Highly Efficient Synchronization Based on Active Memory Operations

Lixin Zhang
IBM Austin Research Lab
11400 Burnet Road, MS 904/6C019
Austin, TX 78758
zhangl@us.ibm.com

Zhen Fang and John B. Carter
School of Computing
University of Utah
Salt Lake City, UT 84112
zfang, retrac@cs.utah.edu

Abstract

Synchronization is a crucial operation in many parallel applications. As network latency approaches thousands of processor cycles for large scale multiprocessors, conventional synchronization techniques are failing to keep up with the increasing demand for scalable and efficient synchronization operations.

In this paper, we present a mechanism that allows atomic synchronization operations to be executed on the home memory controller of the synchronization variable. By performing atomic operations near where the data resides, our proposed mechanism can significantly reduce the number of network messages required by synchronization operations. Our proposed design also enhances performance by using fine-grained updates to selectively "push" the results of offloaded synchronization operations back to processors when they complete (e.g., when a barrier count reaches the desired value).

We use the proposed mechanism to optimize two of the most widely used synchronization operations, barriers and spin locks. Our simulation results show that the proposed mechanism outperforms conventional implementations based on load-linked/store-conditional, processor-centric atomic instructions, conventional memory-side atomic instructions, or active messages. It speeds up conventional barriers by up to 2.1 (4 processors) to 61.9 (256 processors) and spin locks by a factor of up to 2.0 (4 processors) to 10.4 (256 processors).

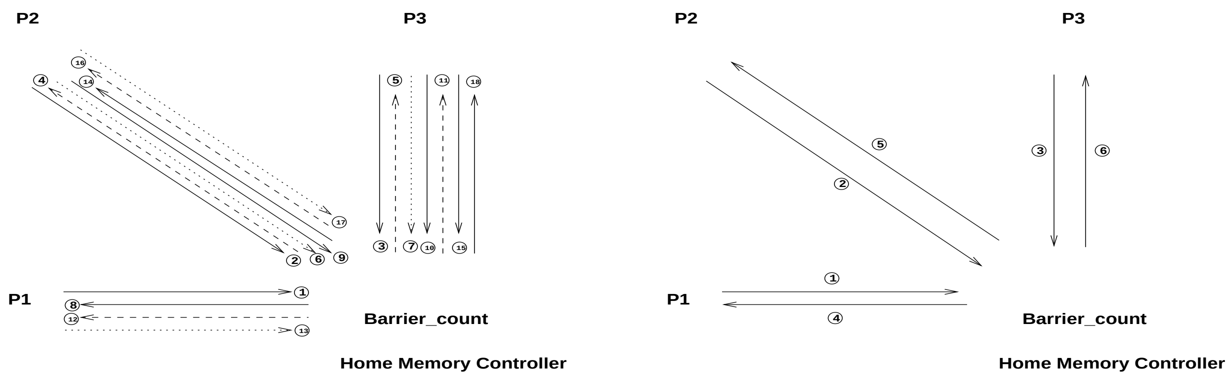
1 Introduction

Barriers and spinlocks are common synchronization primitives used by many parallel applications. A barrier ensures that no process in a group of cooperating processes advances beyond a given point until all processes have reached the barrier. A spin lock ensures atomic access to data or code protected by the lock. Given the serializing nature of synchronization operations, their performance often limits the achievable concurrency, and thus performance, of parallel applications.

The performance of synchronization operations is limited by two factors: (i) the number of remote accesses required for a synchronization operation and (ii) the latency of each remote access. Its impact on the overall performance of parallel applications is increasing due to the growing speed gap between processors and memory. Processor speeds are increasing by approximately 55% per year, while local DRAM latency is improving only approximately 7% per year and remote memory latency for large-scale machine is almost constant due to speed of light effects. These trends are making synchronization operations more and more expensive. For instance, a 32-processor barrier operation on an SGI Origin 3000 system takes about 90,000 cycles, during which time the 32 400MHz R14K processors could execute 5.76 million FLOPS. The 5.76 MFLOPS/barrier ratio is an alarming indication that conventional synchronization mechanisms hurt system performance.

Most conventional synchronization mechanisms are implemented using some form of processor-centric atomic read-modify-write operations. For example, the Itanium-2 processors support semaphore instructions [11] and many RISC processors use load-linked/store-conditional instructions [4, 13, 16] to implement synchornization operations. An LL instruction loads a block of data into the cache. A subsequent SC instruction attempts to write to the same block. It succeeds only if the block has not been evicted from the cache since the preceding LL. Any intervention request from another processor between the LL and SC pair causes the SC to fail. To implement an atomic operation, library routines typically retry the LL/SC pair repeatedly until the SC succeeds.

A main drawback of processor-centric atomic operations is that they introduce interprocessor communication for every atomic operation. Figure 1(a) illustrates a simple scenario in a CC-NUMA (cache-coherence non-uniform memory access) system, in which three processors synchronize using a barrier based on processor-centric atomic operations. Solid lines of this figure represent request and data messages, dashed lines represent intervention messages, and dotted lines represent inter-



(a) *using conventional atomic ops* (b) *using AMOs*

Figure 1. A three-processor barrier

vention replies. At the beginning, all three processors request exclusive ownership (*line (1), (2), and (3)*), but only one of them will succeed at one time; the others must retry. As the figure shows, even without additional interference, conventional barriers needs 18 one-way messages before all three processors can proceed past the barrier.

In an attempt to speed up synchronization operations without surrendering to increased programming complexity, we are investigating the value of augmenting a conventional memory controller (MC) with an **Active Memory Unit** (**AMU**) capable of performing simple atomic operations. We refer to such atomic operations as **Active Memory Operations** (**AMOs**). AMOs let processors ship simple computations to the AMU on the home memory controller of the data being processed, instead of loading the data in to a processor, processing it, and writing it back to the home node. AMOs are particularly useful for data items, such as synchronization variables, that are not accessed many times between when they are loaded into a cache and later evicted. Synchronization operations can exploit AMOs by performing atomic read-modify-write operations at the home node of the synchronization variables, rather than bouncing them back and forth across the network as each processor tests or modifies them.

To further reduce network traffic and speed up synchronization, we propose to exploit a fine-grained update mechanism so that AMUs can selectively push word-grained updates to processors to update cached data. Applications can control when and what to update, e.g., for a barrier operation, an update will be sent only when the barrier count indicates that all participating processes have reached the barrier. We assume that each node contains a remote access cache (RAC) where updates can be pushed so that word-grained updates can be supported without processor modifications. Issuing updates in this manner, rather than having processes spin across the network, dramatically reduces network traffic.

Figure 1(b) illustrates how an AMO-based barrier works. By employing AMOs, processor-issued requests will no longer be retried. Only two message (one request and one reply) are needed for each process. In this three-processor case, the total number of network messages required to perform a synchronization operation drops from 18 to 6. This dramatic reduction can lead to significant performance gains. In Section 4 we will show that AMO-based barriers outperform even highly optimized conventional barriers by a factor of 2.1 (4 processors) to 61.9 (256 processors), and spin locks by a factor of 2.0 (4 processors) to 10.4 (256 processors).

In the rest of the paper, Section 2 surveys relevant existing mechanisms not covered in this section, including tree-based barriers, ticket locks, array-based queuing locks, active messages, and simple memory-side atomic operations. Section 3 presents the architecture of the AMU and show how it can optimize barriers and spin locks. Section 4 describes our simulation environment and presents the performance numbers of barriers/locks based on AMOs, LL/SC instructions, atomic instructions, active messages, and memory-side atomic operations. Section 5 summarizes our conclusions and discusses future work.

2 Related Work

Active Message (ActMsg) is an efficient way in organizing parallel applications [3, 27]. An active message includes the address of a user-level handler to be executed by the home node processor upon message arrival using the message body as argument. Active messages can be used to implement AMO-style synchronization operations on a fixed "home node", without shuttling the relevant data back and forth across the network. However, using the node's primary processor to execute the handlers has much higher latency than dedicated hardware and interferes with useful work. In particular, the load imbalance induced by having a single node handle synchronization traffic will tend to severely impact performance due to Amdahl's Law effects. With an AMU, we achieve the same latency-reducing benefits of centralizing the simple synchronization operations without impacting the performance of any particular processor.

A number of barrier solutions have been proposed over the decades. The fastest of all approaches is to use a pair of dedicated wires between every two nodes [5, 23]. However, such approaches are only possible for small-scale systems, where the number of dedicated wires is small. For large-scale systems, the cost of having dedicated wires between every two nodes is prohibitive. In addition to the high cost of physical wires, hardware-wired approaches cannot support more than one barrier at one time and do not interact well with load-balancing techniques, such as processes migration, where the process-to-processor mapping is not static.

The SGI Origin 2000 [14] and Cray T3E [22] have a set of memory-side atomic operations (MAOs), triggered by writes to special IO addresses on the home node of synchronization variables. MAOs do not work in the coherent domain and rely on software to maintain coherence. To spin on the synchronization variable, each load request must bypass the cache and load data directly from the home node. Performance of MAO-based sync operations can be improved by spinning on a separate variable. Our experimental results will show that even this optimized version is slower than a simple AMO version.

The *fetch-and-add* instruction in the NYU Ultracomputer [7] is also implemented in the memory controller. It uses a combining network that tries to combine loads and stores for the same memory location within the routers. However, the hardware cost for queueing circuitry at each node is high, so there is a performance penalty for references that do not use the combining feature.

Some researchers have proposed *barrier trees* [9, 21, 28] , which use multiple barrier variables organized as a tree for a barrier operation so that atomic operations on different variables can be done in parallel. For example, in Yew *et al.*'s software combining tree [28], the processors are leaves of the tree and are organized into groups. The last processor in a group to arrive at a barrier increments a counter in the group's parent node. Continuing in this fashion, the last processor reaching the barrier point works its way to the root of the tree and triggers a reverse wave of wakeup operations to all processors. Barrier trees achieve significant performance gains on large-scale systems due to reduced hot spot effects, but they entail extra programming efforts and their overall performance is constrained by the base case combining barrier performance. AMOs can be used to improve the performance of the base combining barrier operation, thereby allowing flatter barrier trees.

QOLB [6, 12] by Goodman *et al.* serializes the synchronization requests through a distributed queue supported by hardware. The hardware queue mechanism greatly reduces network traffic. The hardware cost includes three new cache line states, storage of the queue entries, a "shadow line" mechanism for local spinning, and direct node-to-node transfer of the lock.

Off-loading the task of synchronization from the main processor to network processors is an approach taken by several recent clusters [10, 20, 26]. Gupta *et al.* [10] and Tipparaju *et al.* [26] use user-level one-sided protocols of MPI to implement communication functionalities, including barriers. The Quadrics^{TM}QsNet interconnect used by the ASCI Q supercomputer [20] supports both pure hardware barrier with a crossbar switch and hardware multicast, and a hybrid hardware/software tree barrier running on the network processors. The usability of the Quadrics hardware barrier is limited by two requirements: only one processor per node can participate in the synchronization and the participating nodes must be adjacent.

Spin locks are usually implemented in software using hardware primitives that the underlying machine provides, *e.g.,* LL/SC or atomic instructions. They suffer the same problems as barriers, though in a slightly different fashion. A number of spin locks have been proposed [1, 17]. Most of them achieve better performance at the cost of increased programming complexity.

3 AMU-Supported Synchronization

Our proposed mechanism adds an Active Memory Unit that can perform simple atomic arithmetic operations to the memory controller, extends the coherence protocol to support fine-grained updates, and augments the ISA with a few special AMO instructions. AMO instructions are encoded in an unused portion of the MIPS-IV instruction set space. We are considering a wide range of AMO instructions, but for this study we focus on `amo.inc` (increment by one) and `amo.fetchadd` (fetch and add). Semantically, these instructions are just like the atomic instructions implemented by processors.

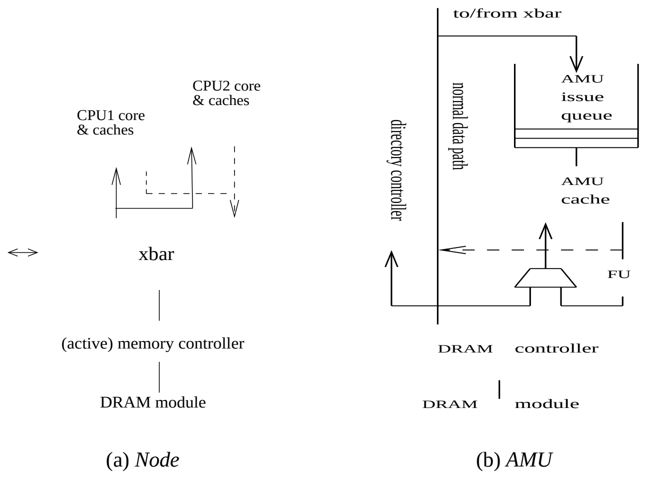


Figure 2. Hardware organization of the Active Memory Controller.

Programmers can use them as if they were processor-side atomic operations.

3.1 Hardware Organization

Figure 2 depicts the architecture that we assume. A crossbar connects processors to the network backplane, from which they can access remote processors, as well as their local memory and IO subsystems, shown in Figure 2 (a). In our model, the processors, crossbar, and memory controller all reside on the same die, as will be typical in near-future system designs. Figure 2 (b) is the block diagram of the Active Memory Controller with the proposed AMU delimited within the dotted box.

When a processor issues an AMO instruction, it sends a command message to the target address' home node. When the message arrives at the AMU of that node, it is placed in a queue waiting for dispatch. The control logic of the AMU exports a READY signal to the queue when it is ready to accept another request. The operands are then read and fed to the function unit (FU in Figure 2 (b)).

Accesses to synchronization variables exhibit high temporal locality because every participating process accesses the same synchronization variable. To further improve the performance of AMOs, we add a tiny cache to the AMU. This cache effectively coalesces operations to synchronization variables, eliminating the need to load from and write to the off-chip DRAM every time. Each AMO that hits in the AMU cache takes only two cycles to complete, regardless the number of processors contending for the synchronization variable. An *N*-word AMU cache allows *N* outstanding synchronization operations. For this study, we assume an eight-word AMU cache.

3.2 Fine-grained Updates

AMOs operate on coherent data. AMU-generated requests are sent to the directory controller as fine-grained "get" (for reads) or "put" (for writes) requests. The directory controller still maintains coherency at the block level. A fine-grained "get" loads the coherent value of a word (or a double-word) from local memory or a processor cache, depending on the state of the block containing the word. The directory controller changes the state of the block to "shared" and adds the AMU to the list of sharers. Unlike traditional data sharers, the AMU is allowed to modify the word without obtaining exclusive ownership first. The AMU sends a fine-grained "put" request to the directory controller when it needs to write a word back to local memory. When the directory controller receives a put request, it will send a word-update request to local memory and every node that has a copy of the block containing the word to be updated. [1].

To take advantage of fine-grained gets/puts, an AMO can include a "test" value that is compared against the result of the operation. When the result value matches the "test" value, the AMU sends a put request along with the result value to the directory controller. For instance, the "test" value of `amo.inc` can be set as the total number of processes expected to reach the barrier and then the update request is like a signal to all waiting processes that a barrier operation has completed.

A potential way to optimize synchronization is to use a write-update protocol on synchronization variables. However, issuing a block update after each write generates enormous amount of network traffic, offsetting the benefit of eliminating invalidation requests [8]. On the contrary, the put mechanism issues word-grained updates (thereby eliminating false sharing) and in the case of `amo.inc`, it only issues updates after the last process reaches the barrier rather than once every time a process reaches the barrier.

This get/put mechanism introduces temporal inconsistency between the barrier variable values in the processor caches and the AMU cache. In essence, the delayed put mechanism implements a release consistency model for barrier variables, where the condition of reaching a target value acts as a release point. Though we must be careful when applying AMOs to applications where release consistency might cause problems,

[1]Fine-grained "get/put" operations are part of a more general DSM architecture we are investigating. Its details are beyond the scope of this paper.

(a) naive coding

```
atomic_inc( &barrier_variable );
spin_until( barrier_variable == num_procs );
```

(b) "optimized" version

```
int count = atomic_inc( &barrier_variable );
if( count == num_procs-1 )
      spin_variable = num_procs;
else
      spin_until( spin_variable == num_procs );
```

(c) AMO version

```
amo_inc( &barrier_variable,num_procs );
spin_until( barrier_variable == num_procs );
```

Figure 3. There barrier implementations.

release consistency is a completely acceptable memory model for synchronization operations.

3.3 Programming Model

3.3.1 Barrier

The `amo.inc` instruction increments a specified memory location by one and returns the original memory content to the requesting processor. When it is used to implement barriers, its test value is set as the expected value of the barrier variable after all processes have reached the barrier point.

Figure 3(a) shows a naive barrier implementation, where `num_procs` is the number of participating processes. This implementation is inefficient because it directly spins on the barrier variable. Since processes that have reached the barrier repeatedly try to read the barrier variable, the next increment attempt by another process will have to compete with these read requests, possibly resulting in a long latency for the increment operation. Although processes that have reached the barrier can be suspended to avoid interference with the subsequent increment operations, the overhead of suspending and resuming processes is too high to have an efficient barrier.

A common optimization to this barrier implementation is to use another variable, as shown in Figure 3(b). Instead of spinning on the barrier variable, this loop spins on another variable `spin_variable`. Because data coherence is maintained in block level, for this coding to work efficiently, programmers must make sure that `barrier_variable` and `spin_variable` do not reside in the same block. Using the spin variable eliminates false sharing between spin and increment operations. However, it introduces an extra write to the spin variable for each barrier operation, which causes the home node to send an invalidation request to every processor and then every processor to reload the spin variable. Nevertheless, the benefit of using the spin variable often overwhelms its overhead. It is a classic example of trading programming complexity for performance. Nikolopoulos and Papatheodorou [19] have demonstrated that using the spin variable gives 25% performance improvement for a barrier synchronization across 64 processors.

With AMOs, atomic increment operations are performed at the memory controller without invalidating shared copies in processor caches and the cache copies are automatically updated when all processes reach the barrier. Consequently, AMO-based barriers can use the naive coding, as shown in Figure 3(c), where `amo_inc()` is a wrapper function for the `amo.inc` instruction, which uses `num_procs` as the “test” value.

3.3.2 Spin lock

The `amo.fetchadd` instruction adds a designated value to a specified memory location, immediately updates the shared copies in processor caches with the new value, and returns the old value. Different spin lock algorithms require different atomic primitives. We do not intend to elaborate on every one of them. Instead, we apply `amo.fetchadd` to two representative spin lock algorithms, ticket lock and Anderson’s array-based queuing lock [2].

The ticket lock is a simple algorithm that grants acquisition requests in FIFO order. Figure 4 is one of its typical implementations.

```
acquire_ticket_lock( ) {
  int my_ticket = fetch_and_add(&next_ticket, 1);
  spin_until(my_ticket == now_serving);
}
release_ticket_lock( ) {
  now_serving = now_serving + 1;
}
```

Figure 4. Ticket lock pseudo-code

It has two global variables, the sequencer (`next_ticket`) and the counter (`now_serving`). To acquire the lock, a process atomically increments the sequencer, obtains a ticket, and waits for the count to become equal to its ticket number. The reigning process releases the lock by incrementing the counter. Races to the sequencer and delays in the propagation of the new counter value cause the performance of the ticket lock to degrade rapidly as the number of participating processors goes up. Mellor-Crummy and Scott [17] showed that proportional backoff inserted to the spinning stage was very effective in enhancing efficiency of ticket locks. On their evaluation systems, every reference to the global variable `now_serving` was a remote memory access because it was not cached.

Backoff eliminated most of the network and memory traffic and greatly improved lock passing efficiency. However, on modern cache-coherent multiprocessors, backoff is less effective. Because most of spinning reads to `now_serving` hit in local caches, there are very few remote accesses for backoff to eliminate. Inserting backoff is also not risk-free; delaying one process will force a delay on others that arrive later than the process because of the FIFO nature of the algorithm.

T. Anderson's array-based queuing lock [2] uses an array of flags. A counter serves as the index into the array. Every process spins on its own flag. When the lock is released, only the next winner's flag access turns into a remote memory access. All other processors keep spinning on their local caches. The sequencer remains a hot spot, though. Selectively signaling one processor at a time, however, may noticeably improve performance in large systems. In addition, all global variables (the sequencer, the counter and all the flags) must be placed in different cache lines to achieve the best performance.

To implement spin locks using AMOs, we replace the atomic primitive `fetch_and_add` with AMO instruction `amo_fetchadd()`. We also use `amo_fetchadd()` on the counter to take advantage of the put mechanism. In addition, using AMOs makes it a moot point to put global variables into different cache lines.

3.3.3 Programming complexity

Using conventional synchronization primitives often requires significant effort from programmers to write correct, efficient, and deadlock-free parallel codes. In contrast, AMOs work in cache coherent domain, do not lock any system resources and eliminate the need for programmers to be aware of how the atomic instructions are implemented. In addition, we will show in Section 4 that the complex algorithms designed for conventional platforms (*e.g.* combining tree barriers and array-based queuing locks) are no longer needed for AMOs. Since synchronization-related codes are often the hardest parts to code and debug, simplifying the programming model is another advantage of AMOs over other mechanisms, in addition to performance superiority.

4 Evaluation

4.1 Simulation Environment

We use a cycle-accurate execution-driven simulator, UVSIM, in our performance study. UVSIM models a hypothetical future-generation Origin architecture, including a directory-based coherence protocol [24] that supports both write-invalidate and fine-grained write-update, as described in Section 3.2. Each simulated node contains two MIPS next-generation microprocessors connected to a high-bandwidth bus. Also connected to the bus is a future-generation Hub [25], which contains the processor interface, memory controller, directory controller, network interface, IO interface, and active memory unit.

Parameter	Value
Processor	4-issue, 48-entry active list, 2GHz
L1 I-cache	2-way, 32KB, 64B lines, 1-cycle lat.
L1 D-cache	2-way, 32KB, 32B lines, 2-cycle lat.
L2 cache	4-way, 2MB, 128B lines, 10-cycle lat.
System bus	16B CPU to system, 8B system to CPU
	max 16 outstanding L2C misses, 1GHZ
DRAM	16 16-bit-data DDR channels
Hub clock	500 MHz
DRAM	60 processor cycles latency
Network	100 processor cycles latency per hop

Table 1. System configuration.

Table 1 lists the major parameters of the simulated systems. The DRAM backend has 16 20-bit channels connected to DDR DRAMs, which enables us to read an 80-bit burst every two cycles. Of each 80-bit burst, 64 bits are data. The remaining 16 bits are a mix of ECC bits and partial directory state. The simulated interconnect subsystem is based on SGI's NUMALink-4. The interconnect is built using a fat-tree structure, where each non-leaf router has eight children. We model a network hop latency of 50 nsecs (100 cpu cycles). The minimum network packet is 32 bytes.

UVSIM has a micro-kernel that supports all common system calls. It directly executes statically linked 64-bit MIPS-IV executables. UVSIM supports the OpenMP runtime environment. All benchmark programs used in this paper are OpenMP-based parallel programs. All programs in our study are compiled using the MIPSpro Compiler 7.3 with an optimization level of "`-O3`".

We have validated the core of our simulator by setting the configurable parameters to match those of an SGI Origin 3000, running a large mix of benchmark programs on both a real Origin 3000 and the simulator, and comparing performance statistics (e.g., run time, cache miss rates, etc.). The simulator-generated results are all within 20% of the corresponding numbers generated by the real machine, most within 5%.

4.2 Benchmarks and Results

We use a set of representative synchronization algorithms as test benchmarks. The barrier function is from the IRIX OpenMP library. In addition to simple barriers, we construct the software barrier combining tree based

CPUs	Speedup over LL/SC barrier			
	ActMsg	Atomic	MAO	AMO
4	0.95	1.15	1.21	**2.10**
8	1.70	1.06	2.70	**5.48**
16	2.00	1.20	3.61	**9.11**
32	2.38	1.36	4.20	**15.14**
64	2.78	1.37	5.14	**23.78**
128	2.74	1.24	8.02	**34.74**
256	2.82	1.23	14.70	**61.94**

Table 2. Performance of different barriers.

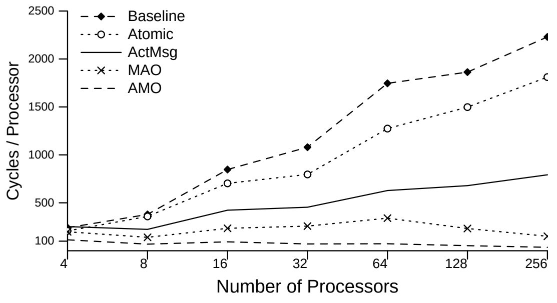


Figure 5. Cycles-per-processor of different barriers.

on the work by Yew *et al.* [28] for each implementation. Ticket locks and array-based queuing locks are based on what is proposed by Mellor-Crummey and Scott [17].

We compare AMOs with LL/SC instructions, active messages ("ActMsg"), processor-side atomic instructions ("Atomic"), and existing memory-side atomic operations ("MAOs"). The LL/SC-based versions are taken as the baseline. The AMU cache is used for both MAOs and AMOs.

4.2.1 Non-tree-based barriers

Table 2 presents the speedups of different barrier implementations over the baseline. A speedup of less than one indicates a slowdown. We vary the number of processors from four (*i.e.*, two nodes) to 256, the maximum number of processors allowed by the directory structure [24] that we use. The ActMsg, Atomic, MAO and AMO versions all perform better than the baseline LL/SC version, and scale better. Specifically, when the number of processors is over 8, active messages outperform LL/SC by a factor of 1.70 to 2.82. Atomic instructions outperform LL/SC by a factor of 1.06 to 1.37. Memory-side atomic operations outperform LL/SC by a factor of 1.21 at four processors to an impressive 14.70 at 256 processors. However, AMO-based version dwarfs all other versions. Its speedup ranges from a factor of 2.10 for four processors to a factor of 61.94 for 256 processors.

In the baseline implementation, each processor loads the barrier variable into its local cache before incrementing it using LL/SC instructions. Only one processor will succeed at one time; other processors will fail and retry. After a successful update by a processor, the barrier variable will move to another processor, and then to another processor, and so on. As the system grows, the average latency to move the barrier variable between processors increases, as does the amount of contention. As a result, the synchronization time in the base version increases superlinearly as the number of nodes increases. This effect can be seen particularly clearly in Figure 5, which plots the per-processor barrier synchronization time for each barrier implementation.

For the active message version, an active message is sent for every increment operation. The overhead of invoking the active message handler for each increment operation dwarfs the time required to run the handler itself. Nonetheless, the benefit of eliminating remote memory accesses outweighs the high invocation overhead, which results in performance gains as high as 182%.

Using processor-centric atomic instructions eliminates the failed SC attempts in the baseline version. However, its performance gains are relatively small, because it still requires a round trip over the network for every atomic operation, all of which must be performed serially.

The MAO version performs significantly better than Atomic. It scales exceptionally well. At 256 processors, it is nearly 15 times faster than the baseline. This result further demonstrates that putting computation near memory is a good solution for synchronization operations.

AMO barriers are four times faster than MAO. This performance advantage of AMOs over MAOs has come from the "delayed update" enabled by the test value mechanism and the fine-grained update protocol. Since all processors are spinning on the barrier variable, every local cache likely has a shared copy of it. Thus the total cost of sending updates is approximately the time required to send a single update multiplied by the number of participating processors. [2]

Roughly speaking, the time to perform an AMO barrier equals $(t_o + t_p \times P)$, where t_o is a fixed overhead and t_p is a small value related to the processing time of an `amo.inc` operation and an update request, and P is the number of processors being synchronized. This expression implies that AMO barriers scale well, which is clearly illustrated in Figure 5. This figure reveals that the per-processor latency of AMO barriers is constant with respect to the total number of processors. In fact, the per-processor latency drops off slightly as the num-

[2]We do not assume that the network has multicast support; AMO performance would be even higher if the network supported such operations.

CPUs	Speedup over LL/SC barrier					
	LL/SC+tree	ActMsg+tree	Atomic+tree	MAO+tree	AMO+tree	AMO
16	1.70	2.41	2.25	2.60	2.59	**9.11**
32	2.24	2.85	2.62	4.09	4.27	**15.14**
64	4.22	6.92	5.61	8.37	8.61	**23.78**
128	5.26	9.02	6.13	12.69	13.74	**34.74**
256	8.38	14.72	11.22	20.37	22.62	**61.94**

Table 3. Performance of tree-based barriers.

ber of processors increases because the fixed overhead is amortized by more processors. In contrast, other implementations see higher per-processor time when the system becomes larger.

4.2.2 Tree-based barriers

For all tree-based barriers, we use a two-level tree structure regardless of the number of processors. For each configuration, we try all possible tree branching factors and use the one that delivers the best performance. The initialization time of the tree structures is not included in the reported results. The smallest configuration that we consider for tree-based barriers has 16 processors. Table 3 shows the speedups of tree-based barriers over the original baseline implementation. Figure 6 shows the number of cycles per processor for the tree-based barriers.

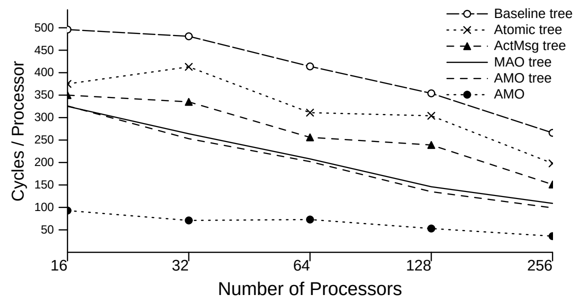


Figure 6. Cycles-per-processor of tree-based barriers.

Our simulation results indicate that tree-based barriers perform much better and scale much better than normal barriers, which concurs with the findings of Michael *et al.* [18]. On a 256-processor system, all tree-based barriers are at least eight times faster than the baseline barrier. As seen in Figure 6, the cycle-per-processor number for tree-based barriers decreases as the number of processors increases, because the high overhead associated with using trees is amortized across more processors and the tree contains more branches that can proceed in parallel.

The best branching factor for a given system is often not intuitive. Markatos *et al.* [15] have demonstrated that improper use of trees can drastically degrade the performance of tree-based barriers to even below that of simple centralized barriers. Nonetheless, our simulation results demonstrate the performance potential of tree-based barriers.

Even with all of the advantages, tree-based barriers are still significantly slower than AMO-based barriers. For instance, the best non-AMO tree-based barrier (MAO + tree) is still 3 times slower than the AMO-based barrier on a 256-processor system.

Interestingly, the combination of AMOs and trees performs worse than AMOs alone in all tested configurations. The cost of an AMO-based barrier includes a large fixed overhead and a very small number of cycles per processor. Using tree structures on AMO-based barriers essentially introduces the fixed overhead more than once, therefore resulting in a longer barrier synchronization time. That AMOs alone are better than the combination of AMOs and trees is another indication that AMOs do not require heroic programming effort to achieve good performance. However, the relationship between normal AMOs and tree-based AMOs might change if we move to systems with tens of thousands processors. Determining whether or not tree-based AMO barriers can provide extra benefits on very large-scale systems is part of our future work.

4.2.3 Spin locks

Table 4 presents the speedups of different ticket locks and array-based queuing locks over the LL/SC-based ticket lock. For traditional mechanisms, when the system has 32 or fewer processors, ticket lock is faster than array lock. Otherwise, array lock is faster. This verifies the effectiveness of array locks in alleviating hot spot in larger systems.

AMOs greatly improve the performance of both locks and make the difference between ticket lock and array lock negligible. This observation implies that with AMOs, we can use the simpler tick locks instead of more complicated array locks without losing any performance.

The main reason that AMOs outperform others is its ability to reduce network traffic. Figure 7 shows the network traffic, normalized to the LL/SC version, of dif-

CPUs	LL/SC		ActMsg		Atomic		MAO		AMO	
	ticket	array	ticket	array	ticket	array	ticket	array	**ticket**	array
4	1.00	0.48	1.08	0.47	0.92	0.53	1.01	0.57	**1.95**	1.31
8	1.00	0.58	1.64	0.56	0.94	0.67	1.07	0.59	**2.34**	2.03
16	1.00	0.60	2.18	0.65	0.93	0.67	1.07	0.62	**2.20**	2.41
32	1.00	0.62	1.48	0.64	0.94	0.76	1.08	0.65	**2.29**	2.14
64	1.00	1.42	0.60	1.42	0.80	1.60	0.64	1.49	**4.90**	5.45
128	1.00	2.40	0.91	2.60	1.21	2.78	1.00	2.69	**9.28**	9.49
256	1.00	2.71	0.97	2.92	1.22	3.25	0.90	3.13	**10.36**	10.05

Table 4. Speedups of different locks over the LL/SC-based locks.

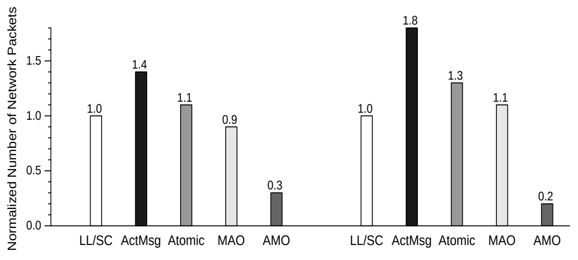


Figure 7. Network traffic for ticket locks.

ferent ticket locks on 128-processor and 256-processor systems. In both systems, AMOs have significantly less traffic than any other approach. Interestingly, active messages, originally designed for eliminating remote memory accesses, now suffer more network traffic than others. It is because the high invocation overhead of the message handlers leads to timeouts and retransmissions of active messages in heavily contention environments.

5 Conclusions

Efficient synchronization is crucial to effective parallel programming of large multiprocessor systems. As network latency rapidly approaches thousands of processor cycles and multiprocessors systems are becoming larger and larger, synchronization speed is quickly becoming a significant performance determinant.

We present an efficient synchronization mechanism based on special atomic active memory operations performed in the memory controller. AMO-based barriers do not require extra spin variables or complicated tree structures to achieve good performance. AMO-based spin locks can use one of the simplest algorithms and still outperform implementations using more complex algorithms.

In conclusion, AMOs enable extremely efficient synchronization at rather low hardware cost, with a simple programming model. Our simulation results show that AMOs are much more efficient than LL/SC instructions, active messages, processor-side and memory-side atomic operations, outperforming them by up to a factor of 62 for barriers and 10 for spin locks.

Acknowledgments

The authors would like to thank Silicon Graphics Inc. for the technical documentations provided for the simulation, and in particular, the valuable input on AMO-related research from Marty Deneroff, Steve Miller and Steve Reinhardt. We also thank Mike Parker of the University of Utah for discussions on the hardware design of the AMU.

References

[1] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, Sept. 2003.

[2] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE TPDS*, 1(1):6–16, Jan. 1990.

[3] M. Banikazemi, R. Govindaraju, R. Blackmore, and D. K. Panda. MPI-LAPI: An efficient implementation of MPI for IBM RS/6000 SP systems. *IEEE TPDS*, 12(10):1081–1093, 2001.

[4] Compaq Computer Corporation. Alpha architecture handbook, version 4, Feb. 1998.

[5] Cray Research, Inc. Cray T3D systems architecture overview, 1993.

[6] J. R. Goodman, M. K. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. In *Proc. of the 3rd ASPLOS*, pp. 64–75, Apr. 1989.

[7] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU multicomputer - designing a MIMD shared-memory parallel machine. *IEEE TOPLAS*, 5(2):164–189, Apr. 1983.

[8] H. Grahn, P. Stenström, and M. Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3):247–271, June 1995.

[9] R. Gupta and C. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *IJPP*, 18(3):161–180, June 1989.

[10] R. Gupta, V. Tipparaju, J. Nieplocha, and D. Panda. Efficient barrier using remote memory operations on VIA-based clusters. In *IEEE International Conference on Cluster Computing (Cluster02)*, pp. 83 – 90, Sept. 2002.

[11] Intel Corp. Intel Itanium 2 processor reference manual. http://www.intel.com/design/itanium2/manuals/25111001.pdf.

[12] A. Kägi, D. Burger, and J. Goodman. Efficient synchronization: Let them eat QOLB. In *Proc. of the 24th ISCA*, May 1997.

[13] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[14] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *ISCA97*, pp. 241–251, June 1997.

[15] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. The effect of multiprogramming on barrier synchronization. In *Proc. of the 3rd IPDPS*, pp. 662–669, Dec. 1991.

[16] C. May, E. Silha, R. Simpson, and H. Warren. *The PowerPC Architecture: A Specification for a New Family of Processors, 2nd edition.* Morgan Kaufmann, May 1994.

[17] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, Feb. 1991.

[18] M. M. Michael, A. K. Nanda, B. Lim, and M. L. Scott. Coherence controller architectures for SMP-based CC-NUMA multiprocessors. In *Proc. of the 24th ISCA*, pp. 219–228, June 1997.

[19] D. S. Nikolopoulos and T. A. Papatheodorou. The architecture and operating system implications on the performance of synchronization on ccNUMA multiprocessors. *IJPP*, 29(3):249–282, June 2001.

[20] F. Petrini, J. Fernandez, E. Frachtenberg, and S. Coll. Scalable collective communication on the ASCI Q machine. In *Hot Interconnects 12*, Aug. 2003.

[21] M. Scott and J. Mellor-Crummey. Fast, contention-free combining tree barriers for shared memory multiprocessors. *IJPP*, 22(4), 1994.

[22] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. of the 7th ASPLOS*, Oct. 1996.

[23] S. Shang and K. Hwang. Distributed hardwired barrier synchronization for scalable multiprocessor clusters. *IEEE TPDS*, 6(6):591–605, June 1995.

[24] Silicon Graphics, Inc. *SN2-MIPS Communication Protocol Specification, Revision 0.12*, Nov. 2001.

[25] Silicon Graphics, Inc. *Orbit Functional Specification, Vol. 1, Revision 0.1*, Apr. 2002.

[26] V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Proc. of the International Parallel and Distributed Processing Symposium*, page 84a, Apr. 2003.

[27] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th ISCA*, pp. 256–266, May 1992.

[28] P. Yew, N. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. on Computers*, C-36(4):388–395, Apr. 1987.