Fast Synchronization on Shared-Memory Multiprocessors: An Architectural Approach

Zhen Fang[1], Lixin Zhang[2], John B. Carter[1], Liqun Cheng[1], Michael Parker[3]

[1] School of Computing
University of Utah
Salt Lake City, UT 84112, U.S.A.
{*zfang, retrac, legion*}*@cs.utah.edu*
Telephone: 801.587.7910 Fax: 801.581.5843

[2] IBM Austin Research Lab
11400 Burnet Road, MS 904/6C019
Austin, TX 78758, U.S.A.
zhangl@us.ibm.com

[3] Cray, Inc.
1050 Lowater Road
Chippewa Falls, WI 54729, U.S.A.
map@cray.com

Abstract

Synchronization is a crucial operation in many parallel applications. Conventional synchronization mechanisms are failing to keep up with the increasing demand for efficient synchronization operations as systems grow larger and network latency increases.

The contributions of this paper are threefold. First, we revisit some representative synchronization algorithms in light of recent architecture innovations and provide an example of how the simplifying assumptions made by typical analytical models of synchronization mechanisms can lead to significant performance estimate errors. Second, we present an architectural innovation called active memory *that enables very fast atomic operations in a shared-memory multiprocessor. Third, we use execution-driven simulation to quantitatively compare the performance of a variety of synchronization mechanisms based on both existing hardware techniques and active memory operations. To the best of our knowledge, synchronization based on active memory outforms all existing spinlock and non-hardwired barrier implementations by a large margin.*

Keywords: distributed shared-memory, coherence protocol, synchronization, barrier, spinlock, memory controller

This effort was supported by the National Security Agency (NSA), the Defense Advanced Research Projects Agency (DARPA) and Silicon Graphics Inc. (SGI). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of SGI, NSA, DARPA, or the US Government.

1 Introduction

Barriers and spinlocks are synchronization mechanisms commonly 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 spinlock ensures atomic access to data or code protected by the lock. Their efficiency 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. The impact of synchronization performance 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 machines is almost constant due to speed of light effects [26].

For instance, a 32-processor barrier operation on an SGI Origin 3000 system takes about 232,000 cycles, during which time the 32 R14K processors could have executed 22 million FLOPS. This 22 MFLOPS/barrier ratio is an alarming indication that conventional synchronization mechanisms hurt system performance.

Over the years, many synchronization mechanisms and algorithms have been developed for shared-memory multiprocessors. The classical paper on synchronization by Mellor-Crummy and Scott provides a thorough and detailed study of representative barrier and spinlock algorithms, each with their own hardware assumptions [21]. More recent work surveys the major research trends of spinlocks [2]. Both papers investigate synchronization more from an algorithmic perspective than from a hardware/architecture angle.

We feel that a hardware-centric study of synchronization algorithms is a necessary supplement to this prior work, especially given the variety of new architectural features and the significant quantitative changes that have taken place in multiprocessor systems over the last decade. Sometimes small architectural innovations can negate key algorithmic scalability properties. For example, neither of the above-mentioned papers differentiates between the way that the various read-modify-write(RMW) primitives (e.g., `test-and-set` or `compare-and-swap`) are physically implemented. However, the location of the hardware RMW unit, e.g., in the processor or in the communication fabric or the memory system, can have a dramatic impact on synchronization performance. For example, we find that when the RMW functionality is performed near the memory/directory controller rather than via processor-side atomic operations, 128-processor barrier performance can be improved by a factor of 10. Moving the RMW operation from the processor to a memory controller can change the effective time complexity of a barrier operation to $O(1)$ network latencies from no better than $O(N)$ in conventional implementations for the same basic barrier algorithm. This observation illustrates the potential problems associated with performing conventional "pen-and-pencil" algorithmic complexity analysis on synchronization mechanisms.

While paper-and-pencil analysis of algorithms tends to ignore many of the subtleties that make a big difference on real machines, running and comparing programs on real hardware is limited by the hardware primitives available on, and the configurations of, available machines. Program trace analysis is hard

because the hardware performance monitor counters provide limited coverage and reading them during program execution changes the behavior of an otherwise full-speed run. Experimenting with a new hardware primitive is virtually impossible on an existing machine. As a result, in this paper we use execution-driven simulation to evaluate mixes of synchronization algorithms, hardware primitives, and the physical implementations of these primitives. We do not attempt to provide a comprehensive evaluation of all proposed barrier and spinlock algorithms. Rather, we evaluate versions of a barrier from a commercial library and a representative spinlock algorithm adapted to several interesting hardware platforms. Detailed simulation helps us compare quantitatively the performance of these synchronization implementations and, when appropriate, correct previous mis-assumptions about synchronization algorithm performance when run on real hardware.

The rest of this paper is organized as follows. Section 2 analyzes the performance of a variety of barrier and spinlock algorithms on a cc-NUMA (cache-coherent non-uniform memory access) system using different RMW implementations, including load-linked/store-conditional instructions, processor-side atomic operations, simple memory-side atomic operations, and active messages. Section 3 presents the design of *active memory*, which supports sophisticated memory-side atomic operations and can be used to optimize synchronization performance. Section 4 describes our simulation environment and presents the performance numbers of barriers and spinlocks implemented using a variety of atomic hardware primitives. Finally, Section 5 draws conclusions.

2 Synchronization on cc-NUMA Multiprocessors

In this section, we describe how synchronization operations are implemented on traditional shared memory multiprocessors. We then describe architectural innovations that have been proposed in recent years to improve synchronization performance. Finally, we provide a brief time complexity estimate for barriers and spinlocks based on the various of available primitives.

2.1 Background

2.1.1 Atomic Operations on cc-NUMA systems

Most systems provide some form of processor-centric atomic RMW operations for programmers to implement synchronization primitives. For example, the Intel Itanium™processor supports semaphore instructions [14], while most major RISC processors [7, 16, 20] use load-linked/store-conditional instructions. An LL(load-linked) instruction loads a block of data into the cache. A subsequent SC(store-conditional) instruction attempts to write to the same block. It succeeds only if the block has not been referenced since the preceding LL. Any memory reference to the block from another processor between the LL and SC pair causes the SC to fail. To implement an atomic primitive, library routines typically retry the LL/SC pair repeatedly until the SC succeeds.

(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 );
```

Figure 1. Barrier code.

A drawback of processor-centric atomic RMW operations is that they introduce interprocessor communication for every atomic operation. In a directory-based write-invalidate CC-NUMA system, when a processor wishes to modify a shared variable, the local DSM hardware sends a message to the variable's home node to acquire exclusive ownership. In response, the home node typically sends invalidation messages to other nodes sharing the data. The resulting network latency severely impacts the efficiency of synchronization operations.

2.1.2 Barriers

Figure 1(a) shows a naive barrier implementation. 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 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 typically too high to be useful.

A common optimization to this barrier implementation is to introduce a new variable on which processes spin, e.g., `spin_variable` in Figure 1(b). Because coherence is maintained at cache line granularity, `barrier_variable` and `spin_variable` should not reside on the same cache line. Using a separate spin variable eliminates false sharing between the spin and increment operations. However, doing so 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 a separate spin variable often outweighs its overhead, which is a classic example of trading programming complexity for performance. Nikolopoulos and Papatheodorou [23] have demonstrated that using a separate spin variable improves performance by 25% for a 64-processor barrier synchronization.

We divide the total time required to perform a barrier operation into two components, *gather* and *release*. *Gather* is the interval during which each thread signals its arrival at the barrier. *Release* is the time it takes to convey to every process that the barrier operation has completed and it is time to progress.

```
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 2. Ticket lock code

Assuming a best case scenario where there is no external interference, the algorithm depicted in Figure 1(b) requires at least 15 one-way messages to implement the *gather* phase in a simple 3-process barrier using LL/SC or processor-side atomic instructions. These 15 messages consist primarily of invalidation, invalidation acknowledgement, and load request messages. Twelve of these messages must be performed serially, i.e., they cannot be overlapped.

In the optimized barrier code, when the last process arrives, it invalidates the cached copies of `spin_variable` in the spinning processors, modifies `spin_variable` in DRAM, and we enter *release* phase. Every spinning process needs to fetch the cache line that contains `spin_variable` from its home node. Modern processors employ increasingly large cache line sizes to capture spatial locality. When the number of processes is large, this burst of reads to `spin_variable` will cause congestion at the DRAMs and network interface.

Some researchers have proposed *barrier trees* [12, 27, 33], which use multiple barrier variables organized hierarchically so that atomic operations on different variables can be done in parallel. For example, in Yew *et.al.*'s software combining tree [33], processors are leaves of a tree. The processors are organized into groups and when the last processor in a group arrives at the barrier, it increments a counter in the group's parent node. Continuing in this fashion, when all groups of processors at a particular level in the barrier tree arrive at the barrier, the last one to arrive increments an aggregate barrier one level higher in the hierarchy until the last processor reaches the barrier. When the final process arrives at the barrier and reaches the root of the barrier tree, it triggers a wave of wakeup operations down the tree to signal each waiting processor. Barrier trees achieve significant performance gains on large systems by reducing hot spots, but they require extra programming effort and their performance is constrained by the base (single group) barrier performance.

2.1.3 Spinlocks

Ticket locks are widely used to grant mutual exclusion to processes in FIFO order. Figure 2 presents a typical ticket lock implementation that employs two global variables, a sequencer (`next_ticket`) and a counter (`now_serving`). To acquire the lock, a process atomically increments the sequencer, thus obtaining a ticket, and then waits for the count to become equal to its ticket number. A process releases the lock by incrementing the counter. Data races on the sequencer and propagation delays of the new counter value degrade the performance of ticket locks rapidly as the number of participating processors increases. Mellor-

Crummy and Scott [21] showed that inserting a proportional backoff in the spinning stage greatly improves the efficiency of ticket locks on a system without coherent caches by eliminating most remote accesses to the global `now_serving` variable. On modern cache-coherent multiprocessors, where most of spinning reads to `now_serving` hit in local caches, there are very few remote accesses for backoff to eliminate. Also, inserting backoff is not risk-free; delaying one process forces a delay on other processes that arrive later because of the FIFO nature of the algorithm.

On a cache-coherent multiprocessor, T. Anderson's array-based queuing lock [3] is reported to be one of the best spinlock implementations [21]. Anderson's lock 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 cached flags. Selective signaling improves performance, but the sequencer remains a hot spot. Nevertheless, selectively signaling one processor at a time noticeably improves performance for large systems. To further improve performance, all global variables (the sequencer, the counter and all the flags) should be mapped to different cache lines to avoid false sharing.

2.2 Related Architectural Improvements to Synchronization

Many architectural innovations have been proposed over the decades to overcome the overhead induced by cc-NUMA coherence protocols on synchronization primitives, as described in Section 2.1.

The fastest hardware synchronization implementation uses a pair of dedicated wires between every two nodes [8, 29]. However, for large systems the cost and packaging complexity of running dedicated wires between every two nodes is prohibitive. In addition to the high cost of physical wires, this approach cannot support more than one barrier at one time and does not interact well with load-balancing techniques, such as process migration, where the process-to-processor mapping is not static.

The *fetch-and-*ϕ instructions in the NYU Ultracomputer [11, 9] are the first to implement atomic operations in the memory controller. In addition to *fetch-and-add* hardware in the memory module, the Ultracomputer supports an innovative combining network that combines references for the same memory location within the routers.

The SGI Origin 2000 [18] and Cray T3E [28] support a set of memory-side atomic operations (MAOs) that are triggered by writes to special IO addresses on the home node of synchronization variables. However, MAOs do not work in the coherent domain and rely on software to maintain coherence, which has limited their usage.

The Cray/Tera MTA [17, 1] uses full/empty bits in memory to implement synchronized memory references in a cacheless system. However, it requires custom DRAM (an extra bit for every word) and it is not clear how it can work efficiently in presence of caches. The *fetch_add* operation of the MTA is similar to and predates the MAOs of the SGI Origin 2000 and Cray T3E.

Active message are an efficient way to organize parallel applications [5, 32]. An active message includes the address of a user-level handler to be executed by the receiving processor upon message arrival using

the message body as the arguments. Active messages can be used to perform atomic operations on a synchronization variable's home node, which eliminates the need to shuttle the data back and forth across the network or perform long latency for remote invalidations. However, performing the synchronization operations on the node's primary processor rather than on dedicated synchronization hardware entails higher invocation latency and interferes with useful work being performed on that processor. In particular, the load imbalance induced by having a single node handle synchronization traffic can severely impact performance due to Amdahl's Law effects.

The I-structure and explicit token-store (ETS) mechanism supported by the early dataflow project Monsoon [4, 24] can be used to implement synchronization operations in a manner similar to active messages. A token comprises a value, a pointer to the instruction to execute (IP), and a pointer to an activation frame (FP). The instruction in the IP specifies an opcode (e.g., ADD), the offset in the activation frame where the match will take place (e.g., FP+3), and one or more destination instructions that will receive the result of the operation (e.g., IP+1). If synchronization operations are to be implemented on an ETS machine, software needs to manage a fixed node to handle the tokens and wake up stalled threads.

QOLB [10, 15] by Goodman *et al.* serializes 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 for the queue entries, a "shadow line" mechanism for local spinning, and a mechanism to perform direct node-to-node lock transfers.

Several recent clusters off-load synchronization operations from the main processor to network processors [13, 25]. Gupta *et al.* [13] use user-level one-sided MPI protocols to implement communication functions, including barriers. The Quadrics^{TM}QsNet interconnect used by the ASCI Q supercomputer [25] supports both a pure hardware barrier using a crossbar switch and hardware multicast, and a hybrid hardware/software tree barrier that runs on the network processors. For up to 256 participating processors, our AMO-based barrier performs better than the Quadrics hardware barrier. As background traffic increases and the system grows beyond 256 processors, we expect that the Quadrics hardware barrier will outperform AMO-based barriers. However, the Quadrics hardware barrier has two restrictions that limit its usability. First, only one processor per node can participate in the synchronization. Second, the synchronizing nodes must be adjacent. Their hybrid barrier does not have these restrictions, but AMO-based barriers outperform them by a factor of four in all of our experiments.

2.3 Time Complexity Analysis

In this section, we estimate the time complexity of the barrier and spinlock algorithms introduced in Section 2.1. We do not survey all existing barrier and spinlock algorithms, nor do we examine the algorithms on all (old and new) architectures. Instead, we use these commonly used algorithms to illustrate the effect of architectural/hardware features that have emerged in the past decade and to identify key features of real systems that are often neglected in previous analytical models of these algorithms.

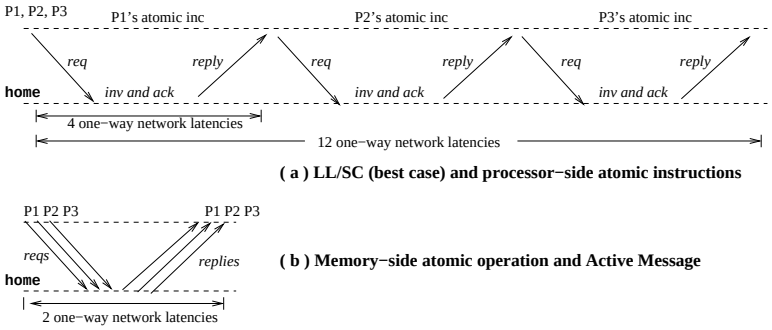


Figure 3. *Gather* **phase of a three-process barrier.**

It is customary to evaluate the performance of synchronization algorithms in terms of the number of remote memory references (*i.e.*, network latencies) required to perform each operation. With atomic RMW instructions, the *gather* stage latency of an N-process barrier includes 4N non-overlappable one-way network latencies, illustrated in Figure 3(a). To increment the barrier count, each processor must issue an exclusive ownership request to the barrier count's home node, which issues an invalidation message to the prior owner, which responds with an invalidation acknowledgement, and finally the home node sends the requesting processor an exclusive ownership reply message. If we continue to assume that network latency is the primary performance determinant, the time complexity of the *release* stage is $O(1)$, because the N invalidation messages and subsequent N reload requests can be pipelined. However, researchers have reported that memory controller (MMC) occupancy has a greater impact on barrier performance than network latency for medium-sized DSM multiprocessors [6]. In other words, the assumption that coherence messages can be sent from or processed by a particular memory controller in negligible time does not hold. If MMC occupancy is the key determinant of performance, then the time complexity of the *release* stage is $O(N)$, not $O(1)$.

Few modern processors directly implement atomic RMW primitives. Instead, they provide some variant of the LL/SC instruction pair discussed in Section 2.1. For small systems, the performance of barriers implemented using LL/SC instructions is close to that of barriers built using atomic RMW instructions. For larger systems, however, competition between processors causes a lot of interference, which can lead to a large number of backoffs and retries. While the best case for LL/SC is the same as for atomic RMWs ($4N$ message latencies), the worst case number of message latencies for the *gather* phase of an LL/SC-based barrier is $O(N^2)$. Typical performance is somewhere in between, depending on the amount of work done between barrier operations and the average skew in arrival time between different processors.

If the atomic operation functionality resides on the MMC, as it does for the Cray [28] and SGI [18] machines that support MAOs, a large number of non-overlapping invalidation and reload messages can be eliminated. In these systems, the RMW unit is tightly coupled to the local coherence controller, which eliminates the need to invalidate cached copies before updating `barrier_variable` in the *gather* phase. Instead, synchronizing processors can send their atomic RMW requests to the home MMC in parallel,

Hardware support	*Gather* stage	*Release* stage	Total
Processor-side atomic	$O(N)$	$O(1)$ or $O(N)$	$O(N)$
LL/SC best case	$O(N)$	$O(1)$ or $O(N)$	$O(N)$
LL/SC worst case	$O(N^2)$	$O(1)$ or $O(N)$	$O(N^2)$
MAO network latency bound	$O(1)$	$O(1)$	$O(1)$
MAO MMC occupancy bound	$O(N)$	$O(N)$	$O(N)$
ActMsg network latency bound	$O(1)$	$O(1)$	$O(1)$
ActMsg MMC occupancy bound	$O(N)$	$O(N)$	$O(N)$

Table 1. Time complexity of barrier with different hardware support.

and the MMC can execute the requests in a pipelined fashion (Figure 3(b)). Since the operations by each processor are no longer serialized, the time complexity of the *gather* phase MAO-based barriers is $O(N)$ memory controller operations or $O(1)$ network latencies, depending on whether the bottleneck is MMC occupancy or network latency. Since MAOs are performed on non-coherent (IO) addresses, the release stage requires processors to spin over the interconnect, which can by introduce significant network and memory controller load.

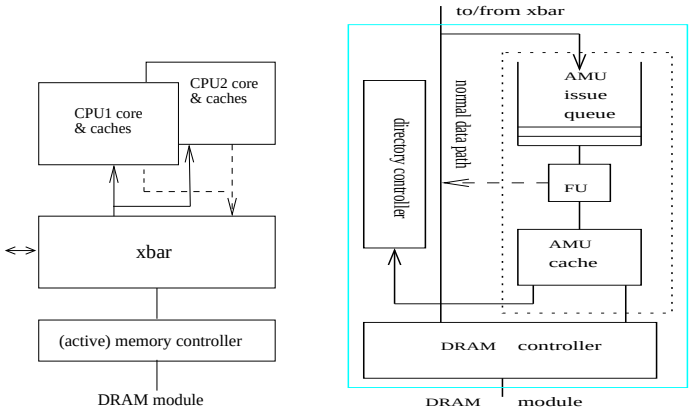
Barriers built using active messages are similar to those built with MAOs in that the atomic increment requests on a particular global variable are sent to a single node, which eliminates the message latencies required to maintain coherence. Both implementation strategies have the same time complexity as measured in network latency. However, active messages typically use interrupts to trigger the active message handler on the home node processor, which is a far higher latency operation than pipelined atomic hardware operations performed on the memory controller. Thus, barriers built using active messages are more likely to be occupancy-bound than those built using MAOs. Recall that when occupancy is the primary performance bottleneck, the *gather* phase has a time complexity of $O(N)$.

In terms of performance, spinlocks suffer from similar problems as barriers. In a program using ticket locks for mutual exclusion, a process can enter the critical section in $O(1)$ time, as measured in network latency, or $O(N)$ time, as measured in memory controller occupancy. A program using Anderson's array-based queuing lock has $O(1)$ complexity using either measurement method.

Table 1 summarizes our discussions of non-tree-based barriers. If we build a tree barrier from one of the above basic barriers, the $O(N)$ and $O(N^2)$ time complexity cases can be reduced to $O(log(N))$, albeit with potentially non-trivial constant factors. In Section 4.2.2, we revisit some of these conclusions to determine the extent to which often ignored machine features can affect synchronization complexity analysis.

3 AMU-Supported Synchronization

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



(a) *Node* (b) *AMU*

Figure 4. Hardware organization of the Active Memory Controller.

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 with poor temporal locality that are not accessed many times between when they are loaded into a cache and later evicted, e.g., synchronization variables. 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. Unlike the MAOs of Cray and SGI machines, AMOs operate on cache coherent data and can be programmed to trigger coherence operations when certain events occur.

Our proposed mechanism augments the MIPS R14K ISA with a few AMO instructions. These AMO instructions are encoded in an unused portion of the MIPS-IV instruction set space. Different synchronization algorithms require different atomic primitives. We are considering a range of AMO instructions, but for this study we consider only `amo.inc` (increment by one) and `amo.fetchadd` (fetch and add). Semantically, these instructions are identical to the corresponding atomic processor-side instructions, so programmers can use them as if they were processor-side atomic operations.

3.1 Hardware Organization

Figure 4 depicts the architecture that we assume. A per-node crossbar connects local processors to the network backplane, local memory, and IO subsystems. We assume that the processors, crossbar, and memory controller all reside on the same die, as will be typical in near-future system designs. Figure 4 (b) presents a block diagram of a 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 awaiting 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 (the FU in Figure 4 (b)).

Accesses to synchronization variables by the AMU exhibit high temporal locality because every participating process accesses the same synchronization variable, so our AMU design incorporates a tiny cache. This cache eliminates the need to load from and write to the off-chip DRAM for each AMO performed on a particular synchronization variable. Each AMO that hits in the AMU cache takes only two cycles to complete, regardless of the number of processors contending for the synchronization variable. An *N*-word AMU cache supports outstanding synchronization operations on *N* different synchronization variables. In our current design we model a four-word AMU cache. The hardware cost of supporting AMOs is negligible. In a 90nm process, the entire AMU consumes approximately 0.1mm^2, which is below 0.06% of the total die area of a high-performance microprocessor with an integrated memory controller.

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 coherence 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 acts as a signal to all waiting processes that the barrier operation has completed.

One way to optimize synchronization is to use a write-update protocol on synchronization variables. However, issuing a block update after each write generates an enormous amount of network traffic, which offsets the benefit of eliminating invalidation requests. In contrast, the "put" mechanism issues word-grained updates, thereby eliminating false sharing. For example, `amo.inc` only issues updates after the last process reaches the barrier rather than once every time a process reaches the barrier.

The "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

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

consistent memory model for barrier variables, where the condition of reaching a target value acts as the release point.

3.3 Programming with AMOs

With AMOs, atomic operations are performed at the memory controller without invalidating shared copies in processor caches. In the case of AMO-based barriers, the cached copies of the barrier count are updated when the final process reaches the barrier. Consequently, AMO-based barriers can use the naive algorithm shown in Figure 1(a) by simply replacing `atomic_inc(&barrier_variable)` with `amo_inc(&barrier_variable, num_procs)`, where `amo_inc()` is a wrapper function for the `amo.inc` instruction.

The `amo.fetchadd` instruction adds a designated value to a specified memory location, updates the shared copies in processor caches with the new value, and returns the old value. To implement spinlocks using AMOs, we replace the atomic primitive `fetch_and_add` in Figure 2 with the corresponding AMO instruction, `amo_fetchadd()`. We also use `amo_fetchadd()` on the counter to take advantage of the "put" mechanism. Note that using AMOs eliminates the need to allocate the global variables in different cache lines. Like ActMsg and MAOs, the time complexity of AMO-based barriers and spinlocks is $O(1)$ measured in terms of either network latency and $O(N)$ in terms of memory controller occupancy. However, AMOs have much lower constants than ActMsg or MAOs, as will be apparent from the detailed performance evaluation.

The various architectural optimizations further reduce the constant coefficients.

Using conventional synchronization primitives often requires significant effort from programmers to write correct, efficient, and deadlock-free parallel codes. For example, the Alpha Architecture Handbook [7] dedicates six pages to describing the LL/SC instructions and restrictions on their use. On SGI IRIX systems, several library calls must be made before actually calling the atomic op function. To optimize performance, programmers must tune their algorithms to hide the long latency of memory references. In contrast, AMOs work in the 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 show in Section 4 that AMOs negate the need to use complex algorithms such as combining tree barriers and array-based queuing locks even for fairly large systems. Since synchronization-related codes are often the hardest portion of a parallel program to code and debug, simplifying the programming model is another advantage of AMOs over other mechanisms.

4 Performance Evaluation

4.1 Simulation Environment

We use a cycle-accurate execution-driven simulator, UVSIM, in our performance study. UVSIM models a hypothetical future-generation SGI Origin architecture, including a directory-based coherence protocol [30]

Parameter	Value
Processor	4-issue, 48-entry active list, 2GHz
L1 I-cache	2-way, 32KB, 64B lines, 1-cycle latency
L1 D-cache	2-way, 32KB, 32B lines, 2-cycle latency
L2 cache	4-way, 2MB, 128B lines, 10-cycle latency
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-cycle latency
Network	100 processor-cycle latency per hop

Table 2. System configuration.

that supports both write-invalidate and fine-grained write-update, as described in Section 3.2. Each simulated node contains two hypothetical next-generation MIPS microprocessors connected to a high-bandwidth bus. Also connected to the bus is a hypothetical future-generation Hub [31], which contains the processor interface, memory controller, directory controller, network interface, IO interface, and active memory unit.

Table 2 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 and 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 employs 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 directly executes statically linked 64-bit MIPS-IV executables and includes a micro-kernel that supports all common system calls. UVSIM supports the OpenMP runtime environment. All benchmark programs used in this paper are OpenMP-based parallel programs. All programs are tuned to optimize performance on each modeled architecture and then compiled using the MIPSpro Compiler 7.3 with an optimization level of "`-O2`".

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 employ four representative synchronization algorithms (a simple barrier, a combining barrier tree, ticket locks, and array-based queuing locks) to evaluate the impact of architectural features on synchronization performance. The simple barrier function comes from the SGI IRIX OpenMP library. The software combining tree barrier is based on the work by Yew *et al.* [33]. Ticket locks and array-based queuing locks are based on the pseudocode given in [21].

Nodes	CPUs	Speedup over LL/SC barrier			
		ActMsg	Atomic	MAO	AMO
2	4	0.95	1.15	1.21	**2.10**
4	8	1.70	1.06	2.70	**5.48**
8	16	2.00	1.20	3.61	**9.11**
16	32	2.38	1.36	4.20	**15.14**
32	64	2.78	1.37	5.14	**23.78**
64	128	2.74	1.24	8.02	**34.74**
128	256	2.82	1.23	14.70	**61.94**

Table 3. Performance of different barriers.

The starting time of all processes in each benchmark are synchronized. The elapsed time of each test is reported as the average over ten invocations of the benchmark, i.e., for `P` processors the total number of simulated spinlock acquisition-release pairs is `10` $\times$ `P`. An empty critical section is used in the spinlock experiments.

We consider five different mechanisms for implementing each synchronization operation: LL/SC/instructions, conventional processor-side atomic instructions ("Atomic"), existing memory-side atomic operations ("MAOs"), software active messages ("ActMsg"), and AMOs. The LL/SC-based versions are used as the baseline. The AMU cache is used for both MAOs and AMOs. We assume fast user-level interrupts are supported, so our results for active messages are somewhat optimistic compared to typical implementations that rely on OS-level interrupts and thread scheduling.

4.2.1 Non-tree-based barriers

Table 3 presents the speedups of four different barrier implementations compared to conventional LL/SC-based barriers. A speedup of less than one indicates a slowdown. We vary the number of processors from four (*i.e.*, two nodes) to 256 (128 nodes), the maximum number of processors allowed by the directory structure [30] that we model. The ActMsg, Atomic, MAO, and AMO barriers all perform and scale better than the baseline LL/SC version. When the number of processors is greater than 8, active messages outperform LL/SC by a factor 1.70 (8 processors) to 2.82 (256 processors). 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 barrier performance dwarfs that of all other variants, ranging from a factor of 2.10 at four processors to a factor of 61.94 for 256 processors. The raw time for AMOs are 456, 552, 1488, 2272, 4672, 6784 and 9216, for 4, 8, 16, 32, 64, 128 and 256 processors, respectively, measured in clock periods of a 2GHz CPU.

In the baseline implementation, each processor loads the barrier variable into its local cache before incrementing it using LL/SC instructions. Only one processor succeeds at one time; other processors 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 of the base LL/SC barrier implementation increases dramatically as the number of nodes increases. This effect

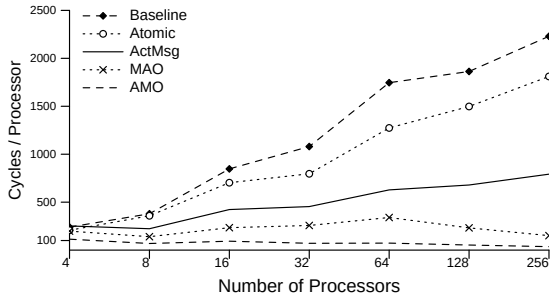


Figure 5. Cycles-per-processor of different barriers.

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. However, 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, the performance gains are relatively small because processor-side atomic operations still requires a round trip over the network for every atomic operation, all of which must be performed serially.

MAO-based barriers perform and scale significantly better than barriers implemented using processor-side atomic instructions. At 256 processors, MAO-based barriers outperform the baseline LL/SC-based barrier by nearly a factor of 15. This result further demonstrates that putting computation near memory improves synchronization performance.

AMO-based barriers are four times faster than MAO-based barriers. This performance advantage derives from the "delayed update" enabled by the test value mechanism and use of a 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. We do not assume that the network has multicast support; AMO performance would be even better if the network supported such operations.

Roughly speaking, the time to perform an AMO barrier equals $(t_o + t_p \times P)$, where t_o is a fixed overhead, 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, i.e., the total barrier time per processor is constant, which is apparent in Figure 5. In fact, the per-processor latency drops off slightly as the number of processors increases because the fixed overhead is amortized by more processors. In contrast, the per-processor time grows as the system grows for the other implementations.

4.2.2 Barrier performance: a case of potential mis-estimates

In Section 2.3, we concluded that the time complexity of barriers built using processor-side atomic instructions is $O(N)$, as measured in terms of the number of remote memory references. Given this estimate, we would expect the per-processor time (total time divided by the number of processors) to be constant. However, the experimental results visible in Figure 5 clearly contradict this expectation. ActMsg, for which we correctly introduced an extra performance limiting parameter (MMC occupancy), also performs worse than what we projected. The per-processor barrier time of ActMsg increases as the number of participating processors increases.

The reason for these underestimates of time complexity is that in a non-idealized (real) system, factors other than network latency and memory controller occupancy can extend the critical path of program execution. Such factors include system bus contention, cache performance, an increase in one-way network latency as a system grows, queuing delays in the network interface, limited numbers of MSHRs, and retransmissions caused by network interface buffer overflows. At first glance, each of these additional factors might appear to be unimportant. However, our simulations reveal that any of them can be a performance bottleneck at some point in the lifetime of a synchronization operation. Ignoring their accumulative effect can easily lead one to overestimate the scalability of a given synchronization algorithm.

Up until now, typical synchronization complexity analyses have assumed that remote memory references can be performed in a constant amount of time. In reality, very few large-scale multiprocessors support a full crossbar interconnect, but rather employ on some kind of tree topology. As a result, remote memory references are not constant, but instead depend on the distance between the source and the destination nodes. For example, SGI Origin systems use an oct-tree in the intermediate routers between different nodes. The cost of one remote memory reference is no less than `2*floor(`$log_m(N)$`)`, where m is the factor of the tree and N is the number of nodes that the source and destination of the message span.

While our measured relative performance of the different barrier implementations did not completely invalidate our analytical estimates, the disparity between real system performance and analytical extrapolation was significant. Given this experience, we caution against using over-simplified when analyzing the performance of synchronization operations. On a complex computer system such as a modern cc-NUMA multiprocessor, over-simplified system models can lead to incorrect or inaccurate assumptions as to what factors determine performance. We strongly recommend that researchers and practitioners always perform a validation check of their analytical performance predictions when possible, preferably through either real system experiments or accurate software simulation.

4.2.3 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

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 4. Performance of tree-based barriers.

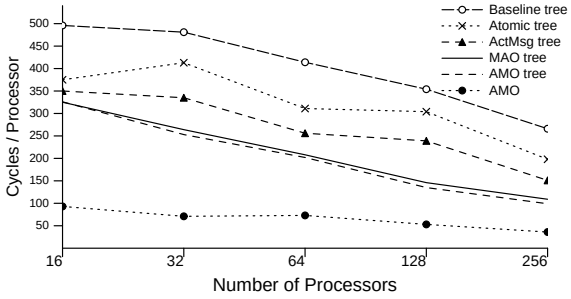


Figure 6. Cycles-per-processor of tree-based barriers.

that we consider for tree-based barriers has 16 processors. Table 4 shows the speedups of tree-based barriers over the baseline (flat LL/SC-based) barrier implementation. Figure 6 shows the number of cycles per processor for the 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.* [22]. On a 256-processor system, all tree-based barriers are at least eight times faster than the baseline barrier implementation. As can be seen in Figure 6, the number of cycles-per-processor 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*. [19] demonstrated that improper use of trees can drastically degrade the performance of tree-based barriers to even below that of simple flat barriers. Nonetheless, our simulation results demonstrate the performance potential of tree-based barriers.

Despite all of their advantages, tree-based barriers still significantly underperform a flat AMO-based barrier implementation for on number of nodes that we study. For instance, the best non-AMO tree-based barrier (MAO + tree) is three times slower than a flat AMO-based barrier on a 256-processor system. Interestingly, AMO-based barrier trees underperform flat AMO-based barriers in all tested configurations. AMO-based barriers include 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. The fact that AMOs alone are better than the combination of AMOs and trees is an indication that AMOs do not require heroic programming effort to achieve good performance. However, we expect that tree-based barriers using AMOs will outperform flat AMO-based barriers if consider systems with tens of thousands

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	1.79	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 5. Speedups of different locks over the LL/SC-based locks.

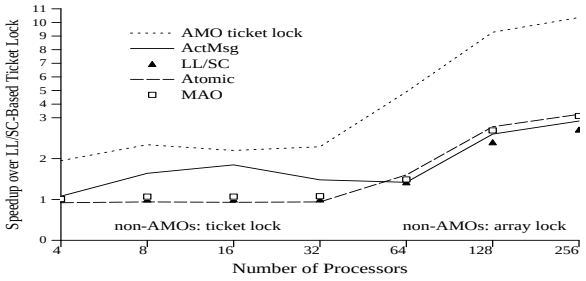


Figure 7. Performance of the different locks

processors. Determining whether or not tree-based AMO barriers can provide extra benefits on such very large-scale systems is part of our future work.

4.2.4 Spin locks

Table 5 presents the speedups of the different ticket lock and array-based queuing lock implementations compared to LL/SC-based ticket locks.

Using traditional hardware mechanisms, ticket locks outperform array-based queueing locks when the system has 32 or fewer processors. Array-based queueing locks outperform ticket locks for larger systems, which verifies the effectiveness of array-based locks at alleviating hot spots in larger systems.

AMOs greatly improve the performance of both styles of locks and negate the performance difference between ticket locks and array-based locks. This observation implies that if AMOs are available we can use simple locking mechanisms (ticket locks) without losing performance.

On a machine that does not support AMOS, the programmer could use the better of the two algorithms depending on the system size to obtain the best performance. Figure 7 plots the performance of the best-performing lock implementation (ticket or array) for each implementation strategy (LL/SC, atomic, MAOs, AMOs, or ActMsg) as we vary the number of processors. For non-AMO platforms, this translates into using ticket locks for up to 32 processors and array-based locks for larger systems.

Figure 7 helps us differentiate the benefits achieved by providing hardware synchronization support from the benefits achieved by the array-based queue lock algorithm. In the range of 4 to 32 processors, ActMsg noticeably outperforms the other conventional implementations. Otherwise, the curves of the various tradi-

tional lock implementations (LL/SC, ActMsg, Atomic and MAOs) track closely despite their vastly different implementations. This convergence reveals that (1) under low contention, the four non-AMO implementations are similar in efficiency except for ActMsg, and (2) the superiority of array-based locks in larger systems derives from the algorithm itself, rather than from the architectural features of the underlying platforms.

The relative performance of AMO-based ticket locks compared to the LL/SC ticket locks keeps increasing until it reaches 10.36 at 256 processors. Thus, it appears that the performance advantage of AMO locks is a result of the unique design of AMOs rather than the implementation strategy.

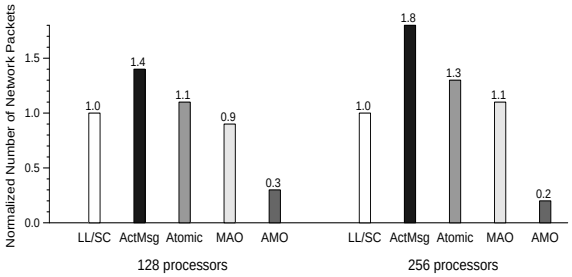


Figure 8. Network traffic for ticket locks.

A major reason that AMO-based locks outperform the alternatives is the greatly reduced network traffic. Figure 8 shows the network traffic, normalized to the baseline LL/SC version, of different ticket lock implementations on 128-processor and 256-processor systems. In both systems, AMO-based ticket locks generate significantly less traffic than other approaches. Interestingly, active message-based locks, which are motivated by the need to eliminate remote memory accesses, require more network traffic than the other approaches because the high invocation overhead of the message handlers leads to timeouts and retransmissions on large systems with high contention.

5 Conclusions

To improve synchronization efficiency, we first identify and analyze the deficiencies of conventional barrier and spinlock implementations. We demonstrate how apparently innocent simplifying assumptions about the architectural factors that impact performance can lead to incorrect analytical results when analyzing synchronization algorithms. Based on these observed problems, we strongly encourage complete system experiments, either via in-situ experiments or highly detailed simulation, when estimating synchronization performance on modern multiprocessors.

To improve synchronization performance, we propose that main memory controllers in multiprocessors be augmented to support a small set of active memory operations (AMOs). We demonstrate that AMO-based barriers do not require extra spin variables or complicated tree structures to achieve good performance. We further demonstrate that simple AMO-based ticket locks outperform alternate implementations using

more complex algorithms. Finally, we demonstrate that AMO-based synchronization outperforms highly optimized conventional implementations by up to a factor of 62 for barriers and a factor 10 for spinlocks.

In conclusion, we have demonstrated that AMOs overcome many of the deficiencies of existing synchronization mechanisms, enable extremely efficient synchronization at rather low hardware cost, and reduce the need for cache coherence-conscious programming. As the number of processors, network latency, and DRAM latency grow, the value of fast synchronization will only grow in time, and among the synchronization implementations we studied, only AMO-based synchronization appears able to scale sufficiently to handle these upcoming changes in system performance.

Acknowledgments

The authors would like to thank Silicon Graphics Inc. for the technical documentations provided for the simulation, and in particular, the valuable input from Marty Deneroff, Steve Miller, Steve Reinhardt, Randy Passint and Donglai Dai. We would also like to thank Allan Gottlieb for his feedback on this work.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. of the 1990 ICS*, pp. 1–6, Sept. 1990.

[2] 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.

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

[4] Arvind, R. Nikhil, and K. Pingali. I-Structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):589–632, 1989.

[5] 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.

[6] M. Chaudhuri, M. Heinrich, C. Holt, J. Singh, E. Rothberg, and J. Hennessy. Latency, occupancy, and bandwidth in DSM multiprocessors: A performance evaluation. *IEEE Trans. on Computers*, 52(7):862–880, July 2003.

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

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

[9] E. Freudenthal and A. Gottlieb. Debunking then duplicating Ultracomputer performance claims by debugging the combining switches. In *Workshop on Duplicating, Deconstructing, and Debunking*, pp. 42–52, June 2004.

[10] 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.

[11] 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.

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

[13] 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.

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

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

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

[17] J. T. Kuehn and B. J. Smith. The Horizon supercomputing system: Architecture and software. In *Proc. of the 1988 ICS*, pp. 28–34, 1988.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] G. Papadopoulos and D. Culler. Monsoon: An explicit token store architecture. In *Proc. of the 17th ISCA*, pp. 82–91, 1990.

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

[26] T. Pinkston, A. Agarwal, W. Dally, J. Duato, B. Horst, and T. B. Smith. What will have the greatest impact in 2010: The processor, the memory, or the interconnect? HPCA8 Panel Session, `available at http://www.hpcaconf.org/hpca8/`, Feb. 2002.

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

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

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

[30] Silicon Graphics, Inc. *SN2-MIPS Communication Protocol Specifi cation, Revision 0.12*, Nov. 2001.

[31] Silicon Graphics, Inc. *Orbit Functional Specifi cation, Vol. 1, Revision 0.1*, Apr. 2002.

[32] 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.

[33] 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.