Proceedings of the 5th International Symposium on High Performance Computer Architecture, January 1999.

Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor

Dean M. Tullsen
Dept. of Computer Science and Engineering
University of California, San Diego
tullsen@cs.ucsd.edu

Jack L. Lo
Transmeta Corporation
Santa Clara, CA
jlo@transmeta.com

Susan J. Eggers, Henry M. Levy
Dept. of Computer Science and Engineering
University of Washington
{eggers,levy}@cs.washington.edu

Abstract

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. We also introduce the concept of lock release prediction, which gains an additional improvement of 40%. Overall, we show that these improvements in synchronization cost enable parallelization of code that could not be effectively parallelized using traditional techniques.

1. Introduction

The performance of a multiprocessor's synchronization mechanisms determine the granularity of parallelism that can be exploited on that machine. Synchronization on a conventional multiprocessor carries a high cost due to the hardware levels at which synchronization and communication must occur (e.g., main memory). As a result, compilers and programmers must decompose parallel applications in a 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 [10, 9]. Multithreaded processors 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, 7], those studies relied on traditional synchronization mechanisms, ignoring the potential advantages (and problems) of synchronizing in an SMT CPU.

A simultaneous multithreading processor differs from a conventional multiprocessor in several crucial ways that influence the design of SMT synchronization: (1) 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, can impede other threads. (2) Data shared by threads is held closer to the processor, i.e., in the thread-shared L1 cache; therefore, communication is dramatically faster between threads. Synchronization must experience a similar increase in performance to avoid becoming a bottleneck. (3) Hardware thread contexts on an SMT processor share functional units. This opens the possibility of communicating synchronization and/or data much more effectively than through memory.

This paper presents a scalable synchronization mechanism for SMT 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*. The lock box is a simple hardware mechanism that enables the transfer of memory-based locks between threads on the same processor in just a few cycles. This latency can be further reduced by a new technique called *lock-release prediction*, which minimizes the cost of restarting a blocked thread.

Our results show an order of magnitude improvement in the granularity of parallelism made available with this new synchronization, relative to synchronization on conventional shared-memory multiprocessors. We demonstrate that it is sufficiently lightweight to permit parallelization of new codes that could not previously be parallelized.

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.

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 `test-and-set` modifies memory, optimizations typically allow the spinning to take place in the local cache to reduce bus traffic. More recently, *Lock-Free synchronization* has been widely studied [4] and is included in modern instruction sets, e.g., the DEC Alpha's load-locked (`ldl_l`) and store-conditional (`stl_c`) (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.

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, where the full/empty bit acts as the lock, and the data is returned only if the lock succeeds.

The M-Machine [5], attaches full/empty bits to registers. Synchronization among threads on different clusters or even within the same cluster is 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. Keckler et al. [5] provide a good description of these mechanisms in a study with similar goals to ours. We do not consider register full/empty bits to be sufficient in themselves for SMT synchronization. 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 multicomputer, so its synchronization mechanisms do not have to scale to a shared-memory MP; and (2) no single execution unit is shared by all threads, so the M-Machine cannot use execution-unit-based synchronization. The CRAY X-MP also introduced shared registers for synchronization, but with the synchronization bits decoupled from the data registers.

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.* High performance implies both high throughput and low latency. Full/empty bits on main memory provides high throughput but high latency.

(2) *Resource-conservative.* Both spin locks and lock-free synchronization consume processor resources while waiting for a lock, either retrying or spinning, waiting to retry. To be resource-conservative on a multithreaded processor, 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.* The same primitives should be usable to synchronize threads on different processors and threads on the same processor, even if the performance differs. Full/empty bits on registers are not scalable.

None of the existing synchronization mechanisms presented in Section 2.1 meets all of these goals when used in the context of SMT.

2.3. A Mechanism for Blocking SMT Synchronization

Here we present a design for SMT synchronization that meets our criteria. 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 two 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.

`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.

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 implement these primitives directly in hardware.

The synchronization instructions are implemented with a small processor structure associated with a single functional unit. The structure, which we call a *lock-box*, 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 and a valid bit.

When a thread fails to acquire a lock (a read-modify-write of memory returns nonzero), the lock address and instruction id are stored in that thread's lock-box entry, 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 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.

Flushing a blocked thread from the instruction queue (and pre-queue pipeline stages) is critical to preventing deadlock. The mechanism needed to flush a thread is the same mechanism used after a branch misprediction on an SMT processor. We can prevent starvation of any single thread without adding information to the lock box simply by always granting the lock to the thread id that comes first (including wrap-around) after the id of the releasing thread.

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. We discuss several ways that this could be accomplished in [11].

3. Characterizing Synchronization Efficiency

Using a detailed trace-driven simulator, we compare several alternative 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 8-issue SMT processor. More details of the processor and memory system model are given in [11].

In this section we define an efficiency metric for synchronization and use it to evaluate the speed of different synchro-

```
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 1. Our synchronization efficiency test

nization schemes. Our vehicle for expressing the metric is a loop containing a mix of loop-carried dependent (serial) computation and independent (parallel) computation, that can represent a wide range of loops or codes with different mixes of serial and parallel work (Figure 1). Our efficiency metric is the *ratio* of parallel-to-serial computation at which the threaded version of the loop begins to outperform a single-threaded version. 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 and a floating point add. The result is then added to `A[i]` in the critical section.

In Figure 2 *Single-thread* is the performance of the serial version of the loop, which defines the break-even point. *SMT-block* is the base SMT synchronization with blocking acquires using the lock-box mechanism. *SMT-ll/sc* 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 and the release is a store instruction. *SMP-** each use the same primitives as SMT-block, but force the synchronization (and data sharing) to occur at different levels in the memory hierarchy. This mimics the synchronization and communication performance of systems with contexts less tightly coupled than on an SMT, such as a typical shared-memory multiprocessor (SMP-Mem), a tightly-coupled cluster of processors sharing an off-chip cache (SMP-L3), and a single-chip multiprocessor with a shared secondary cache (SMP-L2). Synchronization within a processor is more than an order of magnitude more efficient than synchronization in memory. 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., the lock-free LL-SC).

However, blocking synchronization does outperform lock-free synchronization; for this benchmark the primary factor is not resource waste due to spinning, but the *latency* of the synchronization operation. We observed the critical path through successive iterations of the `for` loop when the independent

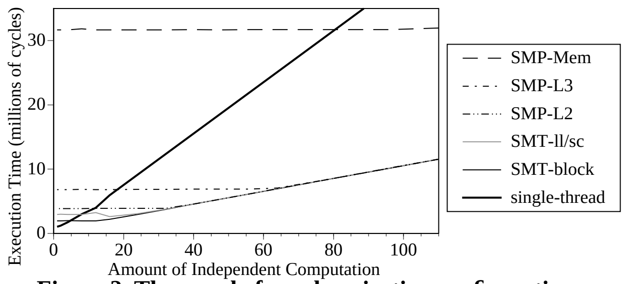

Figure 2. The speed of synchronization configurations.

computation is small and performance is dominated by the loop-carried calculation. In that case the critical path becomes the locked (serial) region of each iteration. For the lock-free synchronization, the critical path is at least 20 cycles per iteration [11]. A key component is the branch misprediction penalty when the thread finally acquires the lock and the LL-SC code stops looping on lock failure. For blocking SMT synchronization, the critical path through the loop is 15 cycles. This time is dominated by the restart penalty (to get a blocked thread's instructions back into the CPU).

In summary, 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. Those loops, like our efficiency benchmark, will put the speed of synchronization on the critical performance path. The following optimization reduces that critical path length.

Faster Synchronization Via Speculative Restart. The restart penalty for a blocked `acquire` assumes that the blocked thread is not restarted until the corresponding `release` instruction retires. It then takes several cycles to fetch the blocked thread's instruction stream into the processor. While the `release` cannot perform until it retires (or is at least guaranteed to retire), it is possible to speculatively restart the blocked thread earlier; the thread can begin fetching and even execute instructions that are not dependent on the `acquire`.

In Figure 3, we show the results of speculatively restarting a blocked thread as soon as the `release` is seen by the decode unit. A history based on thread ID and PC is used to predict which thread will be released by a given instruction. Speculatively restarting a thread before the releasing instruction retires reduces the critical path from 15 cycles to nine cycles (when the prediction is correct), lowering the break-even point to about 3 iterations of the independent loop (Figure 3).

4. Case Studies in Parallelization With Fast Synchronization

Any program traditionally regarded as parallel has achieved performance in the face of relatively high-cost synchronization, and should run well with any of the synchronization mechanisms we have considered. However, efficient fine-grain parallelism will create a new class of "parallel" programs. The size of that new class, and their exact performance, will in large part be determined by the speed of the synchronization

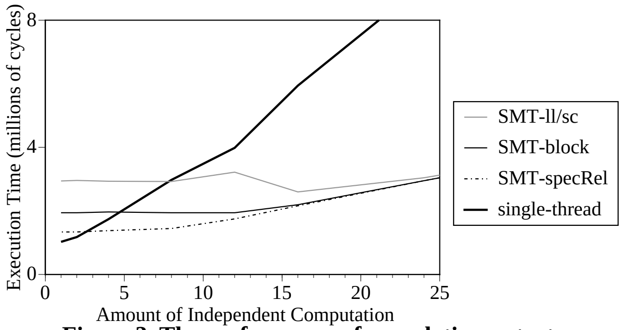

Figure 3. The performance of speculative restart.

mechanism. Because this type of fine-grain parallelism is not available on existing systems, we have access to no compiler which can identify and transform code appropriately. Thus we have identified some potential code by hand to use as case studies. Although we hand-insert synchronization in these loops, we do so at the source level, and do not otherwise alter the code significantly (except for obvious compiler transformations, like a reduction in one case). For much more detail on how the loops were parallelized, see [11].

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.We attempt to parallelize these loops across 8 threads using our fine-grained SMT synchronization mechanism, and report the success stories and one less-than-successful effort.

Espresso. For the SPEC benchmark *espresso* and the input file ti.in, a large part of the execution time is spent in two loops in the routine *massive_count*.

The first loop is a doubly-nested loop with both ordered and un-ordered dependences across the outer loop. The first dependence in this loop is a pointer that is incremented until it points to zero (the loop exit condition). The second dependence is a large array of counters which are conditionally incremented based on individual bits of calculated values.

Figure 4 (Loop 1) shows that both SMT versions perform well; however, there is little performance difference with speculative restart because collisions on the counters are unpredictable (thus restart prediction accuracy is low). With LL-SC, ordered access to the pointer must use software versions of `acquire` and `release`. The atomic incrementing of counters is more tailor-made for lock-free synchronization; however, there are still enough collisions to create some wasted 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) and two updates to shared structures that need not be ordered.

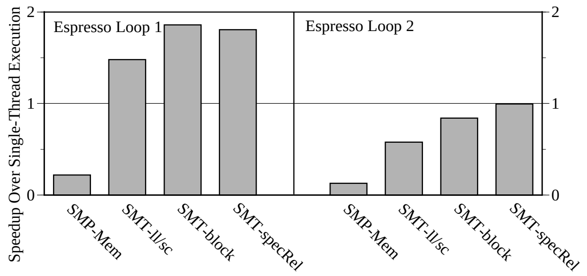

Figure 4. The speedup on two *espresso* loops.

The performance of the blocking synchronization was disappointing (Figure 4, loop 2). Further analysis uncovered several contributing factors. (1) The single-thread loop already has significant ILP. (2) Most of the shared variables are in registers in the single-thread version, but must be stored in memory for parallelization. (3) The locks constrained the efficient scheduling of memory accesses in the loop. (4) The branches are highly correlated from iteration to iteration, allowing our branch predictor to do very well for serial execution; however, much of this correlation was lost when the iterations went to different threads. Despite all this, choosing to parallelize this loop still would not hurt performance given our fast synchronization mechanisms.

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.

Livermore Loops. Unlike most of the Livermore Loops, loops 6, 13, and 14 are not parallelized by, for example, SUIF, because they each contain cross-iteration dependencies. These loops have a reasonable amount of code that is independent, however, and should be amenable to parallelization, given fine-grained synchronization.

Loop 6 is a doubly-nested loop that reads a triangular region of a matrix. The inner loop accumulates a sum. While parallelization of this loop (Figure 5, Loop 6), does not make sense on a conventional multiprocessor, it becomes profitable with standard SMT synchronization, and more so with speculative restart support. Later in this section, we'll show a trivial change to get much better performance. LL-SC performs poorly due to the high overhead of threads spinning at the barrier around the reduction for the sum.

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

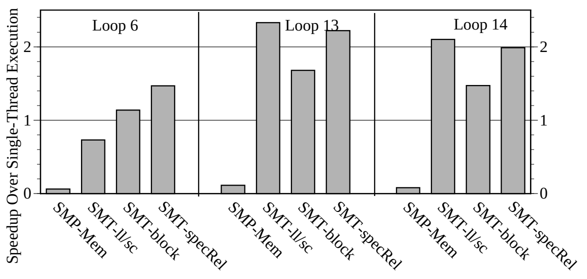

Figure 5. Speedup for three of the Livermore loops.

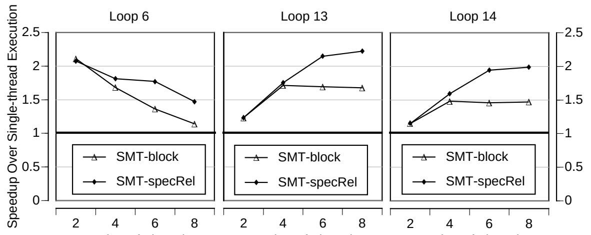

Figure 6. Blocking synchronization performance with varying number of threads.

Loop 14 is actually three loops, which are trivial to fuse. This maximizes the amount of parallel code available to execute concurrently with the serial portion. The serial portion is two updates to another indexed array, like *Loop 13*.

Figure 6 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, then fell off. Performance on *Loops 13* and *14* only improves beyond 2 threads with speculative restart. These results indicate that (1) it is important to choose the level of parallelism carefully and (2) the optimal choice of threads is dependent on the loop and the underlying synchronization mechanism.

For the five loops examined (two from espresso, three from livermore), none of which could be parallelized on a conventional multiprocessor, fine-grained synchronization enabled significant parallel speedups on four and no speedup or slowdown on one. In each case, parallelization created execution paths that made the speed of synchronization critical to the performance of the code, as all were sensitive to the exact mechanism used.

5. Related Work

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

Pai, et al. [8] 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 retries the lock so that software does not have to loop. They do not block the thread, nor do they associate releases with locks in their structure.

Bradford and Abraham [3] propose hardware-implemented semaphores which block a thread waiting for a semaphore. They compare this scheme with spin-waiting and OS-implemented blocking synchronization.

6. Summary

We have proposed a new synchronization mechanism based on a simple hardware structure called a lock box, tailored specifically for an SMT processor. This mechanism (1) maximizes synchronization efficiency by ensuring that threads waiting on synchronization consume no execution resources, and (2) minimizes synchronization latency by using lock-release prediction to resume blocked threads with no restart delay.

Acknowledgments

This work was funded by NSF CAREER grant MIP-9701708, NSF grant MIP-9632977, UC MICRO grant 97-018, and a DEC external research grant US-0040-97. Joel Emer and Rebecca Stamm contributed ideas and insight to this project.

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. Porterfi eld, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, June 1990.

[3] J. Bradford and S. Abraham. Effi cient synchronizatin for multithreaded processors. In *Workshop on Multithreaded Execution Architecture and Compilation*, January 1998.

[4] M. Herlihy. A methodology for implementing highly concurrent data objects. In *Symposium on Principles and Practices of Parallel Programming*, March 1990.

[5] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee. Exploiting fi ne-grain thread level parallelism on the MIT multi-alu processor. In *International Symposium on Computer Architecture*, June 1998.

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

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

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

[9] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *International Symposium on Computer Architecture*, May 1996.

[10] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture*, June 1995.

[11] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fi ne-grained synchronization on a simultaneous multithreading processor. Technical Report CS98-587, UCSD, June 1998.