synchronization is not resource-conservative if the store-conditional repeatedly fails. However, lock-free synchronization has an important advantage, namely, that it prevents a thread from being de-scheduled (e.g., due to I/O or page fault) while holding a critical lock. Mechanisms such as load-locked and store-conditional are not incompatible with SMT design, and we examine them in our experiments.

Full/empty bits on memory fail our easy-to-build criteria: they require additional bits in memory and a complex memory controller. F/E memory bits also fail to meet our high-performance goal: they provide high synchronization throughput but also high latency (because passing a lock explicitly from one thread to another requires a round-trip to main memory). We consider register full/empty bits as in the M-Machine for the SMT processor, but only in conjunction with SMT-based synchronization (described in Section 2.4). Taken alone, register F/E bits are not a sufficient mechanism for synchronization: they do not meet our scalability criteria and do not handle unordered access, particularly where the producer does not know the identity of the consumer. Finally, unlike memory-address-based synchronization, register-address-based synchronization severely limits the number of synchronization locations that can be live at once.

2.3 An SMT Mechanism for Blocking Synchronization

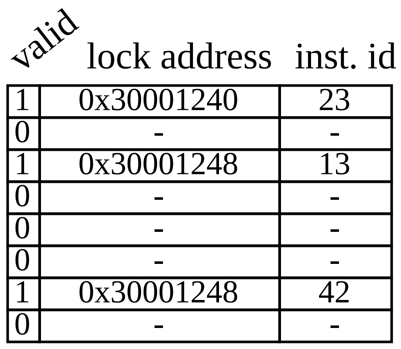
As noted above, none of the existing synchronization mechanisms meet all of our criteria. Here we present a design for SMT synchronization that does. It uses hardware-based blocking locks. A thread that fails to acquire a lock blocks and frees all resources it is using except the hardware context itself. A thread that releases a lock upon which another is blocked causes the blocked thread to be restarted. The actual primitives consist of three instructions:

`Acquire(lock)` – This instruction acquires a memory-based lock. The instruction does not complete execution until the lock is successfully acquired; therefore, it appears to software like a `test-and-set` that never fails. The result of the `acquire` never needs to be tested and the instruction never needs to be retried in software.

`Release(lock)` – This instruction writes a zero to memory if no other thread in the processor is waiting for the lock; otherwise, the next waiting thread is unblocked and memory is not altered.

`Try-Acquire(lock)` – This instruction tries to acquire a memory-based lock. Unlike `Acquire`, it always completes immediately, but does not always succeed. `Try-Acquire` allows the program to prevent blocking in cases where it has alternate work to do. (This paper does not examine any code that would take advantage of this primitive.)

These primitives look familiar, not because they are common hardware primitives, but because they are common software interfaces to synchronization, typically implemented with spinning locks. For the SMT processor, we choose to implement them directly in hardware. We assume no hard-



valid	lock address	inst. id
1	0x30001240	23
0	-	-
1	0x30001248	13
0	-	-
0	-	-
0	-	-
1	0x30001248	42
0	-	-

Figure 1: The structure of the lock box. In this example, three threads are blocked, two waiting for the same lock.

ware support for fast barrier synchronization, using software based on these primitives instead. Extending our hardware implementation for faster barriers would be straightforward.

The synchronization instructions are implemented with a small processor structure associated with a single functional unit (e.g., one of the load/store FUs is also the synchronization FU). The structure, which we call a *lock-box* (Figure 1), has one entry per context (per hardware-supported thread). Each entry contains: the address of the lock, a pointer to the lock instruction that blocked, a valid bit, and optionally, bits for maintaining the order of threads blocked on the same lock (not shown in the figure).

When a thread fails to acquire a lock (a read-modify-write of memory returns a nonzero value), the lock address and instruction identifier are stored in that thread's lock-box entry, the valid bit is set, and the thread is flushed from the processor after the lock instruction. When another thread releases the lock, hardware performs an associative comparison of the released lock address against the lock-box entries. On finding the blocked thread, the hardware allows the original lock instruction to complete, allowing the thread to resume, and invalidates the blocked thread's lock-box entry. A release for which no thread is waiting is written to memory.

The `acquire` instruction is restartable. Because it never commits if it does not succeed, a thread that is context-switched out of the processor while blocked for a lock will always be restarted with the program counter pointing to the `acquire` or earlier, so the retry of the `acquire` will be automatic. The lock-box entry will always be cleared on a context switch.

The three primitives above meet all our synchronization criteria. They are resource-conservative, because a waiting thread consumes no execution resources. They allow fast transfer of locks when the consumer is waiting for the lock and even conserve memory bandwidth, because the lock can be passed in many cases without updating memory.

Flushing the blocked thread from the instruction queue (and pre-queue pipeline stages) is critical to preventing deadlock. If a blocked thread continues to have access to the fetch unit and the instruction queue, it would fill the shared instruction queue, preventing the process that holds the lock from making progress. The mechanism needed to flush a thread is essentially the same mechanism used after a branch misprediction on an SMT processor.

The entire mechanism is scalable (i.e., it can be used between processors), as long as a release in one processor is visible to a blocked thread in another. This could be accomplished by a periodic retry of failed acquires. Any release that leaves a lock free updates memory; therefore, a processor in a multiprocessor system can retry (in hardware) a blocked acquire at regular or increasingly-long intervals. Not all threads have equal access to a lock in this scheme, as a very heavily-contended lock may remain on a single processor for an extended time. More complex schemes might provide fairer access to locks, but this simple scheme provides the best performance for highly-contended locks. Given a choice, it gives the lock to the thread that will use it most efficiently (i.e., the one that can obtain the lock in the fewest cycles, and which probably has the protected data structures in its first-level cache).

2.4 SMT Blocking Locks With Register Full/Empty Bits

While full/empty registers alone do not meet all of our criteria, we may find them useful for specific situations where multiprocessor scalability is not an issue. Full/empty registers have several performance advantages, improving both communication and synchronization speed. They allow shared data to be in registers, and thus eliminate memory operations; they save instructions, because the synchronization and the data transfer coincide; and, unlike `acquire`, `release` and the generic `test-and-set`, they do not impose ordering constraints on surrounding memory operations.

In this study, we assume that registers are initially empty, and that a full/empty bit is only accessed for two new instructions:

```
read_full rs, rd
write_fill rs, rd(rt)
```

`Read_full` moves the value in `rs` to `rd` only if `rs`'s full/empty bit is set to full. Both `rs` and `rd` belong to the executing thread's own register file. When the instruction completes, `rs`'s full/empty bit is set to empty. If `read_full` cannot complete immediately (the register is empty), it and its thread block in the same manner as a failed `acquire`. `Write_fill` writes the value in the executing thread's register `rs` to register `rd` in thread `rt`'s register file, setting that register's full/empty bit to full. This instruction will block if `rd` is already full. The value in register `rt` selects the id of the destination thread, allowing for 2^{64} thread names. The hardware must be able to map a software thread ID to a hardware context. If the thread ID does not match a running context, the hardware traps to software. Adding the ability to access other threads' registers adds no new data paths to the SMT architecture, which already assumes a shared physical register file.

2.5 Summary of Synchronization Mechanisms

In this section we stated our goals for synchronization and compared a number of synchronization alternatives in light of those goals. We presented a new mechanism for synchronization in an SMT processor that relies on a hardware *lock-box* unit to maintain the state of a blocked thread's context without consuming processor resources. The rest of the paper focuses on the goal of fast

Active list	64 entries/context
Instruction queue	32 integer entries, 32 floating point entries
Functional units	6 integer (of which 4 can perform loads/stores); 4 FP
Architectural registers	32*8 integer, 32*8 floating point
Renaming registers	100 integer, 100 floating point
Processor pipeline	9 stages, 2 each for register reads/writes
Branch prediction	256-entry BTB, 2K x 2-bit PHT, gshare
Function return stack	12-entries

Table 1: Processor parameters used in the simulator

Cache	Size	Line Size (bytes)	Miss latency to next level (cycles)	Associativity	Fill Latency (cycles)
onchip L1 I cache	32KB	64	10	DM	2
onchip L1 D cache	32KB	64	10	DM	2
onchip L2 cache	256KB	64	15	4-way	4
off-chip L3 cache	2MB	64	125		8

Table 2: Cache parameters used in the simulator

synchronization. We will show how the alternative mechanisms compare in performance on an SMT processor, how synchronization speed changes the way we parallelize code, and how crucial synchronization speed is to overall performance.

3 Methodology

Using a detailed trace-driven simulator, we compare several alternative hardware and software synchronization mechanisms on an SMT architecture. The simulator executes unmodified Alpha object code using emulation-based, instruction-level simulation techniques. It models the execution pipelines, memory hierarchy, TLBs, and branch prediction logic of an SMT processor, including throughput and latency constraints at all levels of the memory hierarchy and incorrect path execution following a branch misprediction. Table 1 provides more details of the processor model; Table 2 contains memory system parameters.

We simulate the ICOUNT.2.8 instruction fetch scheme from [18], which fetches up to eight instructions from up to two threads each cycle. ICOUNT.2.8 is the most aggressive fetch scheme proposed for SMT, and is much more tolerant of synchronization mechanisms that are *not* resource conservative than less aggressive schemes.

Our analysis is based on a synthetic synchronization efficiency benchmark, on the two primary loops from the *espresso* benchmark from the SPEC92 suite, and on three Livermore loops (C version) that are not currently parallelized by the SUIF compiler. We compiled all with *gcc* with full optimization, inserting synchronization directly into the source code with macro-defined *asm()*

statements. For the espresso loops, the execution times reported are the combined execution times of the first 200 invocations of each loop.

Synchronization instructions constrain the issuing of memory operations. We assume implicit memory barriers around synchronization instructions, i.e., `acquire`, `release`, `ldl_l` and `stl_c` do not issue until all previous stores have completed, and loads cannot pass an `acquire` or `stl_c`.

Because synchronization takes place inside the processor, where many instructions reach the execute stage in a speculative state, we must ensure that synchronization in one thread is not visible to other threads until it is no longer speculative. In general, we assume that synchronization operations have no effect on machine state until they retire.

We also assume that an `acquire` access goes into the store buffer at execution. When the instruction commits, a read-modify-write is performed to the local cache (for a cache hit). Load operations are allowed to issue the following cycle if the lock succeeds; because the load will take two cycles to get to the execute stage, the performance impact will be minimal even if we assume that the read-modify-write takes more than one cycle. We assume that a value written through a `write_fill` operation is available for reading the cycle after the `write_fill` retires.

4 Characterizing the Efficiency of Synchronization

In this section we define an efficiency metric for synchronization and use it to evaluate the speed of different synchronization schemes. Our vehicle for expressing the metric is a loop containing a mix of loop-carried dependent (serial) computation and independent (parallel) computation, representing a wide range of loops or codes with different mixes of serial and parallel work. By modifying the amount of independent work, we vary the ratio of serial code to parallel code from mostly serial to mostly parallel. Our efficiency metric is the *ratio* of parallel-to-serial computation at which the parallel version of the loop begins to outperform a single-threaded version. [1]

The simple efficiency benchmark is shown in Figure 2. The amount of independent computation (work that contains no loop-carried dependences in the `i` loop) is varied by enclosing it in a loop that iterates between 1 and 128 times. Each iteration of the independent computation loop does a load (a cache hit), a floating-point multiply and a floating-point add. The result is then added to `A[i]` in the critical section.

We first ran this test using mechanisms that synchronize in different levels of the memory hierarchy, to confirm our assertion that tightly-coupled multithreaded processors (like SMT) can exploit parallelism in a profoundly different way than conventional multiprocessors. The synchronization schemes we simulated are:

- *Single-thread.* This is the serial version of the loop. It helps us define the break-even point.
- *SMT-block.* This is the base SMT synchronization, with blocking acquires.

[1]If mechanism S1 requires half as much parallel computation to surpass the serial version as mechanism S2, we say it is twice as efficient.

```
single-threaded:
for (i = 0; i < N; i++)
    A[i+1] = A[i] + independent_computation()
```

```
parallelized:
(for each thread)
for (i = threadId; i < N; i += numThreads)
    temp = independent_computation()
    acquire(lock[threadId])
    A[i+1] = A[i] + temp
    release(lock[nextId])
```

Figure 2: Serial and parallel version of our synchronization efficiency test

- *SMT-LL-SC*. This scheme uses the lock-free synchronization currently supported by the Alpha. To implement the ordered access in the benchmark, the acquire primitive is implemented with load_locked and store_conditional, as given in [17], and the release is a store instruction.

- *SMP-**. These each use the same primitives as SMT-block, but force the synchronization (and data sharing) to occur at different levels in the memory hierarchy (i.e., the L2 cache, the L3 cache, or memory). This mimics the performance of systems with contexts less tightly coupled than on an SMT. SMP-Mem represents the synchronization and communication performance of a typical shared-memory multiprocessor; SMP-L3 (off-chip cache) represents a tightly-coupled cluster of processors sharing an off-chip cache; and SMP-L2 represents a single-chip multiprocessor with a shared secondary cache. The execution model for all SMP-* simulations is SMT.

Figure 3 compares the synchronization speeds of the different schemes. The figure shows that synchronization within a processor is more than an order of magnitude more efficient than synchronization in memory. For example, the break-even point for parallelization is about 5 computations for SMT-block, and over 80 for memory-based synchronization. Thus, an SMT processor will be able to exploit opportunities for parallelism that are an order of magnitude finer than those needed on a traditional multiprocessor, even if the SMT processor is using existing synchronization primitives (e.g., LL-SC).

While the single-chip multiprocessor (SMP-L2) is only half as efficient as SMT-block, it is much closer to SMT-block than to the memory-based multiprocessor (SMP-MEM). Therefore, many of our arguments about new opportunities for parallelism extend to the single-chip multiprocessor model [15] as well.

Figure 3 shows that blocking synchronization outperforms lock-free synchronization; however, for this benchmark the primary factor is not resource waste due to spinning, but the *latency* of the synchronization operation. This is made clear by examining the critical path through successive iterations of the `for` loop. This critical path is shown in Figure 4 for LL-SC and `acquire-release` primitives, assuming that the next thread is waiting for passed data when the current thread produces it. When the independent computation is small and performance is dominated by the loop-carried calculation, the critical path becomes the locked (serial) region of each iteration.

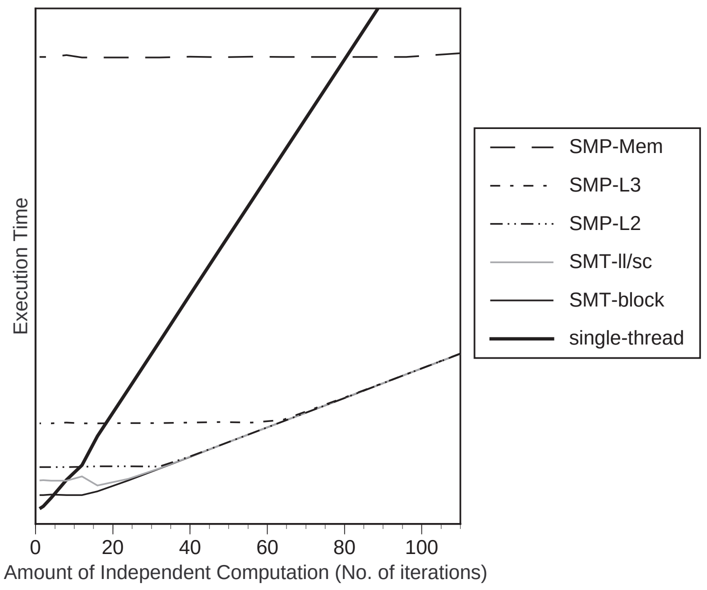
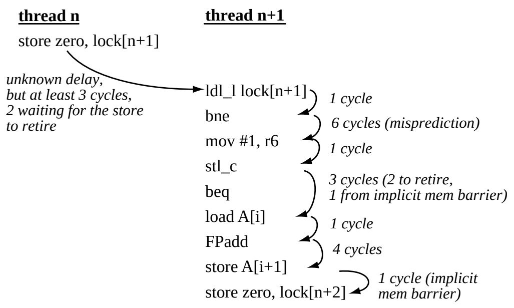
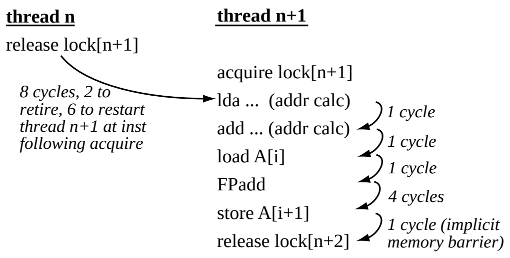


Figure 3: The performance of various synchronization configurations.



(a) Load_locked/Store_conditional



(b) Acquire-Release

Figure 4: The critical path through one iteration of the benchmark with (a) `load_locked` and `store_conditional`, and (b) `acquire` and `release`.

For the lock-free synchronization, the critical path is at least 20 cycles per iteration. A key component is the branch misprediction penalty when the thread finally acquires the lock. Lock-free synchronization is less likely to achieve the minimum delay along the critical path, because of (1) the unknown delay before the waiting thread retries the load, and (2) the inter-thread contention for resources throughout execution (i.e., no concurrent threads are blocked while this thread is executing).

For blocking SMT synchronization, the critical path through the loop is 15 cycles. The minimum time is dominated by the restart penalty (the process of getting a blocked thread's instructions back into the processor).

4.1 Summary

In this section we used a simple synchronization efficiency metric to evaluate the limits of the alternative synchronization schemes. The results show that fine-grained synchronization, when performed close to the processor, changes the available granularity of parallelism by an order of magnitude. We will examine this potential in more detail later using common program loops. Our analysis of the critical path also opens the possibility to further optimization, which we examine next.

5 Tuning the Performance of Synchronization

The previous section demonstrated that fast synchronization enables a new class of applications to be parallelized, namely those with cross-iteration loop dependences requiring tight synchronization. Unlike current parallel programs, these applications put the latency of synchronization squarely on the critical performance path.

This section examines optimizations to our baseline synchronization mechanism with blocking acquires. These include (1) speculative restart of blocked threads based on lock-release prediction and (2) judicious use of full-empty registers, both with and without a similar speculative restart mechanism. These optimizations cut the critical path through the efficiency metric loop in half and decrease the parallelism cross-over point by a factor of five.

5.1 Faster Synchronization Through Speculative Restart

The restart penalty for a blocked `acquire` assumes that the blocked thread is not restarted until the corresponding `release` instruction is retired. As previously noted, it then takes several cycles to reload the blocked thread's instruction state into the processor. While the `release` cannot perform until it retires (or is at least guaranteed to retire), it is possible to restart the blocked thread earlier; the processor could then begin fetching from the thread and even execute instructions that are not dependent on the `acquire`. (In any case, all memory operations must wait until the `release` is performed.)

It is possible to perform the speculative restart in several places. Restarting in the execute stage would more accurately select the right thread to restart; restarting in the fetch or decode stage allows much earlier restarts. In Figure 5, we show the results of speculatively restarting a blocked thread as soon as the `release` is seen by the decode unit. We assume that a history based on thread ID and PC is used to predict which thread will be released by a given instruction. [2]

Speculatively restarting a thread before the releasing instruction retires reduces the critical path to nine cycles (see Figure 6a), lowering the break-even point for parallelization to about 3 iterations of the independent loop (Figure 5).

[2]We abort a speculative restart when the release instruction that caused it is squashed, even if there have been other release instructions that have since been fetched. Also, since we are performing the speculative restart in the decode stage, a thread that blocks after the instruction that will release it is in the processor will miss the opportunity for an speculative restart.

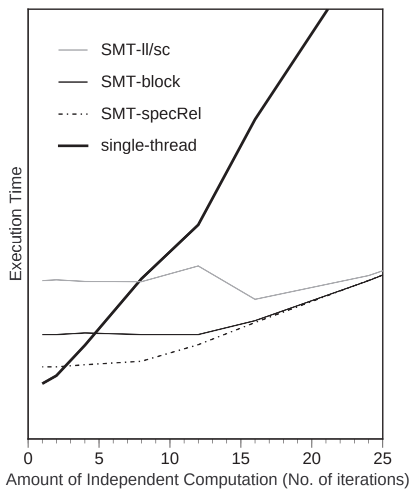
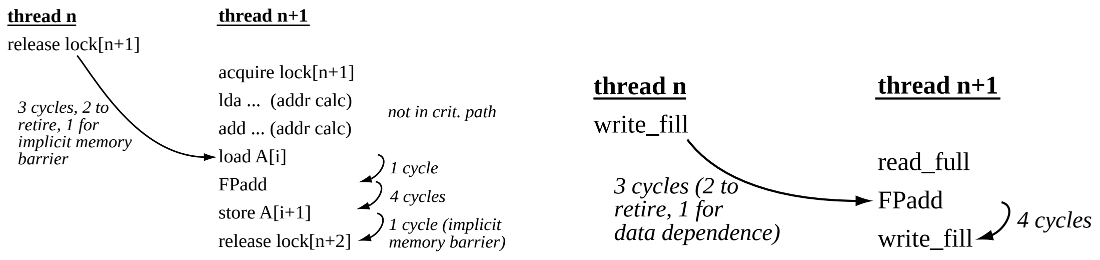


Figure 5: The performance of speculative restart.



(a) Acquire-Release (b) Full/Empty Registers

Figure 6: The critical path through one iteration of the benchmark with speculative restart, for (a) the acquire-release version of the loop, and (b) full-empty registers.

5.2 Faster Synchronization Through Full/Empty Registers

In this section we consider relaxing our scalability criteria to add M-machine-style full/empty registers to our proposed SMT synchronization primitives, in an attempt to further reduce the cost of both synchronization and communication.

The per-thread full/empty-register code for our efficiency metric is implemented as follows:

```
r4 = nextId
for (i = threadId; i < N; i += numThreads)
    temp = independent_computation()
    read_full r2, r3
    r3 = r3 + temp
    write_fill r3, r2 (thread r4)
    A[i+1] = r3
```

Notice that (compared to Figure 2) the load is no longer needed and the store is moved out of the critical section of the loop. This scheme can still suffer the restart penalty, because we must block a thread when the synchronization operation fails. This happens when the `read_full` finds the register empty (as is the common case for this loop when the independent computation is small) or when the `write_fill` finds the register full (which never happens for this loop). We can also apply speculative restart to this loop, and the results for both cases are shown in Figure 7.

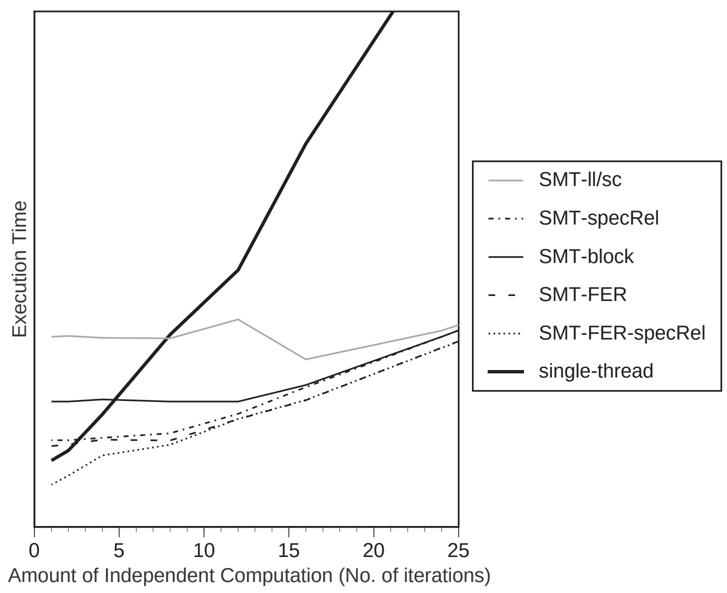


Figure 7: The execution time performance of the benchmark with full/empty registers.

Adding F/E registers reduces the critical path through the computation for the case of the speculative restart to 7 cycles (Figure 6b). The break-even point is now less than a single iteration of the independent computation loop (Figure 7).

By reducing the critical path through a simple loop-carried dependence to under ten cycles for the `acquire/release` primitives, and to 7 cycles for full/empty registers, SMT has the potential to parallelize code that requires finer-grained parallelism than can be supported by a conventional multiprocessor. This code only becomes parallelizable with fast synchronization. The next section demonstrates the existence of such code.

6 Case Studies in Parallelization With Fast Synchronization

Any program traditionally regarded as (either implicitly or explicitly) parallel has achieved performance in the face of relatively high-cost synchronization. Such applications should run well with any of the low-latency synchronization mechanisms we have considered. However, we have argued that efficient fine-grain parallelism will create an entirely new class of "parallel" programs. We claim that both the size of that new class, and the exact performance they see will in large part be determined by the particular speed of the synchronization mechanism. This section investigates that claim.

We examine five loops that a standard parallel compiler would not parallelize: the two most important loops in *espresso* and three of the Livermore loops. These loops are significant exactly because of the compiler's assumption that parallelizing would not be worthwhile, due to synchronization and communication costs. We attempt to parallelize these loops using our fine-grained SMT synchronization mechanism, and report the success stories and one less-than-successful effort.

Both cases are useful in understanding the impact of very-fine-grained synchronization on an SMT processor.

6.1 Espresso

For the SPEC92 benchmark *espresso* and the input file ti.in, 22% of the execution time is spent in the routine *massive_count*. This routine is primarily composed of two loops, both of which contain heavy cross-iteration dependences. We describe and evaluate the parallelization of those loops.

The first loop is a doubly-nested loop with both ordered and un-ordered dependences across the outer loop. The inner loop is actually completely parallel; however, the inner-loop parallelism is small (for some of the input files, the iteration count is always 1) and unbalanced (the iteration often ends with an initial compare to zero). Thus, we chose to parallelize the outer loop and synchronize the dependences.

The first of two dependences in this loop is a pointer that is incremented and compared against zero as the exit condition. In parallelizing this loop, the pointer is updated and passed to the next thread; when it becomes null, it is passed (but not updated) around all the threads an extra time to allow each to see the exit condition.

The second dependence is a large array of counters which are conditionally incremented based on individual bits of a series of calculated values. Protecting the entire counter array with a single lock eliminates parallelism, but protecting individual counters has a high overhead. We achieved better performance with a scheme in the middle that protects ranges of eight counters with a single lock. Since incrementing is associative, access to the counters need not preserve the serial ordering.

Figure 8 (Loop 1) shows that all SMT versions perform well. There is only a small gain with full/empty registers — the counters are not a good match for F/E registers, and so we only use them to pass the pointer more quickly. For both, there is little performance difference with speculative restart, because collisions on the counters are unpredictable.

With LL-SC, the pointer must use the spinning `acquire-release` mechanism; however, the atomic incrementing of counters is tailor-made for lock-free synchronization. Each counter is protected individually by LL-SC's ability to do lock-free atomic updates, which reduces collisions. However, there are still enough collisions to create some wasted computation, and to confuse the branch predictors for the branch on the success-fail result. LL-SC also suffers from competition from threads waiting at the barrier at the end of the computation.

Memory-based synchronization clearly cannot overcome the high cost of synchronization and communication.

The second component of *massive_count* is a single loop primarily composed of many nested `if` statements with some independent computation. The loop has four scalar dependences for which we must preserve original program order (shared variables are read and conditionally changed). There are two more updates to shared structures that need not be ordered.

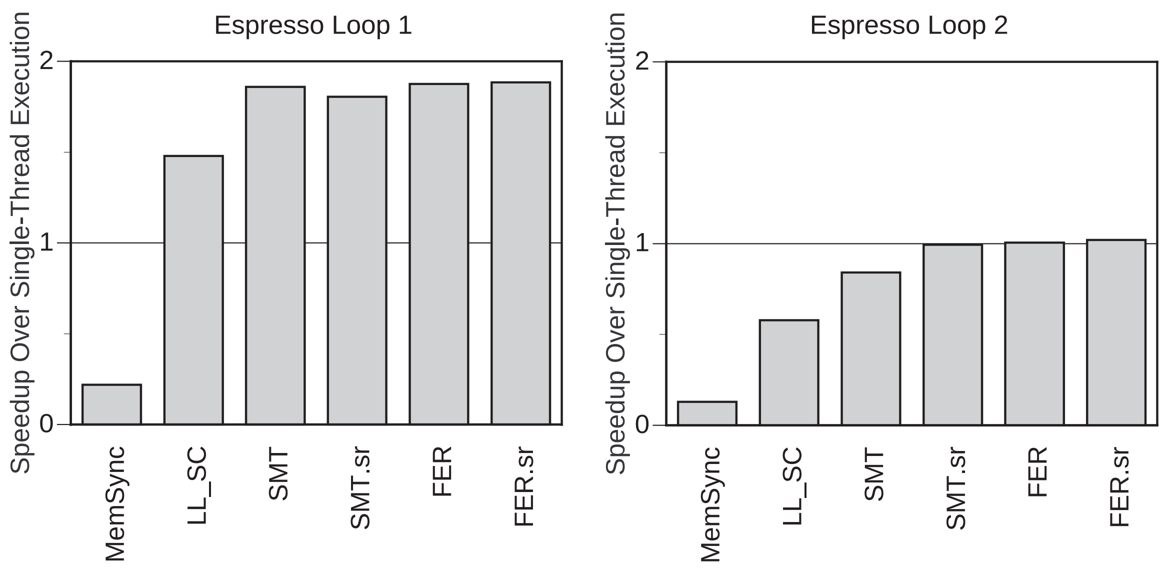


Figure 8: The speedup over single-thread execution on the two loops in massive_count from *espresso* for various synchronization alternatives.

The most frequent path through the loop updates none of the shared-ordered variables; therefore a single lock around all shared variables produced better performance than protecting each shared variable with an individual lock.

The performance of the blocking synchronization was disappointing (Figure 8, loop 2). Further analysis uncovered several factors that limited performance. First, the single-thread loop already has significant ILP, close to four instructions per cycle, leaving little room for improvement. Second, most of the shared variables are in registers in the single-thread version, but must be stored in memory for blocking synchronization. Third, the blocking locks constrained the efficient scheduling of memory accesses in the loop. Fourth, the branches are highly correlated from iteration to iteration, allowing our default branch predictor to do very well for serial execution; however, much of this correlation was lost when the iterations went to different threads, resulting in a 3-4X increase in branch mispredictions.

F/E Registers allowed shared variables to be kept in registers, and eliminated the memory-access ordering constraints — even the unordered accesses were updated in-order with F/E mechanisms. However, performance was still limited by the high ILP and branch mispredictions. The performance gain over blocking synchronization alone is significant, and moves (but barely) into positive ground. The speculative-restart results are nearly identical, because the full/empty code was already blocking much less often.

With LL-SC, the ordered accesses had to be protected in the same manner as the blocking synchronization, but with software acquire and release. For the unordered variables, they were each updated atomically using LL-SC. The overall performance was poor due to spinning for contested locks.

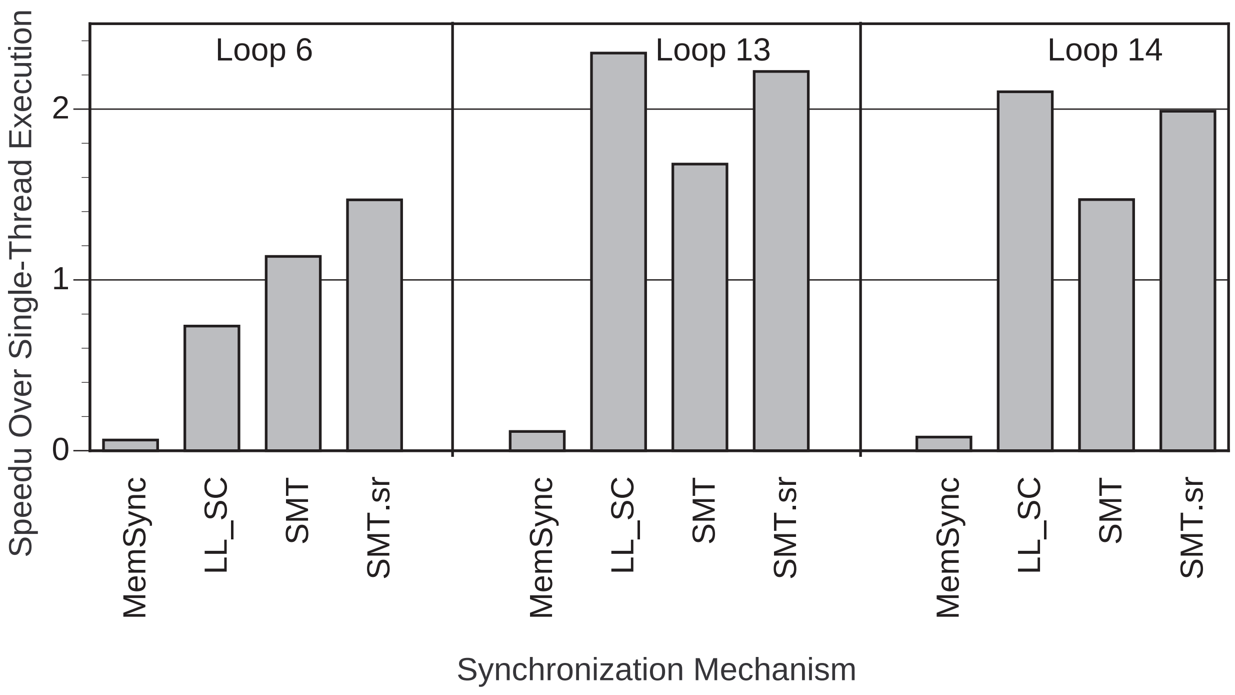


Figure 9: The performance of synchronization configurations for three of the Livermore loops executing with 8 threads.

6.2 Livermore Loops

Livermore loops 6, 13, and 14 all have cross-iteration dependencies, and consequently the SUIF compiler [7] chose not to parallelize them. These loops have a reasonable amount of code that is independent, however, and should be amenable to parallelization, if fine-grained parallelism (and synchronization) were available.

Loop 6 is a doubly-nested loop that reads a triangular region of a matrix. The inner loop accumulates a sum in `w[i]`. Since `i` is invariant within the inner loop, we can privatize `w[i]`, parallelize the inner loop, and sum the values at the end through a reduction. However, the next iteration of the outer loop reads all previously calculated `w[i]`'s, so we require a barrier following each inner loop to prevent read-after-write violations on `w`. Because of the triangular nature of the loop, some of the inner loops do very few iterations, making the cost of a reduction and barrier critical to loop performance. We assume no hardware support for barriers in this example, but combine the barrier and the reduction in software.

Figure 9 (Loop 6) shows the result for eight threads. While parallelization of this loop does not make sense on a conventional multiprocessor, it becomes profitable with standard SMT synchronization, and more so with speculative restart support. LL-SC performs poorly due to the high overhead of threads spinning at the barrier.

Loop 13 has only one potential cross-iteration dependence, the incrementing of an indexed array, which thus happens in an unpredictable, data-dependent order. Although it is not necessary to preserve the order of these updates, we chose to do so (using a mechanism very similar to the code in Section 4), because (1) these loops are very uniform, so the performance cost of ordering was small, and (2) the performance of the speculative restart prediction is better with a forced ordering.

The results of parallelizing *Loop 13* are shown in Figure 9. *Loop 13* achieves more than double the throughput of single-thread execution with SMT synchronization and the speculative restart optimization. Here LL-SC also performs well, since the only dependence is a single unordered atomic update. This can be done with a single `ldl_l`, `stl_c` pair and a check for success (although conversion from integer to floating point complicates it slightly). Collisions are few because each address is protected individually.

Loop 14 actually contains three loops, which appear to be a single loop that is split to decrease ILP-reducing dependencies within a single iteration. (In fact, for the single-thread version, performance is slightly better than with the three loops fused.) Our parallel version fuses these loops, which maximizes the amount of parallel code available to execute concurrently with the final serial portion. The serial portion is two updates to another indexed array, so once we have fused the loops of *Loop 14*, it looks much like *Loop 13*, and the performance results confirm this.

Since the dependences in these loops are all either barrier-based or unordered updates, we did not apply any full-empty register optimizations.

We made no heroic efforts to get better memory-based synchronization performance on these loops. Slightly better performance could have been achieved with very large, padded arrays of locks (much larger than the data they were protecting) on two of the loops; but performance would still have been extremely poor, due to sharing (both true and false) on the data. Consequently, the point remains that traditional multiprocessors cannot achieve the performance of multithreaded processors on loops with cross-iteration dependences.

One aspect of parallelization that is not simplified on an SMT machine is the choice of the number of threads to use; however, the freedom to make this choice provides a new opportunity for both performance optimization and conservation of resources (so that they can be used by other executing programs).

6.3 The Right Number of Threads

In a multiprocessor that is otherwise idle, the decision of how many threads to create is rarely a difficult one. In an SMT processor, however, the tradeoffs are more complex. On one hand, there is little benefit in creating more threads after a shared execution resource is saturated; and there is a greater probability of suffering a restart penalty as the number of threads increases. On the other hand, more threads contribute more independent instructions to hide latencies. In this section, we examine the implications of varying the number of threads used for the same Livermore loop computations.

Figure 10 shows that different numbers of threads were appropriate for different loops and different SMT synchronization mechanisms. *Loop 6*, which only achieved a small speedup with 8 threads, had linear speedup with two. Performance on *Loops 13* and *14* without speculative restart was equivalent (but higher) with all numbers of threads beyond 2. With speculative restart,

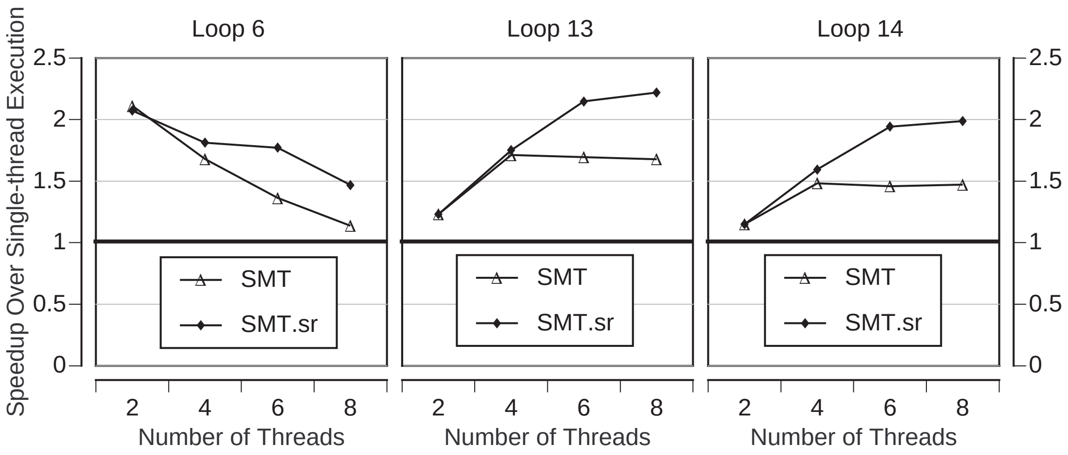


Figure 10: Synchronization performance with varying number of threads and blocking synchronization (with and without speculative restart).

speedups increased with the numbers of threads. Overall, the results indicate that (1) it is important to choose the level of parallelism carefully, rather than simply use the maximum available, because the best choice can significantly improve over serial performance (double in the best cases of these loops); and (2) the optimal choice is dependent on the loop and the underlying synchronization mechanism. Thus, optimal performance requires the decision to be made separately for each loop in the program.

6.4 Summary

For the five loops examined, none of which could be parallelized on a conventional multiprocessor, fast synchronization enabled significant parallel speedups on four and marginal speedup on one. In each case, parallelization created execution paths that made the speed of the synchronization critical to the performance of the code, as all were sensitive to the exact mechanism used.

Of the five loops considered, two had unordered single-item atomic updates as their only dependence. Not surprising, LL-SC handled those well, but in both cases, SMT synchronization was within 5%. However, if there was at least one access to data that had a forced ordering (the other three loops), SMT synchronization outperformed LL-SC by an average of 67%.

7 Related Work

Section 2.1 described other multithreaded architectures and multithreaded synchronization mechanisms. Other work which has impacted or is related to this study follows.

There have been several papers that describe the design of simultaneous multithreading processors [18] and analyze their performance on parallel [12, 6, 13] and multiprogrammed [18, 6] workloads. The first group [12, 6, 13], which examined simultaneous multithreading with a more traditional multiprocessor workload, and in light of more traditional parallel compiler transformations is the most relevant to this work. These studies showed that an SMT processor achieves significant speedups on code that is parallelized for multiprocessors, but that assumptions about certain compiler transformations appropriate for a multiprocessor are not necessarily appropriate for SMT.

The Tera MTA architecture [2] supports fine-grain parallelism via multithreading and full-empty memory synchronization. Despite having a higher-latency synchronization mechanism than what we propose, the Tera processor and compilation system exploit fine-grain synchronization among cooperating threads, confirming our thesis that a different style of code generation is required for multithreaded processors.

Keckler, et al., [10] describe mechanisms for efficient synchronization, communication, and thread creation in the Multi-ALU Processor (MAP). There are several key differences between their mechanisms and ours, and between the target applications of the two studies. These result from the two different machine models, as discussed in Section 2.1. Their mechanism does not scale in our environment, and our acquire-release is not possible in theirs. Our focus is on uniprocessor speedup on traditionally serial applications.

Pai, et al. [16] describe a synchronization buffer for multiprocessors of single-threaded processors. The synchronization buffer is an off-chip structure which holds lock addresses from executing lock instructions. It takes responsibility for retrying the lock so that software does not have to loop. They do not otherwise block the thread, nor do they associate releases with locks in their structure.

Bradford and Abraham [5] propose hardware-implemented semaphores which block a thread waiting for a semaphore. They compare this scheme with spin-waiting and OS-implemented blocking synchronization, running SPLASH applications on a fine-grain multithreaded processor with perfect caches.

8 Summary

For existing traditional parallel programs, any reasonably-fast synchronization mechanism on a simultaneous multithreading processor is likely to be sufficient. However, that approach misses the tremendous opportunity of hardware multithreading, which is to enable a style of parallelism not available on other architectures — very-fine-grain parallelism.

We have shown that very-fine-grain parallelism greatly expands our definition of what is "parallelizable." The speed and efficiency of synchronization is critical to this goal. We have proposed a new synchronization mechanism, tailored specifically for an SMT processor, that (1) maximizes synchronization efficiency by ensuring that threads waiting on synchronization release all shared resources quickly to other threads, and (2) minimizes synchronization latency by using lock-release prediction to resume blocked threads with no fetch or restart delay.

The fine-grain acquire-release mechanism enables significant speedups (in excess of 2.0) on applications that cause significant slowdowns when parallelized on traditional parallel machines. The lock-release prediction speeds application performance by as much as 29%.

References

[1] A. Agarwal, J. Kubiatowicz, D. Kranz, B-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, June 1993.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[3] T.E. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):1–16, January 1990.

[4] T.E. Anderson, E. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12), December 1989.

[5] J.P. Bradford and S. Abraham. Efficient synchronizatin for multithreaded processors. In *Proceedings of the Workshop on Multithreaded Execution Architecture and Compilation in conjunction with HPCA-4*, January 1998.

[6] S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, R.L. Stamm, and D.M. Tullsen. Simultaneous multithreading: A foundation for next-generation processors. *IEEE Micro*, September 1997.

[7] M.W. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12), December 1996.

[8] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee. The M-Machine multicomputer. In *28th Annual International Symposium on Microarchitecture*, November 1995.

[9] M. Herlihy. A methodology for implementing highly concurrent data objects. In *2nd ACM Symposium on Principles and Practices of Parallel Programming*, pages 197–206, March 1990.

[10] S.W. Keckler, W.J. Dally, D. Maskit, N.P. Carter, A. Chang, and W.S. Lee. Exploiting fine-grain thread level parallelism on the MIT multi-alu processor. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[11] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 11(1):806–811, November 1977.

[12] J.L. Lo, S.J. Eggers, J.S. Emer, H.M. Levy, S.S. Parekh, R.L. Stamm, and D.M. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, pages 322–354, August 1997.

[13] J.L. Lo, S.J. Eggers, H.M. Levy, S.S. Parekh, and D.M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[14] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1), February 1991.

[15] B.A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *23nd Annual International Symposium on Computer Architecture*, pages 67–77, May 1996.

[16] V.S. Pai, P. Ranganathan, S.V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ilp processors. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1996.

[17] R.L. Sites and R.T. Witek. *Alpha AXP Architecture Reference Manual, Second Edition*. Digital Press, 1995.

[18] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23nd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[19] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.