This paper appears in the 9th International Conference on Supercomputing, July, 1995. Reprinted by permission of ACM.

Techniques for Reducing Overheads of Shared-Memory Multiprocessing

Alain Kägi, Nagi Aboulenein, Douglas C. Burger, James R. Goodman

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706 USA
{alain, dburger, goodman}@cs.wisc.edu, nagi@cs.uno.edu

Abstract

The fine-grain nature of shared-memory multiprocessor communication introduces overheads that can be substantial. Using the Scalable Coherent Interface (SCI) as a base hardware platform and the SPLASH benchmark suite for applications, we analyze three techniques to reduce this overhead: (i) efficient synchronization primitives, and in particular a hardware primitive called QOLB; (ii) weakened memory ordering constraints; and (iii) optimization of the cache-coherence protocol for two nodes sharing data. We perform simulations both for current technology and technology that we anticipate will be available five years hence. We find that QOLB (of which this study performs the first detailed simulations) shows a large and consistent improvement, much larger than that predicted by Mellor-Crummey and Scott [19]. The relaxation of memory ordering constraints also provides a consistent performance improvement. In accordance with prior results, we show that a more aggressive memory model produces more substantial performance improvements. The optimization for two-node sharing shows mixed results, correlating unsurprisingly with the presence of that sharing pattern in an application. Our most important results are (i) that the overheads eliminated with these optimizations are largely orthogonal—the performance gains from supporting multiple optimizations concurrently are for the most part additive—and (ii) that technological improvements increase both these overheads and the success of the optimizations at reducing them.

1 Introduction

The shared-memory multiprocessing paradigm has had substantial impact in academic circles, but has only established itself with small-scale machines in the industrial community. The increasing reliance of supercomputer manufacturers on commodity parts has contributed to the lack of large-scale shared-memory machines, as no such shared-memory parts have previously been available. The establishment of standards—such as the IEEE Scalable Coherent Interface [1]—has resulted from this growing industrial inertia. Consequently, parts are becoming available that integrate entire aspects of these standards, reducing system design complexity, time-to-market, and total system cost. Convex based their Exemplar system [8], for instance, largely on third-party components.

The growing number of bus-based shared-memory systems will further strengthen the success of the shared-memory processing paradigm. The increasing prevalence of these systems will create a large base of parallel applications, which should ease the acceptance of larger-scale shared-memory systems.

The shared-memory model has persuasive advantages. It provides a uniform global address space, transparent communication of data, and relative ease of programming. However, the fine-grain nature of its communication introduces overhead that can be substantial. The bulk of this overhead comes from synchronization and the maintenance of data coherence, making the support of these two operations critical for the efficient execution of shared-memory applications.

This paper explores three classes of optimizations aimed at reducing shared-memory overheads: (i) sophisticated synchronization primitives, (ii) improved processor utilization through weakened memory ordering constraints, and (iii) optimized coherence latencies for common data sharing patterns. Although these optimizations are all options of SCI, the results of this study are more generally applicable.

The first class of optimizations we study is improved synchronization primitives. A straightforward approach to building synchronization functions uses instructions provided by the commodity microprocessor (such as the atomic **`swap`** instruction in the SPARC instruction set [10]) in much the same way as uniprocessor platforms use them. Typically, the processor accesses a lock repeatedly until the processor finds it unlocked. On a multiprocessor, these repeated accesses often translate directly into network traffic that leads to heavy network contention and potentially severe performance degradation. We therefore compare two more advanced primitives, *QOLB* [25] and *MCS locks* [19].

The second class of optimizations that we examine consists of a range of memory models. Programmers naturally assume a memory model formally called *sequential consistency*, defined by Lamport [17]. The strict ordering of sequential consistency severely limits concurrency of memory operations in a parallel computer. Most memory operations could in fact be completed out of order without affecting the program result. The identification of those which could affect the result is difficult, however, as synchronization through shared variables can be extremely subtle. Memory models—such as data-race-free-0 [3] and release consistency

This work is supported in part by NSF Grant CCR-9207971, an unrestricted grant from the Apple Computer Advanced Technology Group, and donations from Thinking Machines Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618, with matching funding from the University of Wisconsin Graduate School.

© 1995 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or <permissions@acm.org>.

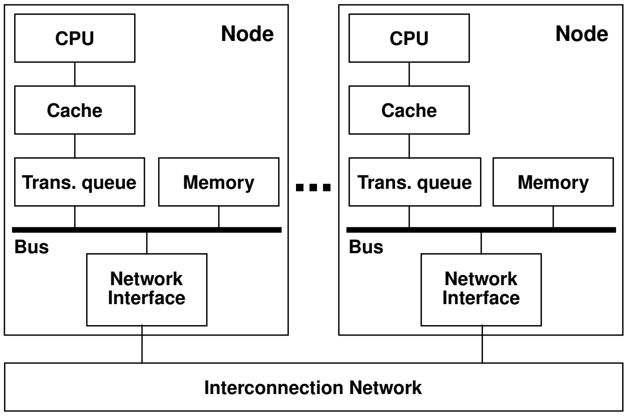

Figure 1. Target system

[13]—allow the system to relax the constraints of sequential consistency by creating a contract between the software and the hardware, defining what memory orderings are legal. Implementations may then be conservative or aggressive in supporting the memory model.

The final optimization we study reduces overhead for one particular pattern of data sharing. General cache-coherence protocols may not optimally handle the majority of common data sharing patterns. Consequently, researchers have investigated many protocol extensions [7, 9, 11, 24] that allow existing protocols to perform better under specific classes of data sharing patterns. These extensions attempt to reduce network traversals and/or memory accesses. Two common classes of sharing patterns are migratory data [16] and producer-consumer data sharing. In this paper we study only *pairwise sharing*—an optimization aimed at two-node sharing—primarily because pairwise sharing is a feature of the SCI cache-coherence protocol.

In addition to the three classes of optimizations, we quantify the effects of two additional related issues: optimization interaction and technological advances. An important goal of this work is determining the level of overlap that occurs among our three optimization classes. Thus, in addition to focusing on the effectiveness of each optimization independently, we also analyze the cross-product of their interactions.

A significant component of network latencies for multiprocessors is the fundamental delay associated with the speed of light. While there will undoubtedly be further reductions in network delays, we expect that continuing improvements in processor technology will far exceed these network improvements. We wish to understand how these technology trends will affect the relative benefits of system optimizations. While we are unable to accurately model the performance of future processors, we believe that raw performance (as measured, for example, by the time to complete a uniprocessor application), will increase relative to improvements in network speeds. Therefore, we have attempted to estimate performance by adjusting relative delays for instruction times versus memory access times and network delays.

The main contributions of this paper are threefold. We present the first quantitative performance analysis of the QOLB synchronization primitive [25]. We analyze the subtle interaction of QOLB and other synchronization primitives with weakened memory ordering constraints. Finally, we demonstrate the effect of advancing technology on the three machine optimizations in general, with an emphasis on QOLB.

We have organized the remainder of this paper as follows: Section 2 describes our simulation environment—the Wisconsin Wind Tunnel—and provides pertinent background information on the Scalable Coherence Interface, synchronization primitives, relaxed consistency model issues, pairwise sharing, and the application benchmarks that we used. Section 3 defines the simulated system, discusses our simulation assumptions, lists the experiments performed, and describes the experimental methodology. Section 4 presents our experimental results, Section 5 reviews some important related work, and Section 6 summarizes our conclusions and contributions.

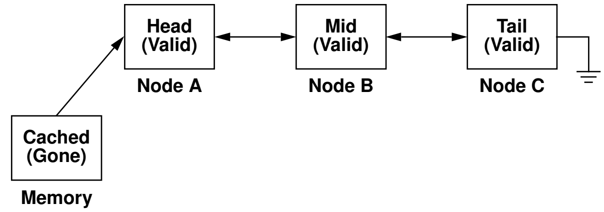

Figure 2. Example of an SCI sharing list

2 Background

Our simulation platform is the Wisconsin Wind Tunnel (WWT), which allows us to study the performance of large-scale, high-performance systems through massive simulation (more than two trillion cpu cycles). In order to establish a complete and detailed environment, we simulate a target system of 32 nodes providing hardware-guaranteed cache coherence by means of the ANSI/IEEE standard 1596 Scalable Coherent Interface (SCI) [1]. The target system consists of workstation-like nodes possessing a processor, cache memory, transaction queue (similar to a functionally-extended write buffer), network interface, and some fraction of the distributed, globally-shared memory with the associated directory entries (see Figure 1). Our target applications are five programs selected from the SPLASH benchmark suite [23].

2.1 The Scalable Coherent Interface

SCI defines both an interface to a network and a cache coherence protocol. The protocol is a robust *hardware* solution to the cache coherence problem. Messages may be addressed to any node in the system, but the protocol does not use broadcast. The protocol is able to survive the loss of packets in the network with no loss of data. SCI guarantees forward progress for all transactions, and was intended to support high performance on a full range of system sizes, from tens to thousands of processors.

SCI specifies a distributed, directory-based cache-coherence protocol. In addition to specifying that the physical memory is distributed across the nodes, the list of nodes holding copies of a given cache line is also distributed, along with the cached copies. The protocol creates a doubly-linked list to maintain the directory information for each valid cached line. The home node's memory directory keeps a pointer to the last requester. An example of an SCI sharing list is shown in Figure 2.

SCI defines the following basic operations on a sharing list: (i) a node may join the list, becoming the head; (ii) a node in the list may delete itself from the list by serially communicating with its upstream and downstream neighbors, informing each in turn of its new neighbor; (iii) a head node may purge (i.e. invalidate) all the other elements of the list (one at a time) to become a single-element list. SCI carefully defines these operations to permit arbitrary concurrency in their execution.

To maintain a local copy of a cache line, a node executes operation (i). It receives a copy of the data either from the home node's memory or from the old sharing-list head. If a node wishes to modify the line (a store instruction), it must perform operations (ii) and

```
struct _locked_data {
int lock;
int data[15]; };

void critical_section(struct _locked_data *x)
{
   EnQOLB(x->lock);   /* Prefetch lock, data */
   /* Various computation here */
   while (! EnQOLB(x->lock));         /* Spin */
   /* Critical section using x->data */
   DeQOLB(x->lock);           /* Release lock */
}
```

Figure 3. QOLB code example

(i) if it is not at the head of the list; once it is at the head of the list it can perform operation (iii).

Standardization efforts must be certain to make standards cost-effective using current technology, without compromising potential high performance for future systems. SCI provides such flexibility by containing numerous optional features, which can provide a range of systems with varied cost and performance. Two such options are efficient support for two-node sharing (pairwise sharing) and efficient support for synchronization (QOLB).

2.2 Synchronization

2.2.1 QOLB

Naive synchronization primitives can drastically increase network traffic in multiprocessor systems. Goodman, Vernon, and Woest proposed the Queue-On-Lock-Bit primitive (*QOLB*—originally called QOSB) as a hardware solution to this problem [25]. QOLB provides a direct implementation of a binary semaphore (with approximate first-come-first-serve service) by building a hardware queue of waiting processors. It avoids unnecessary network traffic as waiting processors spin locally, repeatedly accessing a local "shadow" copy of the lock's cache line *without* generating network traffic. Consequently, the placing of a lock and data in the same cache line does not degrade performance, unlike other synchronization primitives.

When the current holder of a given lock releases that lock, the cache line containing both the lock and the corresponding data is automatically sent directly to the next processor in the queue. This technique substantially reduces latency associated with handing protected data to the next waiting processor. In fact, QOLB reduces the synchronization overhead to the theoretical minimum, unless the system speculatively overlaps critical sections. In addition, QOLB is a non-blocking primitive—it allows the waiting processor to overlap data prefetching with other useful work.

Figure 3 shows an example of how QOLB is used to access data in a critical section. The first call to **`EnQOLB`** (a non-blocking operation) allocates a shadow copy of the line and sends a message that inserts the node into the hardware requester queue. This allows the processor to overlap the fetch time with useful computation. The subsequent calls to **`EnQOLB`** in the loop spin locally until the owner releases the lock and sends it directly to the waiting node. When **`EnQOLB`** returns true, the processor enters the critical section. The processor relinquishes the lock with the call to **`DeQOLB`**, at which point both the lock and any data in the lock's cache line are sent directly to the next waiting processor. Note that in this example, the data associated with the lock are sufficiently small in number that no extra remote accesses need to be made once the lock is prefetched. This will obviously not always be the case.

2.2.2 MCS locks

Mellor-Crummey and Scott proposed software solutions to minimize network traffic and synchronization access latencies, as did Anderson. Mellor-Crummey and Scott (MCS) implement a queue as a linked list, and use atomic operations such as *swap* or *compare-and-swap* to update the list correctly [18]. Anderson presented a scheme that implements a queue as a circular array [4]. Inspired by QOLB, these algorithms also reduce the network traffic to a constant number of traversals per synchronization access, allowing processors to spin locally while waiting for the release of the lock. Unlike QOLB, these algorithms cannot benefit from placing the lock and data in the same cache line, since the linked list along which the lock is passed is composed of different addresses. These algorithms are also unable to prefetch data without extending them and adding significantly to their complexity.

Aboulenein *et al.* [2] showed that Anderson's solution performs no better than the MCS solution; therefore in this study we restrict ourselves to comparing MCS with QOLB. If the synchronization contention is low, the use of a naive synchronization algorithm (as mentioned above) may lead to better performance than the MCS solution, as the latter has a large overhead per synchronization access even in the absence of contention. We will therefore also evaluate simple locks, in an attempt to identify the cases where simple locks outperform MCS and/or QOLB.

2.3 Memory ordering

Lamport defined sequential consistency as follows:

> [A memory system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [17].

Sequential consistency is overly restrictive with respect to multiprocessor memory orderings. Systems can achieve higher performance by relaxing the memory orderings, without compromising the correctness of the program. The class of weakened consistency models that we implemented belong to the *release consistency* model [13], which divides groups of memory accesses with *acquire*, *release*, and *special accesses*. So long as a program obeys the rules specified by this model, many memory accesses can bypass others, allowing the processor to tolerate the longer latencies associated with remote transactions.

Our benchmarks assume a memory system that supports release consistency. We study three different implementations, all of which satisfy the constraints of release consistency: (i) sequential consistency, (ii) a more relaxed consistency model that exploits the easily-obtained concurrency inherent in the SCI cache-coherence protocol, and (iii) an aggressive implementation that attempts to minimize the number of times that the processor must be stalled for memory operations.

This range of consistency models is perfectly compatible with SCI, which defines a network interface, not a system implementation. SCI does not specify the memory model; instead, it permits a variety of models to be supported, including sequential consistency. The SCI standard allows the processor to continue execution concurrently while multiple, sequential transactions are in progress. This flexibility allows us to evaluate a range of memory models.

2.4 Data sharing patterns

In the base SCI protocol, the delay associated with two-element sharing can be substantial. For a producer-consumer relationship—where one node writes a cache line frequently and another

Benchmark	Type of simulation	Input
Barnes	Barnes-Hut N-body	2048 bodies, 11 iterations
Mp3d	Hypersonic flow	24000 mols, 25 iterations
Ocean	Hydrodynamic	98 x 98, 2 days
Pthor	Digital circuit	`risc`, 1000 timesteps
Water	Water molecule	288 mols, 10 iterations

Table 1: Benchmarks

reads it—the protocol repeatedly creates and breaks down the sharing list. The writer must become the head of the list and then purge the reader. To read again, the reader must rejoin the list as the head. In order for the writer to write again, it must remove itself from the list, become the head, then purge the reader. Thus, the transfer requires numerous transactions, including two inquiries to the home node.

An optional extension to the SCI protocol addresses this problem. Known as *pairwise sharing*, this extension permits the tail node of a two-element list to modify its copy of the cache line after notifying the head node to mark its copy **`stale`**. The list is maintained even though the head no longer has a valid copy of the data. The stale head can then read (or write) the data by simply communicating with the tail (which becomes stale in case of a write) and reasserting itself as head. Thus, no enquiries to the home node are necessary for stable pairwise sharing.

The protocol was constructed in a way that pairwise sharing, if supported, degrades gracefully into a conventional list; if the list consists only of two nodes, it employs pairwise sharing. If a third node joins the list, the protocol reverts back to the standard method. SCI accomplishes this transition without extra messages in most cases, although there are circumstances where performance suffers because pairwise sharing defers the breaking-down of a two-element list when a third requester joins the list.

2.5 Wisconsin Wind Tunnel

Our experiments were performed on the Wisconsin Wind Tunnel virtual prototyping system [20]. WWT executes parallel shared-memory programs on a parallel message-passing computer (the *host*). It uses execution-driven, distributed, discrete-event simulation techniques that accurately calculate program execution time. Execution occurs in fixed windows of time, called quanta, which the simulator alternates with synchronization events that maintain causality. WWT exploits similarities between the target system and the host system to allow the host to execute directly all target program instructions and memory references that hit in the target cache. WWT accomplishes this by using the error-correcting codes on the CM-5 to trap into the simulator when a miss occurs in a target cache. Because of the direct execution and efficient traps, the scale of applications that WWT is able to simulate approaches that of real applications.

2.6 Benchmark applications

The target programs we used for our experiments are five applications drawn from the SPLASH benchmark suite. They are **`Barnes`**, **`Mp3d`**, **`Ocean`**, **`Pthor`**, and **`Water`** (see Table 1). A description of these benchmarks appears in the original article [23]. We focus the following discussion on specifics related to our study.

We labeled all memory accesses as aggressively as possible according to the structure proposed by Gharachorloo *et al.* [13]. We then inserted memory fences to achieve release consistency on our simulated hardware platform. The memory fences are consistent with those proposed by Gharachorloo [12, 13].

Technology:	Current	Future
Node parameters		
Processor speed	200 MHz	500 MHz
Sustained IPC	1	2
Cache access	3	12
Directory access	10	40
Network parameters		
Network bandwidth	500 MB/s	1 GB/s
Parse delay	4	6
Wire delay	3	8
Agent delay	22	28
Staging delay	14	28

Table 2: Parameter settings

All delays are in CPU cycles. The network is assumed to have two-byte-wide links. Parsing delay accounts for the time spent on the routing decision; wire delay accounts for buffering and multiplexing; agent delay accounts for dimension switching delays; and staging delay accounts for data transfers occurring at the source and target nodes.

We performed additional optimizations on each benchmark to maximize performance on the simulated hardware. For simulations evaluating QOLB, data structures were modified to couple locks in the same line with the data that they protect. We padded data in each benchmark, where necessary, to eliminate false sharing [14]. We compiled the benchmarks using GCC version 2.6.2 with the option **`-O3`**.

Barnes originally used locks to protect the higher levels of the tree during its tree-building phase, which results in an often-locked root. The designers of Barnes optimized this process by locking tree nodes only for writing, permitting read sharing of write-locked nodes. This optimization is critical for good performance of MCS locks, but is not essential for good performance of QOLB. We therefore turned the optimization off for all QOLB experiments.

We modified Ocean by converting it from FORTRAN to C (WWT currently does not support applications written in FORTRAN). To reduce conflicts in the cache we skewed storage by embedding the working arrays in larger arrays of prime size (131 by 131 elements).

Mp3d can be run with or without locks. Elimination of locks results in a non-deterministic execution, precluding repeatability of results, but potentially permitting higher performance. Because of competitive accesses to shared variables, however, the non-locking version requires great care to assure that it executes correctly on hardware only guaranteeing release consistency. This drawback negates much of the potential benefit of this technique. We used only the locking version of Mp3d for our experiments.

3 Methodology

3.1 Assumptions

Our simulation host is a Thinking Machines CM-5 partition with 32 processors. The simulation target is a 32-node shared-memory multiprocessor supporting the SCI cache-coherence protocol. WWT executes SPARC binaries, and assumes fixed execution time for the instructions (the actual values correspond to the instruction delays listed by the CY7C601 SPARC user's guide

Benchmark	Simulated		Modeled	
	Latency	% Error	Latency	% Error
Barnes	93	-1.18	88	2.12
Mp3d	92	0.26	89	1.72
Ocean	93	1.46	93	3.08
Pthor	102	-2.09	88	4.61
Water	91	-0.27	87	-0.04

Table 3: Inaccuracies of the constant latency network model

We compare execution time using the detailed network simulator to determine execution times using two different constants for the network latency: (i) the mean message latency returned by the simulator, and (ii) the mean message latency returned by our analytical network model.

[10]). The execution times for **`EnQOLB`** and **`DeQOLB`** (executed at the entry and the exit of a critical section) are 3 and 2 cycles respectively. Both instruction and stack accesses to the cache are not simulated in WWT; they are assumed always to hit. Our MCS locks use the **`swap`** primitive and perform all operations locally as much as allowed by the algorithm.

3.1.1 Memory system parameters

The invariant simulation parameters are as follows: a 64-byte line size, consistent with the SCI standard, a 1 cycle hit time for the caches, and a 1 cycle per 32-bit word fill time. The caches were 4-way set-associative. Replacements were selected based on which line least recently missed or write-faulted (we made this choice because of WWT constraints). Memory loads always blocked, and main memory had an invariant of 1 cycle per 32-bit word fill time. WWT allocates private target pages locally, and distributes target pages residing in the shared address space among the target nodes round-robin. Table 2 lists the cache and directory access latencies that the two different technology levels used.

All experiments assumed transaction queues (of 64 entries each) for updating memory asynchronously upon a replacement of a cache line. Note that even a sequentially consistent system would permit such asynchronous flushes because a program memory access is not associated with the flush. A minimal-cost design employing sequential consistency may eliminate the transaction queue entirely. We did not analyze the performance of such a system.

Each target node was capable of having 64 outstanding transactions, consistent with the SCI standard. Although the target processors used blocking memory loads, writes, coherence and synchronization operations were able to proceed in parallel with both each other and any outstanding memory request.

3.1.2 Interconnection Network

A simple network model assumes a fully connected point-to-point target network and assumes that messages take a constant number of cycles for traversal. This assumption of constant latency provides sufficient lookahead at each node to allow efficient parallel simulation. Reducing the minimum end-to-end network latency reduces the node lookahead, which causes severe increases in simulation time [6].

The constant latency assumption ignores network contention, which can play a pivotal role in evaluating various optimizations. Optimizations that reduce target execution time without a corresponding reduction in communication raise the effective load on the network. Other optimizations that reduce the number of messages lower the offered load. A constant latency model may therefore be either too optimistic or too pessimistic, depending on the simulation parameters.

We used the constant latency model for our experiments. In order to account for network contention, we derived a constant network latency to use for each benchmark, which we obtained with an analytical model. The analytical model that we used [22] requires the network load as a parameter. We estimated this aggregate value from the traffic statistics of previous simulations and their total execution times. The model produces the mean latency of a network traversal, to which we set the constant for the network latency. We iterated this process until the difference between the network latency constant and the value produced by the model for that run converged to within one cycle per message.

To validate this process, we used a detailed, event-driven SCI network simulator (based on the original WWT network simulator [6]) that accurately simulates message buffering, message retransmission, and flow control [5]. The implementation serializes the network simulation at a central node, making simulation performance suffer by roughly a factor of 15.

The target network that we used to derive the validation was an 8×4 mesh of rings. The target network routes requests in increasing dimension order and responses in decreasing order. The internal details of the simulated network correspond closely to those of the SCI transport layer standard. A message's delay through the network includes staging time at the source and target nodes, parsing and wire delay through each intermediate node, and possibly a delay through an agent queue [1], if the message switches dimensions. Table 2 lists the specific times for these delays, for both current and future networks.

Table 3 shows the errors (in terms of target execution time) that the constant latency network model suffers when compared against the detailed network simulation. The two columns of network latencies represent the mean message latency returned by the SCI network simulator and the model, respectively. The error columns show the error in target execution time that we calculated by comparing the constant latency runs against runs that used the SCI network simulator. These network validation runs assumed sequential consistency and MCS locks, and were run using smaller data sets than the other experiments.

3.2 Experiments

QOLB, relaxed memory ordering constraints, and pairwise sharing are all techniques for reducing and tolerating the latencies associated with accesses to shared data. An important question is whether they capture the same optimization opportunities or whether they actually optimize different aspects of the parallel execution. We examined the cross-product of all cases, to catch all cases in which optimizations overlap as well as the cases for which the performance gains are additive.

3.2.1 Synchronization

Our goal in this study was to measure the conditions under which QOLB realizes its performance potentials, and obtain quantitative results for the performance improvements QOLB can provide. We did not exploit the prefetching capabilities of QOLB; we used it only for straightforward synchronization accesses.

We also measured the effect of substituting MCS locks for simple locks, in order to identify circumstances where simple locks outperform MCS or QOLB or both. Our implementation of simple locks uses the **`swap`** primitive.

Enabling the QOLB option in SCI automatically enables pairwise sharing also, as the required mechanisms (primarily direct cache-to-cache write transfer of data) are similar. However, to permit the study of the two options separately we extended the coherence protocol to permit QOLB support without pairwise sharing.

Finally, we performed an experiment to determine the sensitivity of benchmark performance to the QOLB instructions' execution cost.

3.2.2 Memory ordering constraints

Our simulator allows for analysis of memory systems with different ordering constraints on memory operations. We analyzed three designs permitted by the SCI protocol, which cover the spectrum from a conservative, inexpensive design to a very aggressive design approaching the limits imposed by release consistency. Our baseline design, called **`seq`**, achieves sequential consistency by restricting memory reads and writes to a single outstanding operation at any time. A second design (**`r1`**) allows substantial overlap by permitting multiple outstanding memory operations (note that WWT does not permit simulation of multiple outstanding loads). This design blocks on writes until the writing node has become head of the sharing list and the cache line to be modified is resident and writable, and then permits the purge and the actual write to proceed in parallel. This approach is a natural relaxation that can be achieved with a minimum of additional complexity in an SCI-based design. A third design (**`r2`**) relaxes the constraints of write operations further by allowing the processor to insert writes in the transaction queue and continue immediately when appropriate. This scheme can be implemented within the SCI protocol, but adds significantly to the complexity of the design. The WWT assumption of non-blocking loads prevents this scheme from being as aggressive as possible, since it only relaxes writes, some synchronization accesses (specifically DeQOLB), and coherence operations. Even so, this model still has a very high implementation cost, particularly when compared to that for **`r1`**.

A subtle point arises when considering ordering constraints on QOLB instructions. Strictly speaking, QOLB is a memory operation, but it can also be considered a hint to the memory system that helps the hardware propagate data in advance of its use. It does not perform either read or write operations, and can not by itself cause violations of memory ordering constraints. Therefore, QOLB instructions may always proceed asynchronously.

QOLB instructions do, however, provide an opportunity for further weakening memory ordering constraints. Since an entire line is passed between nodes atomically, any data modified before the release of a lock contained in the same cache line can be guaranteed to be consistent everywhere at its release point (**`DeQOLB`**). Thus *no other ordering constraints* are necessary for memory accesses to data protected by a lock contained in the same line. While this observation is correct for any system that transfers only entire cache lines, it is only useful where it is efficient to place data in the same cache line as the associated lock. For this reason, we refer to this phenomenon as *QOLB consistency.* Careful analysis of well-structured code can result in the total elimination of many barriers and other synchronization points. We did not, however, perform such optimizations for our benchmarks.

3.2.3 Pairwise sharing

Our goal with respect to examining pairwise sharing was to determine what application classes can take advantage of the pairwise sharing optimization, as well as the quantitative gain obtained by such applications. Also important is how an application's performance is affected when it can not take advantage of pairwise sharing, when the protocol degrades from a pairwise-sharing list to a conventional list.

The pairwise sharing option defers the breakdown of the sharing list, based on the assumption that two sharing caches will continue to share the data in the near future. In cases where this assumption does not hold, the delayed breakdown of the list may cause the sending of additional messages, adversely affecting program performance.

3.2.4 Cache size

Our experiments were performed assuming two sizes of caches. Most of our measurements were taken assuming a 1MB cache, which is a reasonable size for current technology. For the data sets we used, a cache of this size enabled us to evaluate the performance of the three classes of optimizations, without experiencing finite cache effects. Assuming a large cache, benchmarks with small data sets may tend to overemphasize the benefits of improvements in the efficiency of sharing [21]. Thus, we reran some of the experiments with a smaller cache size, to understand the behavior of the optimizations in the presence of capacity and conflict misses. For this experiment we set the cache size to 8KB. We chose our cache size so that the number of sets is comparable to that in a previous study [15].

3.2.5 Future technology

Simulation results for two types of systems are presented in this paper: target systems using current technology and target systems using future technology (our estimates are for approximately 5 years in the future). This permits us to analyze which of the studied optimizations will increase in importance as technology advances. Specifically, we wished to discover whether system parameters that change due to technological improvement will qualitatively change our results. Table 2 lists the system parameters that we use for the two assumed levels of technology (*current* and *future*). Note that the future processor is effectively five times as fast in terms of instructions executed per unit time.

4 Results and discussion

Table summarizes the main results of this paper. We first analyze each of the three optimizations studied in this paper independently, and then discuss their interactions. We also discuss the effects of technology improvements, and finally the effects of smaller caches.

4.1 Synchronization

Benchmarks using QOLB consistently perform better than when using MCS. The most impressive improvement occurs for Mp3d, the execution time of which QOLB improves by nearly 70%. For this benchmark, most of the sharing occurs in migratory fashion where different nodes in turn update fields of structures. Because these updates are protected with locks, the performance of the benchmark benefits particularly well from QOLB.

Barnes, Ocean and Pthor also benefit from QOLB, by 25%, 12%, and 11%, respectively. The gains are less impressive for two reasons. First, unlike Mp3d, not all fields of the protected structures of Barnes and Pthor fit in a single cache line. The overflow fields do not benefit from the improved hand-off latency offered by QOLB (though careful programming or a very sophisticated compiler could increase the benefit). In addition, some fields in these structures were not placed in the same line as the lock because the applications sometimes access them as unsynchronized data, and false sharing of a locked line can obviously be extremely inefficient. Second, the number of accesses to migratory data is not as frequent in these benchmarks. Both Barnes and Pthor have a number of accesses to mostly-read memory locations, for which QOLB offers no improvement. The majority of shared accesses in Ocean are to memory locations that follow producer/consumer patterns. QOLB is theoretically able to handle such patterns. We did not,

Benchmark	Synch	Current			Future			Ratio
		seq	r1	r2	seq	r1	r2	c/f
Barnes	MCS	1.00	1.06	1.13	1.00	1.08	1.18	3.70
	Pair, MCS	1.00	1.06	1.12	1.00	1.07	1.13	
	QOLB	1.25	1.28	1.30	1.34	1.40	1.41	
	Pair, QOLB	1.24	1.28	1.29	1.31	1.38	1.39	
Mp3d	MCS	1.00	1.12	1.26	1.00	1.14	1.28	3.14
	Pair, MCS	0.95	1.06	1.18	0.94	1.06	1.19	
	QOLB	1.68	1.72	1.79	2.02	2.08	2.14	
	Pair, QOLB	1.58	1.65	1.72	1.85	1.95	2.01	
Ocean	MCS	1.00	1.05	1.31	1.00	1.08	1.42	3.18
	Pair, MCS	1.20	1.26	1.43	1.27	1.39	1.62	
	QOLB	1.12	1.17	1.45	1.14	1.23	1.62	
	Pair, QOLB	1.39	1.46	1.63	1.55	1.69	1.96	
Pthor	MCS	1.00	1.03	1.28	1.00	1.03	1.28	3.23
	Pair, MCS	0.98	1.01	1.24	0.98	1.01	1.24	
	QOLB	1.11	1.15	1.41	1.13	1.18	1.46	
	Pair, QOLB	1.09	1.13	1.38	1.10	1.15	1.42	
Water	MCS	1.00	1.02	1.03	1.00	1.03	1.06	4.68
	Pair, MCS	0.99	1.01	1.02	0.98	1.02	1.04	
	QOLB	1.05	1.05	1.05	1.08	1.09	1.09	
	Pair, QOLB	1.04	1.05	1.05	1.07	1.08	1.08	

Table 4: Results for 1MB caches

These numbers depict application speedups, calculated as the ratio of the execution time of the base run to that of the optimized hardware. The rightmost column shows the speedup of the future technology, seq/MCS simulation to that for current technology with seq/MCS.

however, modify Ocean to take advantage of this ability, since we would have had to rewrite most of the application (invalidating our comparisons).

Table 5 shows the speedups for Mp3d running with **`swap`**, MCS, and QOLB locks, and with each of the three relaxed memory models used in this study. We observe that for this particular benchmark the MCS algorithm consistently improves performance over the simple lock runs by at least 12%. QOLB, however, improves performance a minimum of 56%.

We performed several experiments to measure the program sensitivity to the execution time of the QOLB instructions. The experiments increased the latencies for **`EnQOLB`** and **`DeQOLB`** from 2 and 3 processor cycles respectively to 20 cycles each. Slowdowns for these programs were less than 1%, leading us to conclude that the benefits of QOLB are insensitive to large variations in the latencies of QOLB instructions.

4.2 Relaxed memory ordering constraints

Relaxing memory ordering consistently benefits all benchmarks, particularly in the absence of other optimizations. Mp3d shows the largest improvement using **`r1`**, with a speedup of 12%. All benchmarks but Water show significantly larger improvements using **`r2`**, ranging from 13% for Barnes to 31% for Ocean.

While the gains are modest for **`r1`**, the consistency of the benefit, along with the fact that it is readily achievable with minimal additional hardware or may already be implemented by the processor, make this optimization attractive. The more aggressive implementation of release consistency used for **`r2`** produces substantially better performance. As discussed below, however, its benefit is often reduced for systems that also exploit QOLB.

4.3 Pairwise sharing

Ocean is the only application for which pairwise sharing is visibly successful, speeding up its execution by 20%. The bulk of the sharing in Ocean occurs during phases of stencil computation that involve communication of columns between fixed neighbors. This sharing pattern is common in many scientific applications. Conversely, for applications lacking static pairwise sharing patterns, the execution time remains unchanged (*cf.* Barnes) or even increases slightly (*cf.* Mp3d, 5% slowdown).

Pairwise sharing benefits one of our benchmarks substantially and at worst only marginally degrades the performance of the others. Although our results do not indicate that pairwise sharing is frequently cost-effective, a study testing pairwise sharing on a broader range of applications is needed before we can definitively evaluate its utility.

4.4 Interactions among optimizations

While different benchmarks benefit to varying degrees from the different optimizations, the optimizations produce largely independent improvements. The most surprising result is that most of our benchmarks show more improvement by applying both QOLB and pairwise sharing than would be predicted from applying each individually. Using **`seq`**, the QOLB optimization speeds Ocean up by 12%. Applying pairwise sharing (but not QOLB) results in a speedup of 20%. If the reductions in overhead are fully indepen-

Benchmark	Lock	Consistency model		
		seq	r1	r2
Mp3d	Swap	1.00	1.12	1.18
	MCS	1.14	1.27	1.43
	QOLB	1.91	1.96	2.04

Table 5: Performance of Mp3d varying locks

These numbers depict application speedups; the ratio of the execution time of the base run to that of the optimized hardware.

dent, a speedup of 37% would be expected when both are applied. Applying both obtains a speedup of 39%. The two optimizations clearly have a small positive interaction. We speculate that this interaction is due to the fact that since MCS locking structures are not able to exploit pairwise sharing, eliminating them increases the percentage of shared data accesses that can effectively exploit pairwise sharing.

QOLB and relaxed consistency

QOLB efficiently hides many of the remote latencies for Barnes and Mp3d, speeding them up 25% and 68%. These applications gain little more by having an aggressive relaxed consistency model implemented on top of QOLB. Conversely, Ocean and Pthor benefit less from QOLB, speeding up only 12% and 11%, respectively. Additional performance improvements from relaxing consistency are significant (29% and 27%, running **r2**). Two factors explain this improvement. The first is that—in both Ocean and Pthor—there is a repetitive pattern of read-sharing involving many processors followed by writes to the shared data. The first write operation results in a large number of purges that must complete before the write can proceed. This kind of sharing is not captured by QOLB, but is very well captured by weaker memory models. We attribute the other factor—in Ocean only—to static producer/consumer-type sharing, which could be captured by QOLB, but is not in the implementation of Ocean that we currently use.

QOLB and relaxed memory ordering are effective techniques for tolerating latencies associated with given sharing patterns. QOLB does well in hiding latencies that result from sharing patterns involving serialized exclusive access to shared data. A relaxed memory model effectively hides the latencies associated with multiple readers sharing data, followed by write accesses to that data.

Relaxing memory consistency constraints clearly eliminates some of the same overhead removed by the use of QOLB, particularly when the more aggressive **r2** implementation is considered. Applying **r1** along with QOLB and pairwise sharing reduces the overhead by an amount roughly equal to the sum of the reductions from applying the three optimizations individually. In all cases, applying all three reduces the overhead by at least 75% of the sum of the reductions observed by applying them individually. Applying **r2** in the same way was less successful: The **r2** design clearly removes some of the same overhead eliminated by QOLB.

4.5 Future technology

In a previous study Mellor-Crummey and Scott conclude [19] that "special-purpose synchronization mechanisms such as the [QOLB] instruction are unlikely to outperform our MCS lock by more than 30%." This claim does not hold as shown by the results in Table , where QOLB improves the performance by nearly 70% for Mp3d. Furthermore, this claim becomes ever less tenable as processors become faster, system size increases, and the relative cost of interconnect traffic increases. This is borne out by the results we obtained with our future technology assumptions, where improvements gained by using QOLB for synchronization can be greater than 100%. This performance improvement is likely to increase as processors grow even faster, suggesting that special synchronization hardware support such as QOLB will become more important in future shared-memory multiprocessors. We note, however, that high-performance processor designs that use aggressive prefetching and speculative execution may well be able to capture some of the same latency reduction achieved by QOLB.

Benchmark	Synch.	8K Cache		
		seq	r1	r2
Barnes	MCS	1.00	1.02	1.07
	Pair, MCS	1.00	1.02	1.06
	QOLB	1.30	1.30	1.32
	Pair, QOLB	1.30	1.30	1.32
Mp3d	MCS	1.00	1.11	1.25
	Pair, MCS	0.96	1.05	1.19
	QOLB	1.64	1.68	1.76
	Pair, QOLB	1.60	1.64	1.71
Ocean	MCS	1.00	1.02	1.19
	Pair, MCS	1.02	1.03	1.21
	QOLB	1.08	1.09	1.26
	Pair, QOLB	1.11	1.11	1.30
Pthor	MCS	1.00	1.02	1.18
	Pair, MCS	0.99	1.01	1.16
	QOLB	1.07	1.08	1.23
	Pair, QOLB	1.06	1.07	1.22
Water	MCS	1.00	1.01	1.03
	Pair, MCS	1.00	1.01	1.03
	QOLB	1.04	1.04	1.04
	Pair, QOLB	1.04	1.04	1.04

Table 6: Results for 8K caches

These numbers depict application speedups; the ratio of the execution time of the base run to that of the optimized hardware.

The results for the pairwise sharing optimization are consistent with those for current technology, but substantially larger. The relaxing of memory ordering constraints also shows a distinctive performance increase with our future technology assumptions.

Water was relatively insensitive to any of the applied optimizations, since it is a compute-intensive benchmark that communicates little. Such programs are difficult to speed up by improving interprocessor communication precisely because there is so little communication involved. This is why Water showed the greatest improvement in performance when run with future technology assumptions, which sped up the base run by a factor of 4.68. Speeding up the processor relative to the network latencies, however, not only increases the execution speed dramatically; it also increases the benefits of optimization. With the future technology assumptions, the combined speedup from QOLB and **r2** increased from 5% to 9%. The critical observation is that even computation-bound jobs eventually become communication-bound, either as improvements in the processing speed outstrip gains in the network or as these programs are scaled to larger systems. Thus the optimizations investigated here eventually become effective even for such programs.

4.6 Small caches

Table 6 shows the results for runs assuming current technology and a 8KB cache. Two conflicting behaviors affect the performance improvement due to QOLB. Placing lock and data in the same cache line increases the effectiveness of the cache when compared to data organization imposed by MCS. This is countered by the opportunities for QOLB becoming less frequent, because of the longer execution—due to more cache misses—lessening the potential for performance improvement.

We observed both of these trends in our experiments. The performance improvements attributable to QOLB for Barnes (30% speedup) is larger than those with the larger cache. The performance improvements from using QOLB for the other benchmarks are smaller than those with the larger cache.

Relaxing memory ordering shows the same general trends as in the 1M cache experiments, but to a smaller degree. The only exception occurs with Water, the improvements of which using **r2** are comparable for both cache sizes, primarily because the benefits are so small (3% speedups).

The benchmarks that did not benefit from pairwise sharing with 1M caches incurred smaller performance losses with the smaller cache. Ocean does not benefit significantly from the pairwise sharing optimization with an 8K cache. Its working set exceeds the size of the cache, forcing the eviction of sharing information between iterations.

In general, the conclusions drawn earlier in this section remain valid for the smaller cache size. The only exception is the pairwise sharing optimization, which is quite sensitive to the size of the cache. For caches smaller than an application's working set size, the effectiveness of pairwise sharing remains to be demonstrated.

5 Related work

Gupta *et al.* [15] present a comparative evaluation of some latency reduction and tolerance techniques. They study the performance impact of coherent caches, relaxation of the memory consistency model, non-binding prefetching, and multiple-context processors on shared-memory parallel programs. They report consistent performance improvements for coherent caches, relaxed memory ordering and prefetching. The major difference between our two works is our focus on the interaction between synchronization mechanisms and relaxing consistency. Their methodology also differs from ours in several respects. They used a smaller set of benchmarks than is used in our study. The memory hierarchies were much different—their hierarchy had a 16-entry write buffer and two levels of cache, with a 2KB L1 write-through cache and a 4KB L2 write-back cache, both with 16-byte lines. Another difference is the manner in which speedups for release consistency are measured. For Mp3d, Gupta *et al* report speedups of about 1.5 over sequential consistency. These results exceed the speedups that we observed in our simulations. If we combine the speedups resulting from release consistency with those resulting from simple sequentially consistent transaction queues (we support asynchronous flushes in the base case), we obtain results comparable to those reported by Gupta *et al*.

Aboulenein *et al.* [2] present a detailed analysis study of the QOLB synchronization primitive. A QOLB implementation in the framework of the Scalable Coherent Interface (SCI) is presented. They also present a qualitative performance comparison of QOLB versus MCS and Anderson locks, in which the number of interconnect messages and remote memory accesses are compared. This analysis shows that QOLB outperforms both of these algorithms, both in terms of interconnect messages and memory accesses needed to gain access to a critical section.

Cox and Fowler [9], and Stenström, Brorsson, and Sandberg [24] present studies that propose different solutions for dealing with the problem of migratory sharing patterns. Both studies present adaptive schemes that can be implemented by a hardware cache coherence protocol. QOLB—which predates these studies—captures the same opportunities for optimization, while adding less complexity to the protocol.

6 Conclusions

Although the shared-memory paradigm has many advantages, it incurs overhead—as a result of its fine-grained communication—that can be substantial. Many techniques have been proposed in the literature for reducing this overhead. Using the Wisconsin Wind Tunnel as a simulation engine, and a target system based on the Scalable Coherent Interface standard, we have analyzed the performance effects of three such overhead-reducing techniques.

We have shown that the synchronization primitive QOLB provides a substantial benefit over a broad range of applications. The only benchmarks failing to show significant improvement from QOLB are those that are spend most of their time computing or waiting at barriers. In the results reported, QOLB was used exclusively to implement synchronization more efficiently than was possible using MCS locks. Performance was improved by more than 10% for four of the five benchmarks, and Mp3d experienced a speedup of 1.68. Further performance improvements are to be expected with the aggressive use of QOLB as a synchronizing prefetch and possibly with the exploitation of QOLB consistency.

The SCI-specific feature of pairwise sharing showed mixed results. For one benchmark—Ocean—pairwise sharing improved performance by 20%. For all others, performance suffered slightly. While an analysis of the sharing patterns can predict the effectiveness of pairwise sharing, it benefits only a limited range of applications, and even these applications may fail to see a large benefit from the optimization if the application is seriously constrained by capacity misses in the cache. However, the interaction with QOLB, and the fact that their hardware implementations under SCI are similar, suggests that pairwise sharing may be a good design choice when cost/performance is considered.

The SCI protocol exhibits its most severe delays when a write is initiated on a cache line that is widely shared. It is a small change to relax consistency by allowing the processor to proceed in parallel with the purging of cached copies. Such a relaxation of the memory model consistently improves performance, though not dramatically.

QOLB and relaxed consistency exploit mostly different opportunities for tolerating latency. For **r1**, we observed that performance improvements from pairwise sharing or QOLB are nearly additive to those of relaxed consistency. The aggressive implementation of release consistency (**r2**) is sometimes less effective in the presence of QOLB, indicating some overlap in the overhead that they reduce. The greatly increased complexity of **r2** must be weighed against its benefit, particularly if QOLB is also implemented.

QOLB and pairwise sharing benefits show a surprisingly positive correlation. In Ocean, the only benchmark benefitting substantially from pairwise sharing, the combined benefit is greater than that predicted from the two separate improvements. In both the cases where pairwise sharing improved performance and those where it degraded it, the combined effect of the QOLB and pairwise sharing options is generally better than expected from their individual contributions. Among our benchmarks, only Water failed to show this effect, and even there the effects were orthogonal within the margin of error of the study.

The results of this work are more dramatic when the system parameters are adjusted to reflect technology trends. Assuming faster processors that are more heavily penalized by network delays, the benefits from relaxing consistency increase. Pairwise sharing results are still mixed, but the benefit to Ocean is more impressive. The benefits of QOLB also increase substantially, suggesting that this type of primitive will become an important feature of future systems.

While our results are specific to the Scalable Coherent Interface, the lessons are much more broadly applicable. First, we have established that special support for hardware synchronization, such as QOLB, can have a profound effect on the performance of current systems. More importantly, we have shown that future trends are likely to increase that benefit. Second, we have demonstrated that relaxing memory ordering constraints consistently improves performance, and in combination with QOLB, can show dramatic improvement.

Acknowledgments

We would like to thank both the National Science Foundation, and Dave James and Don North of Apple Computer, for their generous support. We thank Steve Scott for use of his analytical SCI network model; Sarita Adve, Satish Chandra, Babak Falsafi, Steve Reinhardt, Michael Scott, and David Wood for their technical assistance and penetrating insights; Glen Ecklund for his critical assistance with the CM-5. Finally, we thank Guri Sohi and Dave Gustavson for their helpful comments on drafts of this paper.

References

[1] Scalable Coherent Interface (SCI). ANSI/IEEE Standard 1596-1992, August 1993.

[2] Nagi M. Aboulenein, James R. Goodman, Stein Gjessing, and Philip J. Woest. Hardware Support for Synchronization in the Scalable Coherent Interface (SCI). In *Proc. of the 8th International Parallel Processing Symposium*, pp. 141–150, April 1994.

[3] Sarita V. Adve and Mark D. Hill. Weak Ordering — A New Definition. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pp. 2–14, May 1990.

[4] Thomas E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. In *Proc. of the 1989 International Conference on Parallel Processing*, vol. II Software, pp. 170–174, Aug 1989.

[5] Douglas C. Burger and James R. Goodman. Simulation of the SCI Transport Layer on the Wisconsin Wind Tunnel. Technical Report 1265, Computer Sciences Department, University of Wisconsin-Madison, April 1995.

[6] Douglas C. Burger and David A. Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *Proc. of the 9th International Parallel Processing Symposium*, April 1995.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th Symposium on Operating Systems Principles*, pp. 152–164, October 1991.

[8] Convex Computer Corporation, Richardson, Texas. *SPP1000 Systems Overview*, 1994.

[9] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pp. 98–108, May 1993.

[10] Cypress Semiconductor, San Jose, California. *CY7C601 SPARC RISC User's Guide*, second edition, 1990.

[11] Frederick Dahlgren, Michel Dubois, and Per Stenström. Combined Performance Gains of Simple Cache Protocol Extensions. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pp. 187–197, April 1994.

[12] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, 1992.

[13] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26, June 1990.

[14] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proc. of the 15th Annual International Symposium on Computer Architecture*, pp. 422–431, May 1988.

[15] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proc. of the 18th Annual International Symposium on Computer Architecture*, pp. 254–263, May 1991.

[16] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[17] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sep 1979.

[18] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[19] John M. Mellor-Crummey and Michael L. Scott. Synchronization Without Contention. In *Proc. of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 269–278, April 1991.

[20] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proc. of the 1993 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pp. 48–60, May 1993.

[21] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pp. 14–25, Jun 1993.

[22] Steven L. Scott, James R. Goodman, and Mary K. Vernon. Performance of the SCI Ring. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pp. 403–414, May 1992.

[23] J. P. Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[24] Per Stenström, Mats Brorsson, and Lars Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proc. of the 20th Annual International Symposium on Computer Architecture*, pp. 109–118, May 1993.

[25] Philip J. Woest and James R. Goodman. An Analysis of Synchronization Mechanisms in Shared-Memory Multiprocessors. In *Proc. of the International Symposium on Shared Memory Multiprocessing*, pp. 152–165, April 1991.