Efficient Synchronization: Let Them Eat QOLB[1]

Alain Kägi, Doug Burger, and James R. Goodman

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706 USA
galileo@cs.wisc.edu - http://www.cs.wisc.edu/~galileo

Abstract

Efficient synchronization primitives are essential for achieving high performance in fine-grain, shared-memory parallel programs. One function of synchronization primitives is to enable exclusive access to shared data and critical sections of code. This paper makes three contributions. (1) We enumerate the five sources of overhead that locking synchronization primitives can incur. (2) We describe four mechanisms (local spinning, queue-based locking, collocation, and synchronized prefetch) that reduce these synchronization overheads. (3) With detailed simulations, we show the extent to which these four mechanisms can improve the performance of shared-memory programs. We evaluate the space of these mechanisms using seventeen synchronization constructs, which are formed from six base types of locks (TEST&SET, TEST&TEST&SET, MCS, LH, M, and QOLB). We show that large performance gains (speedups of more than 1.5 for three of five benchmarks) can be achieved if at least three optimizing mechanisms are used simultaneously. We find that QOLB, which incorporates all four mechanisms, outperforms all other primitives (including reactive synchronization) in all cases. Finally, we demonstrate the superior performance of a low-cost implementation of QOLB, which runs on an unmodified cluster of commodity workstations.

1 Introduction

Shared-memory multiprocessors are rapidly becoming the machines of choice for solving large, fine-grained scientific programs. Multiple factors support this trend. The advent of affordable desktop symmetric multiprocessors (SMPs) will increase the application base. The successful development of shared-memory multiprocessing standards [43] reduce the time to market by decreasing design time and by letting manufacturers use commodity parts. Both the Convex Exemplar [7] and the Sequent STiNG [27] relied on these standards. The emergence of low-cost, fine-grain software implementations of shared-memory, such as SHASTA [38] or TO [35] further reduce the cost of supporting the shared-memory model. Finally, successful research prototypes such as the Stanford DASH [25] have shown that this class of machines can obtain excellent speedups for a wide range of programs that use fine-grained communication.

Traditional message-passing programming models force the programmer to embed implicit synchronization with each communication of data. Such a requirement restricts the parallelization strategy—dynamic task distribution becomes extremely difficult, for example. The shared-memory programming model, conversely, uses cache coherence protocols to keep shared data consistent. The programmer judiciously employs explicit synchronization to provide mutual exclusion for data and code, as well as synchronizing processors between phases of computation.

The two major classes of explicit synchronization operations in shared-memory multiprocessors are barriers and locks. Although barriers are important to efficient shared-memory programs, they are well-understood, and many efficient implementations have been proposed and/or built [15, 20, 23, 32, 44]. In this study, we focus on providing more efficient mutual exclusion through better locks.

Locks provide individual processors with exclusive access to shared data and a critical section of code. This exclusive access is particularly well-suited to the fine-grained nature of many shared-memory parallel programs. Fine-grained programs ideally associate as little data or code as possible with a critical section, minimizing serialized processing, thus maximizing available parallelism. Since access to critical sections is by definition serialized among processors, large overheads when accessing a contested critical section degrade both parallel performance and potential scalability. To maximize both the performance of fine-grain parallel applications that use locking, and the potential to scale to larger numbers of processors, we must minimize the delays associated with the transfer of exclusively accessed resources.

The act of transferring control of a critical section is a complex one, that may involve multiple remote transactions. Complex protocols have been proposed that perform this transfer efficiently, allowing reasonable performance when there is high contention for a lock. The complexity of these protocols causes unnecessary delays when accessing a lock that is not held. Conversely, simple locking schemes that can access a free lock quickly may perform poorly in the presence of contention. This fundamental trade-off has resulted in proposals of numerous primitives in the literature [3, 13, 16, 26, 28, 30, 37].

This paper contains a detailed and thorough evaluation of a range of locking primitives. To understand where the opportunities for optimization lie, we first decompose the time associated with a

1. Pronounced "Colby."

This work is supported in part by NSF Grant CCR-9509589 and by the Intel Research Council.

complete locking period into three phases: *Transfer*, *Load/Compute*, and *Release*. Together, these phases form a *synchronization period*, which determines the global throughput of synchronization operations and thus determines scalability for codes that rely heavily on locks. We then describe four mechanisms that locks may incorporate to reduce the time spent in the three phases: *local spinning*, *queue-based locking*, *collocation* (of a lock and data within the same cache line), and *synchronous prefetch.*

Using detailed simulation with both microbenchmarks and real applications (drawn from the SPLASH and SPLASH-2 suites), we measure the performance of six base primitives: TEST&SET, TEST&TEST&SET [37], LH locks [28], M locks [28], MCS locks [30], and QOLB [13]. We extend these primitives with the mechanisms listed above, plus exponential backoff and prefetches inserted automatically by a compiler. We also measure the performance of reactive synchronization schemes. In all, we study a total of seventeen primitive/mechanism combinations. We find that QOLB, which can incorporate all of the mechanisms listed above, outperforms all other locks in all cases (including reactive synchronization). We also see that very efficient locking can *double* the speedup of real applications (for one of the five benchmarks that we measured). Although QOLB outperforms the other primitives, it requires mechanisms that the others do not (which usually implies hardware support). We discuss exactly what support QOLB requires, and show that much of the necessary support already exists in current systems. Finally, we present performance results of an all-software implementation of QOLB running on an unmodified cluster of commodity workstations, and we show that this low-cost implementation still outperforms the alternatives.

In Section 2, we explain our decomposition of a synchronization period in greater detail. In Section 3, we show how the four optimizing mechanisms that we identified can reduce different parts of the synchronization period. In Section 4, we explain the primitives that we study in detail, and discuss how each of them uses a different set of the four mechanisms. In Section 5, we describe our experimental methodology. In Section 6, we present and discuss our performance results from this experimental space. In Section 7, we discuss the cost of hardware-supported synchronization. Finally, in Section 8 we provide a summary of our main results and conclude.

2 Overhead of mutual exclusion

From the perspective of an individual processor, the time associated with an access to a critical section consists of the time from which the processor first requests access to the corresponding lock, to the time at which the processor completes the release on that lock. This time period does not directly correlate with global performance, however. Multiple processors contending for entry to the same critical section may overlap the time from the issue of their requests to the first release of the lock. A good analogy to this distinction is the difference between the latency of an individual request to a memory system, and the throughput achievable by pipelined accesses to that same memory system.

To determine how these critical section accesses limit global performance and ultimately scalability, we define the notion of a *synchronization period.* The synchronization period is the length of time between completion of two successive synchronization operations (e.g., two successive releases) on the same variable. The successive synchronization operations may occur on different processors. This synchronization period is the service time that the processor incurs once the previous processor releases the lock. Since access to this critical section is by definition serialized, the synchronization period will place an upper bound on possible performance (codes that do not access critical sections heavily will see upper bounds on performance from other sources, of course).

We depict our breakdown of a synchronization period in Figure 1. The figure shows events to synchronization variable X. The first event depicted is the completion of the release of lock X by processor A. Several processors are contending to gain access to X. We assume that processor B wins the ensuing arbitration. When the lock acquire completes, processor B enters the critical section. Upon finishing the work in the critical section processor B prepares to release X, and eventually completes this operation. Our breakdown of a synchronization period consists of three phases:

- *Transfer*: the time at which processor A completes its release of the lock to the time processor B completes its acquire. At the point that the release completes; the releasing processor has atomically written the "unlocked" value to the lock. The contending nodes may then issue or re-issue requests (depending on the locking primitive) to obtain the lock. A period of arbitration may ensue. Once the next recipient of the lock is determined, the lock must be sent to that node.

- *Load/compute*: the time at which processor B completes its lock acquire to the time processor B issues its lock release. Once a processor obtains the lock, it enters the critical section. The processor will most likely have to read some locked data, perform some computation, and write some locked data. Accessing the data to read and write will likely incur some remote accesses.

- *Release*: which is the time from processor B issuing the lock release to the completion of the lock release. When the processor issues a release operation for the lock, remote accesses may be necessary before that operation may complete. Other processors may have removed the lock from the releasing processor's cache, for example, or the releasing processor may have to re-obtain write permission for the lock's cache line. Some aggressive memory models [1, 12] may allow some overlap between the *Load/compute* and *Release* phases.

In addition to illustrating this decomposition in Figure 1, we also list the components of each phase. The components marked with an asterisk are the only ones that are fundamental, which would be part of a truly minimal synchronization period. The components marked with a "+" are overheads that cannot be eliminated, but whose latencies may be partially or entirely hidden. Unmarked components are ripe for elimination through optimization.

3 Synchronization mechanisms

We have isolated what we believe to be a fundamental set of four mechanisms that synchronization primitives may incorporate. In Table 1 we show the overheads (from Figure 1) reduced by each

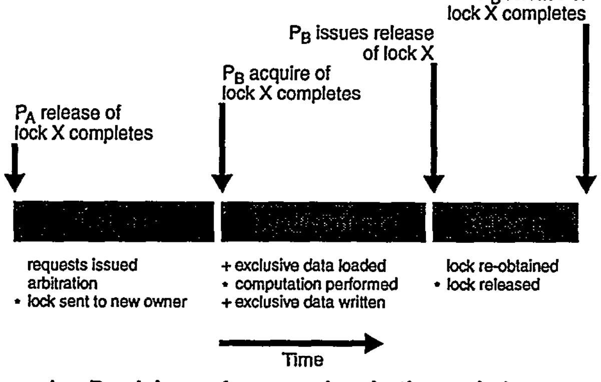

Figure 1 Breakdown of one synchronization period.

	Phase of the synchronization period				
	Transfer		Load/compute		Release
Synchronization mechanisms	Arbitration	Lock transfer	Data read	Data write	Re-obtain lock
Local spinning					
Queue-based locking	✓	✓			✓
Collocation			✓	✓	
Synchronous prefetch		✓	maybe		

Table 1 *How synchronization mechanisms reduce overhead.*

of these mechanisms. The definitions and explanations of each mechanism are as follows:

• *Local spinning*: allows a requesting node to spin on a local copy of the lock. Although local spinning does not directly reduce overheads on the critical path of the synchronization period, it does greatly reduce the load on the network, particularly for longer critical sections. When the lock is released, the coherence mechanism will invalidate all local copies (since the releasing node needed an exclusive copy to modify that line), and when they next access the line, one requester will see that the lock has been freed and will acquire the lock.

• *Queue-based locking*: eliminates arbitration overhead and reduces lock transfer time, both in the *Transfer* phase. This mechanism reduces synchronization overhead in the following ways: (i) creates a queue of waiting requesters, thus performing arbitration when the requests are received and not when the current holder releases the lock; (ii) reduces lock transfer time by restricting communication to be between the releasing node and the acquiring node only (although the number of remote accesses required to perform this transfer will vary among different primitives); (iii) eliminates the overhead of re-obtaining the lock in the *Release* phase, since no other nodes access the lock directly until the holder releases the lock.

• *Collocation*:[1] lets protected data be transferred with the transfer of the lock itself. Since the data arrive with the lock, collocation eliminates read and write overheads in the *Load/Store* phase. The implementations we study in this paper achieve collocation by coupling a lock and critical data together in the same transfer block (a cache line). If the critical data are larger than one cache line, collocation will only partly reduce the read and write access overheads. If the critical data are determined dynamically, effective collocation is difficult.

• *Synchronous prefetch*: allows a processor to issue a request for a particular lock in advance of its critical section. The memory system will effect the transfer of the lock from the holder to the prefetching requester only when the holder releases the lock. Thus this mechanism will not impede the current holder's progress in the critical section. If a node prefetches the lock and the holder releases it before the requester reaches its critical section, the requester may be able to hide the lock transfer latency completely.

1. col.lo.ca.tion (n) \.kal-*-'ka-sh*n\: the act or result of placing or arranging together; specif: a noticeable arrangement or conjoining of linguistic elements (as words) [45] (*words* in this context are 4-byte quantities of data).

4 Synchronization primitives

The six base primitives we discuss in this paper are Test&Set (abbreviated TS), Test&Test&Set (abbreviated TTS), MCS locks, LH locks, M locks, and Qolb. Table 2 shows which primitives incorporate which of the mechanisms described in Section 3. In Table 3, we show the minimum number of remote messages required for acquiring a lock. The counts correspond to messages on the critical path only. Each pair of numbers shown represents the number of messages required for a DASH-like [25] and an SCI-like [43] protocol, respectively. In cases where the lock is not held (columns one and two), the number of transactions is from issue to completion of the lock acquire. If another node holds the lock, the number of remote transactions shown is the number from issue of the release by the lock holder to the completion of the acquire by the requester. In the rest of this section, we define each base primitive and describe each in terms of the mechanisms that it incorporates, as shown in Table 2.

4.1 Test&Set

Test&Set (TS) was the sole synchronization primitive available on numerous early systems (such as the IBM 360 series [17]). TS performs an atomic read-modify-write on a memory location. It reads the value contained therein, and unconditionally sets the value to be non-zero. TS returns the value that was obtained from the read. It may be implemented with an atomic swap of as little as one bit.

We see in Table 3 that the TS primitive is efficient when a lock is not held; the primitive can immediately load the lock into the processor's cache and lock it. TS is less efficient when there is contention for a lock, since the lock's line is shifted from requester to requester in "exclusive" state. When the holder wishes to release the lock, it must re-obtain the lock from the requester that has moved the line into its cache. Concurrently, all requesters continue to send requests for writable copies of the lock. Although this scheme technically guarantees that some processor makes forward progress, it does not guarantee fairness, nor does it prevent starvation. Worse, it generates continuous remote transactions from the requesters (if there are more than one), even while the lock is being held. We see from Table 2 that the only optimization (of those in

	Synchronization mechanism			
Synchronization primitive	Local spinning	Queue-based locking	Collocation	Synchronous prefetch
TS	no	no	optional	no
TTS	yes	no	optional	no
MCS, LH, M	yes	yes	partial	no
Qolb	yes	yes	optional	yes

Table 2 Synchronization primitives. For each synchronization primitive, this table shows which synchronization mechanisms it incorporates. We deemed collocation to be optional, since the programmer may choose not to exercise it.

	Minimal number of remote messages			
Synchronization primitive	Lock idle in memory	Unlocked, cached elsewhere	Locked, single contestant	Locked, N contestants
TS	2, 2	3, 6	5, 11	5, 11
TTS	4, 2	6, 6	8, 11	8, 9+2×N
MCS	2, 2	3, 6	7, 15	5, 9
LH	2, 2	9, 10	5, 11	5, 11
M	2, 2	3, 6	5, 11	5, 11
QOLB	2, 2	3, 4	1, 1	1, 1

Table 3 Number of remote transfers for acquire. The numbers in the table represent the minimal number of messages needed to acquire a lock. The counts correspond to messages on the critical path only. We show numbers for several initial lock states and two cache-coherence protocols. Each number on the left assumes a DASH-like protocol [25], and each number on the right assumes an SCI-like protocol [43]. We assume that the acquiring node, the releasing node (if applicable), and the directory node are all different nodes. In cases where the lock is not held (columns one and two), the number of transactions is from issue to completion of the lock acquire. If another node holds the lock, the number of remote transactions is from issue of the release by the lock holder to the completion of the acquire by the requester.

the table) that TS may implement is collocation. Collocation may be effective if requesters rarely attempt to obtain the lock while held. When a request for a held lock occurs, however, the requester and holder will ping-pong the lock (and collocated data) between their caches, as the holder accesses the data and the requester spins on the lock. The ping-ponging of the block will stall the holder, increasing the length of its critical section and thus increasing the global synchronization period.

A policy often applied to TS is exponential backoff, in which after a failure to obtain the lock a requester waits for successively longer periods of time before issuing another request for a lock [3]. We implemented a backoff scheme closely following the guidelines that appear in the original article: when an attempt to obtain a lock is unsuccessful, the requestor waits for an amount of time randomly selected from a uniform distribution; the algorithm doubles the mean of the distribution after each failed attempt up to a maximum. At the start of a fresh synchronization period the initial mean corresponds to half of the mean used in the previous period. The maximum mean is set to 16K cycles, which is roughly the time required to service a simple write miss (i.e., three network round trips or approximately 600 cycles) times the number of nodes in the system. We initialize the mean to one cycle, which corresponds also to the minimum mean.

4.2 TEST&TEST&SET

Rudolph and Segall first proposed an extension to TS that performs a read of the lock before attempting the actual TS operation [37]. They called this primitive TEST&TEST&SET (TTS). This primitive enables waiting requesters to spin on shared, read-only copies of the lock (local spinning), waiting for the holder to release the lock. When the holder issues the release, the read-only copies are invalidated, the holder obtains a writable copy of the lock, and then releases it. The requesters load readable copies into their caches, and finding the lock released, attempt the TS. One of the requesters will succeed in obtaining a writable copy of the lock and locking it.

Although TTS employs local spinning to reduce interconnect traffic while the lock is held, the time needed to acquire the lock is longer than TS (see Table 3), due to the requesters' initial requests for read-only copies (instead of an exclusive copy, as with TS). The contention when the lock is freed can be substantial, as all requesters attempt to acquire the lock at that point, and then all attempt to upgrade the lock to a writable state. Exponential backoff may therefore improve TTS as well as TS. Collocation with TTS may work better than with TS, since the lock holder can still read data allocated in the lock's cache line, as it is shared with the requesters. TTS collocation is not ideal, however, since the holder will ping-pong the cache line with requesters whenever it writes to the collocated data.

4.3 MCS locks

Several researchers have independently proposed locking primitives that incorporate both local spinning and queue-based locking in software [2, 29, 16]. One of them is the locking primitive called MCS, developed by Mellor-Crummey and Scott [29]. The MCS scheme inserts requesters for a held lock into a software queue at the time of the request, using atomic operations such as SWAP and COMPARE&SWAP to update the list correctly. With queue-based locking, arbitration for the eventual recipient of the lock is therefore performed in advance, first-come, first-serve. Arbitration for TS and TTS, conversely, occurs at the time of lock release, increasing the synchronization period.

The price of maintaining the requester queue in software is larger overhead, especially under contentionless conditions. When a lock is released, however, communication occurs only between the releaser and the requester at the head of the queue. Network traffic is thus reduced to a constant number of network traversals per synchronization access, while the other requesters in the queue continue to spin locally.

Since each requester is spinning on a different address, these software queue-based algorithms cannot easily benefit from collocation. Partial collocation can be achieved by placing protected data along with the data structure that tracks the queue insertion point. If there is little contention, partial collocation may be effective. A more sophisticated approach could better exploit collocation by placing data either with the insertion pointer when there is no contention, or with the appropriate queue element when contention exists. However, this approach requires copying of data which, done carelessly, may sacrifice their integrity (e.g., in the context of recursive data structures). We did not investigate this approach. These algorithms are also unable to prefetch data without significant changes that greatly add to their complexity.

4.4 LH and M locks

Magnusson, Landin, and Hagersten proposed two software queue-based locking primitives, LH and M [28] (Craig independently developed a lock identical to LH [8]). They claimed that their primitives would require one fewer remote access to transfer a lock than does MCS, enabling their schemes to outperform MCS when lock contention exists. The LH lock achieves this behavior at the expense of increased latency to acquire an uncontested lock. The M lock achieves the more efficient lock transfer without increased uncontested lock access latency, at the expense of significant additional complexity in the lock algorithm. We implemented both locks according to the description in their paper, which presents the actual algorithms in detail [28].

4.5 Reactive synchronization

In 1994, Lim and Agarwal proposed "reactive synchronization" schemes [26], which dynamically switch among software locks that perform well under various levels of contention. For instance, it may combine TS for low-contention phases of execution with MCS for periods of high-contention. Reactive synchronization attempts to achieve both low latency lock access and efficient transfer at low cost (e.g., using only all-software primitives).

We implemented reactive synchronization, closely following the guidelines in the paper [26]. For low-contention phases, we used TS with exponential backoff. For high-contention phases, we used MCS (our results show that MCS is the best-performing software lock under high contention, of the locks that we measured). Our implementation switched to MCS after five consecutive lock acquisitions experienced higher levels of contention than a fixed threshold (a mean delay of 32 clock cycles). We switched from MCS to the low-contention lock when the queue was empty upon lock release five consecutive times.

4.6 QOLB

Goodman, Vernon, and Woest proposed the Queue-On-Lock-Bit primitive (QOLB—originally called QOSB) [13], which was the first proposal for a distributed, queue-based locking scheme. QOLB maintains a hardware queue of waiting processors, in which pointers to adjacent queue entries are held in the cache line. Waiting processors spin locally on a "shadow" copy of the lock address, preventing unnecessary network traffic or interference with the lock holder. Because lock requesters spin on the same address as that of the lock, without evicting or downgrading the lock holder's copy, effective collocation is possible (unlike the other primitives that we have discussed). When the holder releases its lock, the lock is sent directly to the requester at the head of the queue, incurring a total of one network crossing to transfer the lock (see Table 3).

In addition to enabling local spinning, collocation, and efficient handoffs through queueing, QOLB is a non-blocking primitive. This characteristic permits a processor to use QOLB for performing synchronous prefetching, allowing the processor to overlap data and lock access times with other useful work. If the prefetch is issued sufficiently far in advance, it is possible for the requester to see *no overhead* associated with the critical section entry, either for accessing the lock or the data. Figure 2 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 cache line and sends a message that inserts the requester into the hardware requester queue. This early request 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. In this example, we assume that the critical section data can fit in 63 bytes. This will not always be the case, of course. Also, QOLB is fair in general, except in the unusual cases when a processor's shadow copy of the lock is replaced from its cache, forcing the processor to rejoin the queue at its end.

5 Experimental methodology

We measured the performance of the six synchronization primitives discussed in Section 4, varying mechanisms from Table 2 when possible, except that we did not simulate collocation in conjunction with the LH and M locks (we will show later that MCS generally performs better than LH and M, which are not inherently more amenable to collocation than MCS). We also measured the performance of reactive synchronization (also without collocation since reactive synchronization is not inherently amenable to collocation). Our seven main locking schemes (and their corresponding abbreviations) are thus as follows: TEST&SET (TS), TEST&SET&SET (TTS), MCS locks, LH locks, M locks, reactive synchronization (R), and QOLB. We used the following abbreviations for optional mechanisms or policies: collocation (+C), hand-inserted synchronous prefetch (+P), compiler-generated synchronous prefetch (+CP), and exponential backoff (+E).

```
struct _locked data {
  char lock;
  char data[63];               /* 64-byte cache line */
};

void
critical_section(struct _locked data *ptr) {
  /* Prefetch lock & data (assumes proper alignment) */
  EnQOLB(&ptr->lock);
  /* Various computation here */
  ...
  while (!EnQOLB(&ptr->lock)) ;                /* Spin */
  /* Critical section here */
  ...
  DeQOLB(&ptr->lock);                  /* Release lock */
}
```

Figure 2 QOLB **code example.**

5.1 Simulation environment

Our simulation platform was the Wisconsin Wind Tunnel (WWT) [33], which uses a 32-processor Thinking Machines CM-5 [23] as its host machine. WWT executes SPARC binaries in native mode on the CM-5, only trapping into the simulator upon a cache miss. WWT assumes fixed execution time for the instructions (the actual values correspond to the instruction delays listed in the CY701 SPARC user's guide [9]). WWT makes some assumptions about the target system to simplify simulation—it assumes both a perfect instruction cache, and that stack accesses always hit in the data cache.

The default WWT network model assumes a fully connected point-to-point target network, in which messages take a constant number of cycles for a one-way network traversal. A large enough constant latency provides sufficient lookahead for efficient parallel simulation, as nodes stop and synchronize only once every C cycles, where C is the constant network latency. Using a small C (or variable-length messages) reduces the node lookahead, which causes severe increases in simulation time [6].

Although we model contention at the node interfaces, memory, and memory directories, using a constant network latency ignores contention in the network itself. To account for network contention, we used an analytical model [41] (which takes the network load as a parameter) to derive a different constant network latency for each benchmark. We estimated this aggregate network load from the traffic statistics of previous simulations and their total execution times. Since the network latency affects execution time and therefore aggregate load, we iterated this estimation until the difference between the network latency constant and the value produced by the model converged to within one cycle (the final latencies for the benchmarks ranged from 85 to 91 cycles). To validate this methodology, we simulated several points for each benchmark using the WWT extended with a detailed, event-driven SCI network simulator. Our network simulator accurately simulates message buffering, message retransmission, and flow control [5]. The target network that we used to derive the validation is an 8×4 mesh of rings that routes requests in increasing dimension order (x, y) and responses in decreasing order (y, x). The internal details of the simulated network correspond closely to those of the SCI transport layer standard [43]. The mean difference between the execution time of simulations using the constant network model and simulations using the detailed network simulator was 2%. The difference was always under 5% [18].

Using a global mean to model contention tends to underestimate execution time, since traffic often occurs in bursts that add more queueing delay than if the same traffic was evenly distributed over time. With our validation, we have bounded this discrepancy. Even so, since our more aggressive synchronization primitives (MCS, QOLB) generate less traffic than do the alternatives, accurately modeling contention in the network would only serve to increase the reported performance gap between the lower- and higher-performing primitives. Our results are therefore conservative.

5.2 Target systems

The target systems that we simulate are all 32-processor, cache-coherent shared-memory systems that use the Scalable Coherent Interface (SCI) [43] as their base cache-coherence protocol. SCI is a particularly appropriate choice for our base platform, since two of the newest shared-memory multiprocessors on the market implement cache-coherent SCI (the Convex EXEMPLAR [7] and the Sequent STING [27]), and numerous other vendors are exploring SCI as an option. Each node in our CC-NUMA target system is workstation-like, containing a processor, a 1-Mbyte four-way set-associative cache memory with 64-byte lines, a 64-entry transaction queue, a network interface, and some fraction of the distributed, globally-shared memory with the associated directory entries. The transaction queue is similar to a functionally extended write buffer. It supports the following asynchronous operations: writes, prefetches, coherence operations, and cache line flushes caused by replacement (rollouts). A complete description of the system parameters and their associated timings appears elsewhere [18]. WWT allocates private target pages locally, and distributes shared target pages to the target nodes round-robin. Our simulated memory system supports release consistency [12].

5.3 Microbenchmark experiment description

We repeat the method used by both Anderson [3] and Lim and Agarwal [26] to measure raw critical section throughput. We constructed a microbenchmark that accesses a critical section in a loop repeatedly (the benchmark accesses the critical section a total of 3,200 times; these accesses are distributed evenly among the processors). Once in the critical section, a processor waits 800 cycles before releasing the lock (this stall simulates access to, and computation of, protected data). After release, the releasing processor waits for a random amount of time selected from a uniform distribution. The mean of the distribution is five times the critical section delay (4,000 cycles). As the number of nodes is increased, the contention for the lock increases, and eventually the reduction in execution time is stopped (and in some cases reversed) by the increasing lock contention.

For this experiment we assumed a fixed network latency between any two nodes of 100 cycles.

5.4 Macrobenchmark experiment descriptions

The benchmark applications that we used for our experiments are Barnes, Mp3d, Ocean, Pthor, and Raytrace, drawn from the SPLASH and SPLASH-2 suites [42, 46]. Descriptions of these benchmarks appear in the original articles. We compiled all benchmarks using GCC version 2.7.2 with the option -O3. We padded data in each benchmark, where necessary, to eliminate false sharing [14]. We modified Ocean both by translating it to C and by skewing its array storage (slightly increasing the size of the working arrays into arrays of prime size, from 128 to 131 elements in each dimension). We used the locking version of Mp3d for all experiments. Pthor assigns a descriptor to each element of the digital circuit being simulated. Only a few fields of this descriptor are frequently modified in the course of the simulation. To take advantage of automatic replication of read-only data and reduce cache misses, we collocated the frequently modified Pthor fields in a single cache line.

BENCHMARK	TYPE OF SIMULATION	INPUT	SYNCH. PERIOD
Barnes	Barnes-Hut N-body	2,048 bodies, 11 iter.	1,840
Mp3d	Hypersonic flow	24,000 mols, 25 iter.	44
Ocean	Hydrodynamic	98x98, 2 days	17,469
Pthor	Digital circuit	RISC, 1,000 timesteps	7,633
Raytrace	3-D rendering	TEAPOT	490

Table 4 Macrobenchmarks.

We list the problems that the benchmarks solve and the inputs that we used in Table 4. The fourth column of Table 4 lists the period of critical section entry for each benchmark, computed by dividing the benchmark execution time (discounting initialization) by the total number of critical section entries (across all 32 processors). We computed this statistic from the sequentially consistent run of QOLB with all mechanisms enabled. The frequency at which locks are obtained is an important metric, since improving the synchronization primitive will have little benefit for an application that uses locks infrequently.

For these macrobenchmarks, we varied the memory model as well as the synchronization primitive. By using two memory models (sequential consistency and aggressive release consistency), we show that the performance gained by improving the synchronization primitive cannot also be gained solely by making the memory model more aggressive. The memory models that we simulated are two different implementations of release consistency: sequential consistency (denoted SEQ), and an aggressive implementation that attempts to minimize the number of times that the processor is stalled by memory operations (denoted REL). For the latter memory model, we labeled all memory accesses as aggressively as possible according to the structure proposed by Gharachorloo and others [11, 12], and inserted the appropriate memory fences to achieve release consistency on our simulated hardware platform. Although our system assumes blocking loads, we implement a merging write buffer of up to 64 non-blocking stores, which allows multiple stores to be combined and loads to be serviced by stores. This large buffer permits very aggressive relaxation of the consistency model for stores.

5.5 Prefetching compiler algorithm

We used an enhanced version of GCC that automatically inserts prefetch operations, developed locally by Aboulenein. This compiler takes a critical section and the address of the associated lock variable, and automatically inserts the ENQOLB and DEQOLB instructions for the lock. More importantly, the compiler attempts to move an ENQOLB instruction to a prespecified distance above the entry point to the critical section, thus performing a synchronous prefetch.

The compiler uses two methods for trying to insert the prefetching ENQOLB instructions. It first attempts to move the prefetch operation into a basic block that *dominates* [24] the basic block containing the entry point of the critical section. If the compiler is unable to locate a basic block that dominates the critical section entry point, the compiler resorts to a technique similar to trace scheduling [10], which inserts ENQOLB operations in non-dominating basic blocks. To ensure correctness, the compiler must also insert DEQOLB operations along all possible paths that do not include the critical section.

6 Results

In this section we present our microbenchmark and macrobenchmark results. We then compare pairs of macrobenchmark runs in an attempt to identify the effect that the individual synchronization mechanisms have on performance.

6.1 Microbenchmark results

We plot completion time of the microbenchmark loop in Figure 3. Since there is no shared data used in the critical section, we do not explore collocation. We measure the throughput of TS and TTS both with and without exponential backoff, MCS, LH and M locks, QOLB, and reactive synchronization (using TS+E for the low-contention case and MCS for the high-contention case). We see that QOLB performs best in all cases, under both low and high contention. TS and TTS perform second- and third-best under low contention (one or two processors), but their performance quickly degenerates for more than four. Adding exponential backoff makes TS and TTS perform worse under low contention, but prevents a severe performance degradation in the presence of numerous requestors. The LH and M locks outperform all primitives other than QOLB under medium contention (four processors).

Under high contention MCS outperforms both LH and M. The difference in performance is attributable to the cache behavior of these primitives and the cache coherence protocol we simulated. Under MCS, a processor always reuses the same queue element (or memory address) to insert itself in the queue. Under both LH and M, queue elements tend to migrate from releasing to acquiring nodes [28]. In SCI, a write to a migrating cache block requires more network transactions than does a write to a block accessed mostly by one processor. Other cache-coherence protocols may not display this behavior.

Magnusson, Landin, and Hagersten [28] state that under high contention, MCS generates one extra cache miss than do LH or M. Careful collocation of the MCS "next" pointer and the lock bit (as implied in the original article [29]) prevents this extra cache miss. Under high contention, this collocation permits two read accesses to be satisfied by a single miss instead of two. For all our experiments we assumed that the MCS tail pointer is indeed collocated with the lock bit, which improves its performance under high contention but hinders performance slightly under medium contention. In MCS, placing the lock bit and tail pointer together can result in extra remote accesses when a node is adding itself to a two-element queue at the same time the head of the queue is freeing the lock. Finally, we see that our reactive synchronization scheme is successful in that it closely tracks the performance of the best software alternative under both low- and high-contention conditions.

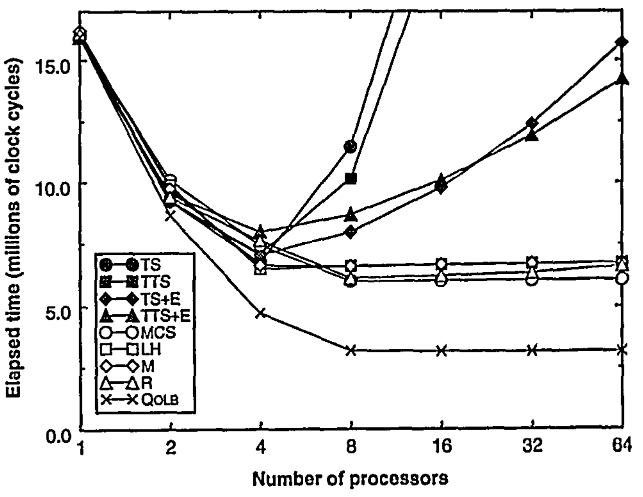

Figure 3 Microbenchmark performance.

6.2 Macrobenchmark results

We present the results of the macrobenchmark experiments in Table 5. TS is the base case for each benchmark and memory model. We list the simulated execution time of each base experiment (in millions of cycles) in parentheses in the TS row and SEQ column of Table 5. The other numbers in Table 5 are all speedups relative to their particular base case. The running times that we present correspond to the entire execution of the benchmarks.

What is most striking about these results is the magnitude of the speedups, considering that the only parameter being varied is the synchronization primitives. Raytrace executes twice as fast in

	BENCHMARK									
	BARNES		MP3D		OCEAN		PTHOR		RAYTRACE	
EXPERIMENT	SEQ	REL	SEQ	REL	SEQ	REL	SEQ	REL	SEQ	REL
TS	(190)	0.94	(231)	1.02	(16.5)	1.19	(221)	1.16	(826)	1.22
TS+C	1.67	1.85	1.03	1.12	1.31	1.69	0.86	1.13	2.47	2.56
TS+E	1.17	1.40	0.86	1.21	1.12	1.37	0.88	1.22	2.06	2.15
TS+E+C	1.31	1.67	0.90	1.29	1.31	1.68	0.93	1.34	2.56	2.65
TTS	1.02	1.11	1.05	1.11	1.02	1.22	1.04	1.23	1.03	1.12
TTS+C	1.72	1.87	1.09	1.18	1.32	1.70	0.95	1.36	2.54	2.61
TTS+E	1.17	1.40	0.83	1.18	1.11	1.40	0.87	1.21	2.03	2.15
TTS+E+C	1.32	1.66	0.87	1.25	1.26	1.70	0.94	1.35	2.56	2.65
MCS	1.57	1.61	1.18	1.30	1.24	1.55	1.06	1.25	2.31	2.26
MCS+C	1.58	1.63	1.25	1.36	1.25	1.65	1.17	1.37	2.29	2.33
LH	1.21	1.48	0.81	1.12	1.24	1.55	0.87	1.22	2.26	2.31
M	1.21	1.47	0.75	1.06	1.24	1.55	0.87	1.18	2.25	2.29
R	1.19	1.47	0.76	1.08	1.19	1.49	0.87	1.20	2.28	2.35
QOLB	1.79	1.83	1.46	1.60	1.31	1.65	1.11	1.34	2.58	2.64
QOLB+C	1.89	1.92	1.65	1.75	1.34	1.70	1.25	1.51	2.62	2.69
QOLB+C+P	1.89	1.92	1.65	1.75	1.31	1.68	1.26	1.54	2.63	2.70
QOLB+C+CP	1.89	1.93	1.64	1.74	1.35	1.70	1.25	1.53	2.63	2.70

Table 5 Speedups of different synchronization primitives. The numbers in parentheses represent the execution time (in millions of clock cycles) for the particular benchmark running on sequentially consistent hardware. The other numbers represent speedups, calculated as the ratio of the execution time of the base run to that of the optimized synchronization primitive.

30 cases. The smallest speedup, with all four mechanisms employed, is sequentially consistent Pthor with 25%. Although Pthor uses locks more than does Barnes or Ocean, Barnes has bursty streaks of accesses to locks. Lock accesses in Pthor are more evenly distributed, so they do not degrade performance nearly so much as in Barnes (thus leaving less opportunity for improvement). The speedups for Ocean are small not because the mechanisms are ineffective, but because Ocean uses locks less frequently than do the other benchmarks (see Table 4). For all benchmarks, however, QOLB with collocation consistently captures the bulk of the performance improvement to be gained. Our implementation of synchronous prefetching is generally ineffective, speeding up or slowing down the execution by at most 2%.

Three of the benchmarks (Barnes, Ocean, and Raytrace) exhibit similar performance for QOLB and TS (or TTS) with collocation. This is untrue for Mp3d and Pthor, however. Using collocation with TTS improves the performance of Mp3d little, and makes the performance of Pthor deteriorate. The lower performance of Pthor with collocation results from the relatively long length of Pthor's critical sections. These long critical sections give requesters the opportunity to attempt to obtain the lock, pulling both the lock and critical section data out of the holder's cache. This behavior does not occur with QOLB because waiting nodes in a QOLB queue spin on shadow lines, not the actual addresses.

Partial collocation with MCS improves the performance of all benchmarks, except for Barnes and the sequentially consistent runs of Ocean and Raytrace. In these cases collocation either has little impact (Ocean and Barnes) or degrades performance slightly (Raytrace). Unlike TS, MCS causes only a fixed number of memory operations to be issued per synchronization access, thus limiting the disturbance caused by collocation.

Raytrace exhibits much larger speedups than does any other benchmark. The Raytrace base case (TS) is extremely slow (as is TTS). Adding any other mechanism besides local spinning improves the performance of Raytrace substantially. These two primitives perform so poorly because much of the locking is for very small critical sections, for which there is heavy contention. Collocation makes the small critical sections extremely fast. Queue-based locking eliminates the large relative overhead that occurs due to contention when the lock is released.

Adding exponential backoff improves performance moderately for all benchmarks but Mp3d and Pthor in the sequentially consistent runs, in which we observed slowdowns of up to 20%.

Reactive synchronization is generally within 25% of the best performing synchronization primitives (disregarding the collocation mechanism and the QOLB runs). The exceptions are the sequentially consistent run of Barnes and Mp3d, where reactive synchronization is 32% and 53% slower than MCS, respectively.

6.3 Individual mechanisms

This section isolates the performance contributions of the individual mechanisms in Section 3. Figure 4 shows performance differences between eight pairs of experiments (for each benchmark). Each pair of experiments isolates one particular mechanism. There is doubtless interaction between an "isolated" mechanism and the other components of the synchronization primitive. This decomposition is not intended to quantify the performance contribution of individual mechanisms definitively, but to aid in understanding of how their combinations affect performance. We also isolate the exponential backoff policy. We list the isolated mechanisms or policies below, along with their corresponding experiment pairs:

	EXPERIMENT PAIR	
ISOLATED MECHANISM OR POLICY	With	Without
Local spinning	TTS	TS
Exponential backoff	TTS+E	TTS
Queue-based locking	MCS	TTS
	QOLB	TTS
Collocation	TS+C	TS
	TTS+C	TTS
	QOLB+C	QOLB
Synchronous prefetch	QOLB+C+CP	QOLB+C

All runs in Figure 4 assume a sequentially consistent memory model. The y-axis plots speedup. Figure 4 shows that local spin-

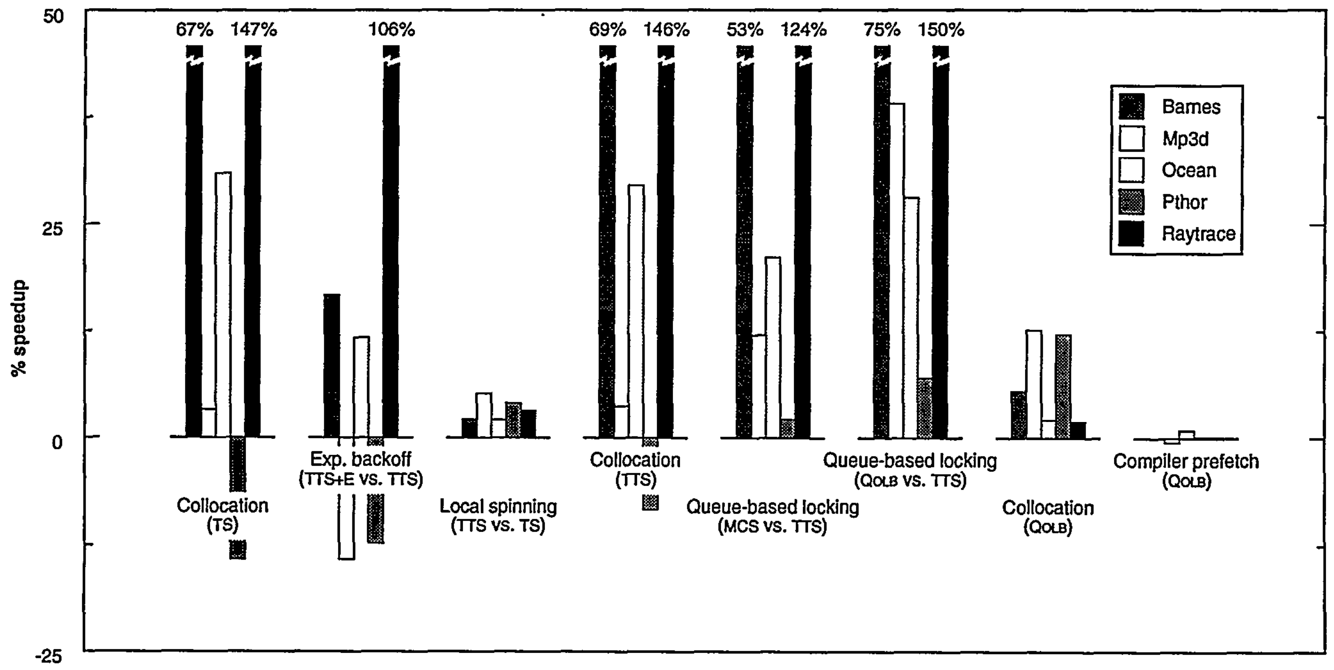

Figure 4 Effects of individual mechanisms.

ning is generally ineffective. Queue-based locking (using MCS) increases speedup for all benchmarks. Using collocation with TS and TTS causes very different behavior across the benchmarks: reducing speedup (Pthor), having a negligible effect (Mp3d), causing a moderate increase (Ocean), and causing a large increase (Barnes and Raytrace). This high variance with collocation exists because requesters may either steal the data from the lock holder, hurting performance, or prevent extra remote transfers into a network filled with arbitration traffic, thus mitigating exceptionally poor performance.

Synchronization prefetching is ineffective, never affecting the running time by more than 2%. We suspect that there is much more opportunity for improvement with synchronous prefetch, as the compiler algorithm was not as aggressive as possible, and we did not restructure the codes or algorithms to exploit the power of the QOLB prefetch operator.

7 Cost of QOLB

In Section 6, we showed that QOLB outperforms all other synchronization primitives in all cases. This performance comes with an associated cost. Most of the other primitives that we discuss can be implemented almost entirely in software, requiring only an atomic memory operation, such as SWAP, in hardware. QOLB, conversely, requires additional hardware support. In this section, we enumerate the additional mechanisms that QOLB requires, discuss the cost spectrum of possible implementations of these mechanisms, and present performance results of a low-cost implementation of QOLB that runs on a cluster of commodity workstations.

QOLB requires four mechanisms for a fully functional implementation: non-blocking synchronizing instructions, direct node-to-node transfer of the lock (from lock releaser to acquirer), storage of the queue state information (such as the next node in the queue), and the capability for multiple nodes to perform operations on the same address without invoking the coherence protocol (the "shadow line" described in Section 4.6). The highest-performance implementation of QOLB requires hardware support in both the processor and the memory system: specialized non-blocking QOLB instructions in the processor, plus extra state, direct cache-to-cache transfer of the lock, and "shadow copy" support in the cache. The SCI standard's implementation of QOLB incorporates the latter three mechanisms,[1] for example, and leaves the processor implementation undefined.

Much lower-cost implementations, which achieve much of the potential performance of QOLB, are possible. QOLB instructions in the processor may be replaced with generic non-blocking loads or stores, eliminating the need to modify a commodity processor. To use generic memory operations, the memory controller must be able to recognize the issued instructions as synchronization operations. These operations may be "flavored," if the processor supports such loads and stores, or they may be memory-mapped into a special "synchronization space." These operations must also be marked uncachable, lest they hit in on-chip caches and never reach the memory controller.

In addition to the processor support, low-cost memory system alternatives for QOLB exist. Recent multiprocessor implementations have begun to use protocol processors at individual nodes to handle inter-node communication. These implementations may use a custom protocol engine, such as Wisconsin Typhoon [34], Stanford FLASH [21], or Sequent STiNG [27], or a commodity protocol processor with some additional off-chip hardware support [35]. These protocol engines can store the QOLB state in either specialized storage or main memory, send direct messages to ship the lock bit to the next waiting processor in a queue, and bypass (or supplement) the global coherence protocol to permit shadow spinning.

To determine if low-cost implementations of QOLB will still outperform other primitives, we compared the performance of QOLB, MCS, and a message-based centralized queue lock (CQL) [39] implemented on an unmodified cluster of commodity workstations. The workstations used the Blizzard run-time system [40] to provide the illusion of shared memory. Blizzard is an implementation of the Tempest interface [34] which, through user-level software, lets users customize the behavior of shared memory to suit the needs of their parallel applications. MCS and CQL are part of the locally available Blizzard distribution and are implemented directly on top of the Tempest interface. We implemented QOLB using the Tempest interface. Our implementation follows closely the QOLB specification in the SCI standard [43]. Specific details on the lock implementations are described elsewhere [19]. Our cluster of workstations consist of 40 unmodified dual processor Sun SPARCStation 20s, each with two 66-MHz HyperSPARC processors [36] and a Myricon Myrinet interface [4]. For our measurements, we used only a single processor per node; that processor is responsible for executing both the program and the Tempest handlers. The detection of message arrival is achieved through polling. A binary rewriting tool [22] automatically inserts polling instructions in the parallel program.[2] The time spent polling is minimized by exploiting the coherence protocol in the memory bus. The polling code checks the status of the network interface through an access to a cachable location, thus limiting the number of these accesses that require the bus to complete. The network interface uses its DMA interface to update the polled location [31]. We set the cache block size to 128 bytes.

To evaluate these implementations, we used a microbenchmark similar to that described in Section 5.3. To explore the impact of collocation, this microbenchmark does not wait a fixed amount of time in the critical section; instead, it writes a value into a shared-memory location. If the synchronization primitive permits collocation, this location may be collocated with the lock. As before, once a processor exits the critical section, it waits for a randomly generated amount of time (selected from a uniform distribution with a mean of approximately 1350μs). The benchmark executes the loop body 100,000 times, divided evenly among the contending nodes.

In Figure 5, we show the elapsed time (in seconds) of the microbenchmark loop under contention levels ranging from one to 16.[3] The figure depicts the elapsed time for four synchronization configurations: MCS, CQL, QOLB, and QOLB with the lock and the variable collocated (QOLB+C). When there is no contention, MCS performs better than either CQL or QOLB. The difference is due to the fact that the latter two implementations require invocation of protocol handlers to acquire or release a lock, while MCS can perform the same operations using simple loads and stores that hit in the cache. Under high contention, QOLB+C outperforms the other primitives. In the 16-node configuration, QOLB+C completes the loop 5.6 as fast as MCS and 2.6 times as fast as CQL. CQL and QOLB perform similarly, with CQL being about 10% faster than QOLB (without collocation) under high contention.

Considering message counts only we could conclude that QOLB should clearly outperform CQL. Indeed, under high contention, QOLB has a single message on the critical path (see Table 3), while CQL has two (one message from the releaser to the lock manager and another one from the manager to the acquirer). The observed behavior is due to the transmission times of the messages used by these implementations; CQL uses short messages to communicate

1. To our knowledge the SCI standard is the only design that includes QOLB.

2. That tool is also responsible for inserting checks before each shared-memory access [39].

3. Due to a shortcoming in the Myrinet interface we could not collect numbers with more than 16 nodes.

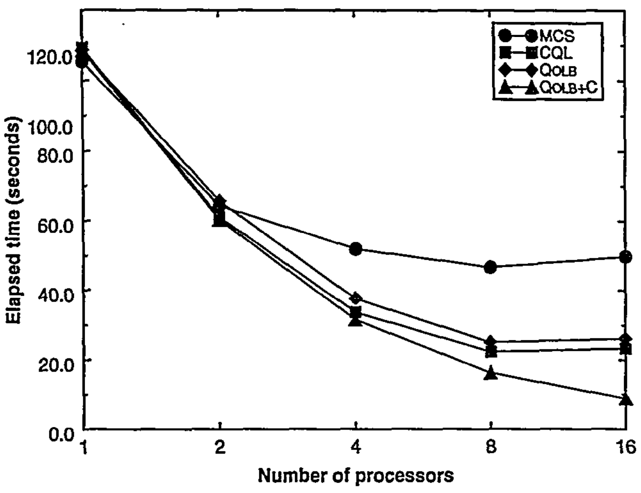

Figure 5 **Performance of software QOLB.**

with the node managing the centralized queue, while QOLB transfers entire cache lines of 128 bytes. On our system, the round trip time of a message carrying a cache line is roughly twice the round trip of a short message.

8 Summary and conclusion

This paper focused on providing efficient locking primitives to improve the performance and scalability of fine-grain shared-memory parallel programs. Instead of focusing on the individual latencies associated with mutually exclusive accesses to critical sections, we focused on the global throughput of critical section accesses. We defined the notion of a *synchronization period*: one "cycle" of multiple serialized accesses to a critical section. We broke this time into three phases (*Transfer*, *Load/compute*, and *Release*), and classified the components of each of these phases as either unavoidable latencies or removable overheads. We identified four optimizing mechanisms (*local spinning*, *queue-based locking*, *collocation*, and *synchronous prefetch*) that can assist in eliminating the removable overheads of critical section accesses.

We performed a thorough evaluation of this space, simulating the performance of seventeen locking constructs (formed from six base primitives: TEST&SET, TEST&TEST&SET, MCS, LH, M, and QOLB) in detail with both real parallel applications and the more traditional microbenchmarks. We also demonstrated the performance of our synchronous prefetching compiler. Finally, we compared the performance of three queue-based locking schemes running on an unmodified cluster of workstations, the results of which support our simulation results.

Our results showed that local spinning consistently aids performance but not very much. Queue-based locking was very effective, except in the cases where the overhead of MCS, LH, and M locks hurt low-contention critical section access latencies. Collocation of the lock and locked data in the same cache line showed wildly different effects with TEST&SET and TEST&TEST&SET; collocation may greatly increase or decrease performance, depending on the benchmark. Collocation consistently improved the performance of QOLB. Synchronous prefetching was the least effective of any of the mechanisms.

The most important result of our experiments is the consistent and large performance gain that QOLB achieves, which is further increased by collocation. Graunke and Thakkar [16] concluded that "... elaborate hardware [synchronization] schemes are unnecessary even when considering larger non-bus-based [systems]." Mellor-Crummey and Scott stated [30] that "special purpose synchronization mechanisms, such as QOLB, are unlikely to outperform our MCS lock by more than 30%." Our results refute these assertions; QOLB outperforms MCS by 40% for Mp3d.

Lim and Agarwal claimed [26] that reactive synchronization "reduces the motivation for providing hardware support for queue locks." Since QOLB outperforms the best software locks under either low- or high-contention conditions, it should also outperform reactive synchronization schemes. Our results confirm this hypothesis—QOLB speedups were from 10% to 92% higher than reactive synchronization, and this disparity only increased by adding collocation and synchronous prefetch to QOLB.

Finally, we claim that the inherent cost requirements of QOLB are not prohibitive. Hardware queue-based locking is not prohibitively expensive, as DASH implemented one such synchronization scheme [25] (it differs from QOLB in that the centralized memory directory kept track of queued requesters). QOLB is an integral part of the SCI standard [43], and uses many of the same mechanisms needed to implement the coherence protocol. As we showed in Section 7, many current- and next-generation multiprocessors already contain most of the hardware support needed to implement hardware-supported QOLB. We also showed that a low-cost, lower performance version of QOLB can be implemented on current systems with no additional hardware support and still outperform the alternatives.

Acknowledgments

We are also thankful to the following people: David Wood first proposed to decompose synchronization primitives into mechanisms. Mary Vernon commented on drafts of this paper. Satish Chandra, Babak Falsafi, Steve Reinhardt, Ioannis Schoinas, Brian Toonen helped clarify some fine points of Tempest and Blizzard. We benefited from discussions with T. N. Vijaykumar.

References

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

[2] Thomas E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II Software, pages 170–174, August 1989.

[3] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] Nanette J. Boden, Danny Cohen, Robert E. Feldermann, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[5] Douglas C. Burger and James R. Goodman. Simulation of the SCI Transport Layer on the Wisconsin Wind Tunnel. In *Proceedings of the Second International Workshop on SCI-Based High-Performance Low-Cost Computing*, March 1995.

[6] Douglas C. Burger and David A. Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 22–31, April 1995.

[7] Convex Computer Corporation, Richardson, TX. *SPP1000 Systems Overview*, 1994.

[8] Travis S. Craig. Building FIFO and Priority-Queueing Spin Locks from Atomic Swap. Technical Report 93-02-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, February 1993.

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

[10] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[11] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for Different Memory Con-

sistency Models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, 1992.

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

[13] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Shared-Memory Multiprocessors. In *Proceedings of the Third Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, April 1989.

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

[15] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer–Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[16] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[17] International Business Machines, Inc., Poughkeepsie, NY. *IBM Systen/360 Principles of Operation*, ninth edition, May 1970.

[18] Alain Kägi, Nagi Aboulenein, Douglas C. Burger, and James R. Goodman. Techniques for Reducing the Overheads of Shared-Memory Multiprocessing. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 11–20, July 1995.

[19] Alain Kägi and James R. Goodman. SoftQOLB: An Ultra-Efficient Synchronization Primitive for Clusters of Commodity Workstations. Technical Report 1327, Computer Sciences Department, University of Wisconsin, Madison, WI, November 1996.

[20] R. E. Kessler and J. L. Schwartzmeier. CRAY T3D: A New Dimension for Cray Research. In *Proceedings of the 38thIEEE Computer Society International Conference (COMPCON)*, pages 176–182, February 1993.

[21] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[22] James R. Larus and Eric Schnarr. EEL: Machine Independent Executable Editing. In *Proceedings of the 1995Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[23] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[24] Thomas Lengauer and Robert E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[25] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[26] Beng-Hong Lim and Anant Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, October 1994.

[27] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 304–315, May 1996.

[28] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient Software Synchronization on Large Cache Coherent Multiprocessors. Technical Report T94:07, Swedish Institute of Computer Science, Kista, Sweden, February 1994.

[29] 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.

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

[31] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interface for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[32] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, August 1985.

[33] 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 *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 48–60, May 1993.

[34] Steven K. Reinhardt, James L. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[35] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 35–44, May 1996.

[36] ROSS Technology, Inc., Austin, TX. *SPARC RISC User's Guide: hyperSPARC Edition*, third edition, September 1993.

[37] Larry Rudolph and Zary Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[38] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

[39] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lukas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing Fine-Grain Distributed Shared Memory on Commodity SMP Workstations. Technical Report 1307, UWCS, March 1996.

[40] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, October 1994.

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

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

[43] IEEE Computer Society. Scalable Coherent Interface (SCI). ANSI/ IEEE Std 1596-1992, August 1993.

[44] Teruo Utsumi, Masayuki Ikeda, and Moriyuki Takamura. Architecture of the VPP500 Parallel Supercomputer. In *Proceedings of Supercomputing '94*, Washington, D.C., November 1994.

[45] Webster. *Webster's Seventh Dictionary*. 1965.

[46] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.