SOFTQOLB: An Ultra-Efficient Synchronization Primitive for Clusters of Commodity Workstations

Alain Kägi and James R. Goodman

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706
galileo@cs.wisc.edu

Abstract

Efficient synchronization can dramatically improve the performance of shared-memory parallel programs. Past work has proposed pairwise synchronization primitives—such as Queue-on-Lock-Bit (QOLB) [6, 5] and the transactional memory model [7, 26]—that offer higher performance than currently widely used primitives— such as MCS locks [16]. QOLB and the transactional memory model, as originally conceived, require modifications to commodity processors, and thus have not yet been implemented. In this report, we present an implementation of QOLB, called SOFTQOLB, that runs entirely in software, and can thus run on unmodified commodity workstation clusters. We describe our implementation in detail and present an evaluation of its performance using a microbenchmark. We show that the SOFTQOLB primitive consistently outperforms MCS when there is high contention (SOFTQOLB is in these cases at least 5 times as fast as MCS) and the performance of these two primitives is comparable under low-contention conditions.

1 Introduction

Distributed-memory parallel machines supporting the shared-memory paradigm are becoming an attractive alternative for CPU and memory intensive applications (e.g., database applications, world-wide web servers, graphic processing, and simulations of physical phenomena). The following facts support this claim:

- The shared-memory model gives the programmer a familiar framework to develop parallel applications. It also eases the parallelization of existing sequential applications.
- The advent of affordable desktop symmetric multiprocessors (SMPs) will increase the parallel application base.
- The successful development of shared-memory multiprocessing standards [25] reduce the time to market by decreasing design time and by letting manufacturers use commodity parts. Both the Convex Exemplar [4] and the Sequent STING [15] relied on these standards.
- The emergence of low-cost, fine-grain software implementations of shared-memory, such as Shasta [22] or T0 [20] further reduce the cost of supporting the shared-memory model.
- Finally, successful research prototypes such as the Stanford DASH [13] 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.

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-grained 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.

Many locking primitives that perform better than currently implemented alternatives have been proposed [5, 7, 26]. One such locking primitive is the Queue-On-Lock-Bit (*QOLB*) synchronization primitive proposed by Goodman, Vernon, and Woest [5]. A study [9] has shown that QOLB performs better (up to 100%) than MCS [17], an efficient all-software queue-based locking primitive inspired by the QOLB work. However, more efficient locking primitives in general and QOLB in particular are currently not available on multiprocessors because, as described, they require modifications to current generation commodity processors.

Support for synchronization in current generation commodity processors consists of instructions such as SWAP or the pair LOAD-LINKED and STORE-CONDITIONAL [8]. With them a number of synchronization primitives can be readily synthesized, such as TEST&SET and MCS. However these instructions are not well suited to support the more efficient synchronization techniques proposed recently.

Recent trends in the design of fine-grained distributed shared-memory systems present an opportunity to support very efficient synchronization primitives cheaply.[1] Such systems expose mechanisms so that system programmers or users can specify cache coherence protocols in software and/or optimize them for given applications [11, 19, 15, 22]. In particular, these mechanisms allow us to tailor the behavior of shared memory to support more efficient synchronization.

One such system is the Blizzard run-time system [24]. Blizzard uses an executable editing tool [12] to support the shared-memory model in software on a distributed-memory multiprocessor. The executable editing tool annotates each access to shared variables to provide system-wide shared memory. Annotations check that each memory access can complete safely (i.e., data is valid for read accesses and data is writable for write accesses); should a check fail the annotations invoke the appropriate protocol handler to bring the faulting memory location to a suitable state.

This report describes an implementation of QOLB on Blizzard called SOFTQOLB. The organization of the remainder of this report is as follows. Section 2 presents an overview of queue-based locking primitives in general and QOLB in particular, Section 3 describes SOFTQOLB — our implementation of QOLB on the Blizzard run-time system, Section 4 explains our experimental methodology, and Section 5 presents a preliminary evaluation of this implementation using a microbenchmark and compares it to other locking alternatives. Finally, Section 6 presents our conclusions.

2 Queue-based locking primitives and QOLB

Queue-based locking primitives reduce synchronization overheads in three ways. First, they create a queue of waiting requesters, thus performing arbitration when the requests are received and not when the current holder releases the lock (as is the case in the TEST&SET primitive, for instance). Second, they reduce lock transfer time by restricting communication to be between the releasing node and acquiring node (although the number of remote access required to perform this transfer will vary among different primitives). Third, they eliminate the overhead of reobtaining the lock by the releaser, since no other nodes access the lock directly until the holder releases the lock (unlike the TEST&SET primitive, for instance).

Goodman, Vernon, and Woest [6, 5] proposed the first queue-based locking scheme, which they called Queue-On-Lock-Bit (*QOLB*). QOLB implements a binary semaphore providing first-come-first-serve service[2] by maintaining a queue of waiting processors. As processors wish to acquire a lock, they join the corresponding queue; when a processor releases a lock, it

1. This observation does not mean that more expensive support cannot achieve better performance.
2. Some implementations are only able to provide approximate first-come first-serve service.

leaves the queue and forwards the lock directly to the next waiting processor, if any.

In addition to implementing queue-based locking, QOLB also supports *local spinning*, *collocation* (of lock and protected data), and *synchronous prefetch*. QOLB avoids unnecessary network traffic as waiting processors spin locally, repeatedly accessing a local "shadow" copy of the lock's address without generating network traffic. The shadow copy acts as a placeholder for the lock that the requester will eventually receive. In an implementation using caches, the shadow copy could be a cache line allocated when the processor issues a lock request. The cache line would contain stale data until lock arrives. QOLB permits data to be associated (collocated) with the lock. Execution of a critical section typically requires access to data. These data are ideal candidates for collocation. When the current lock holder releases the lock, it ships not only the lock but also the collocated data to the next processor in the queue. When the next processor receives the lock, it will not waste extra time requesting the collocated data as they are already locally available. Finally, QOLB is a non-blocking primitive, it allows the waiting processor to request the lock ahead of time, thus ideally overlapping protected data prefetching with other useful work.

Inspired by QOLB, Anderson, and Mellor-Crummey and Scott proposed pure software solutions to minimize network traffic and synchronization access latencies. Mellor-Crummey and Scott (MCS) implement a queue as a software linked list, and use atomic operations such as SWAP and COMPARE-AND-SWAP to update the list correctly [17]. Anderson presented a scheme that implements a queue as a circular array [2]. Like QOLB, these algorithms also reduce the network traffic to a constant number of traversals per synchronization access (however these schemes require at least six network traversals versus one for QOLB [1]) and allows processors to spin locally while waiting for the lock release. Unlike QOLB, these algorithms are unable to prefetch lock and cannot easily benefit from collocation.

The QOLB protocol can be implemented on many types of computing platforms including cacheless and bus-based cache-coherent multiprocessors (e.g., the multicube [6]). To illustrate how QOLB works and how a program can use this primitive, the following discussion assumes a distributed cache-coherent multiprocessor similar to the DASH system [13].

Consider a cache-coherent multiprocessor consisting of nodes connected by an arbitrary network. Each node contains a processor, a cache, a fraction of the distributed memory, and an interface to the interconnecting network. In such a system, extending the cache-coherent protocol to support QOLB is a convenient alternative requiring few changes to the base hardware structure and protocol. These changes introduce extra cache states to support QOLB and its interaction with the base cache coherence protocol. QOLB also requires a new field with each cache line to store the queuing information (usually in the form of a pointer to the next processor in the queue). Figure 1(a) illustrates the extended cache entry. In this example, we assume a distributed queue built using a linked list with the pointer pointing to the next queued element. Other queue implementations are also possible. For instance, on a bus-based system, it may be more advantageous to build a linked list with pointers pointing towards the previously queued element. Collocation is simply achieved by storing protected data in the cache line upon which QOLB operations are performed.[1] Figure 1(b) shows how cache entries are connected to form a three-element queue. Unless otherwise specified, we assume in this section 128-byte cache lines and we assume that the program has allocated the line as follows: one word (four bytes) for a lock and 31 words for data protected by that lock. Figure 1(c-e) depicts the transfer of a lock from node A to node B. Initially, node A owns the lock and no other nodes have requested access to the lock (see Figure 1(c)). Then, while node A is still in the critical section, node B requests the lock. This action allocates a shadow copy of the cache line and sends a request to node A. Upon receiving the request, node A changes its cache state from being a single element queue to being at the head of the queue, and updates the pointer field to

1. If the critical data cannot all fit in one cache line, collocation will only partially reduce the overheads of accessing protected data.

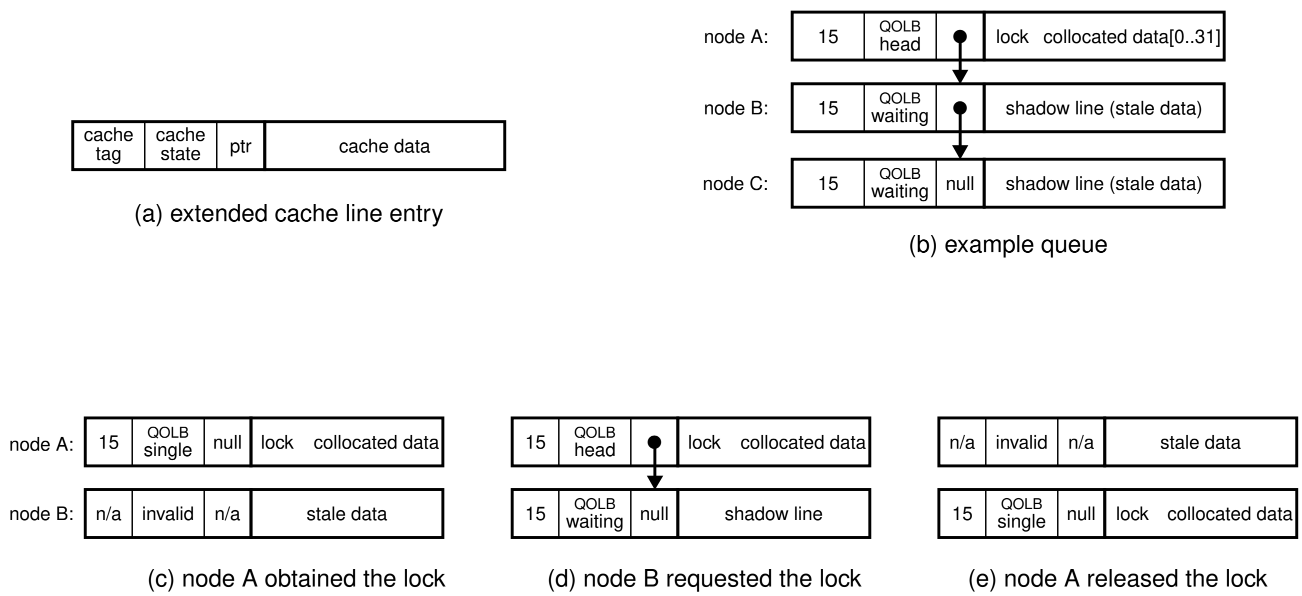


Figure 1. A QOLB implementation and QOLB operations

point to node B (see Figure 1(d)). When node A releases the lock it ships the corresponding cache line directly to node B. Upon reception, node B updates its state to being a single element queue (see Figure 1(e)).

The QOLB programming interface consists of two functions: ENQOLB and DEQOLB. ENQOLB is a non-blocking operation that allocates a shadow copy of the cache line and sends a request to join the queue. It returns true if the corresponding cache line is already at the head of the queue, false otherwise. DEQOLB ships the cache line to the next processor. Figure 2 shows an example of how QOLB is used to access data in a critical section. We have expressed all the algorithms in this report using the C programming language [10]. The first call to ENQOLB sends a message that inserts the requester into the 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 on the allocated shadow copy until the owner releases the lock and sends it directly to the waiting node. When the ENQOLB returns true, the processor is at the head of the queue, has received the lock, and thus, can enter the critical section. The processor relinquishes the lock with a call to DEQOLB, at which point the cache line and its content is sent directly to the next waiting processor.

The implementation just described only provides approximate first-come first-serve service. The reason for this approximation is that a cache conflict may cause the replacement of a line currently in a QOLB state. When the cache replaces the line, its place in the queue is lost. Solutions that preserve fairness are possible. They must maintain the information regarding a cache line's position in the queue. These solutions include additional buffering or pinning a cache line preventing eviction by the replacement algorithm. All these solutions introduce complexity and add to the cost of the overall system. Alternatively, the processor can forsake fairness and rejoin the queue at its end. Considering today's large caches and the relatively short-lived (with the respect to a particular cache) migrating nature of a lock, replacement of a cache line in a QOLB state is probably infrequent enough not to justify the added cost and complexity of a solution preserving fairness. SCI and the implementation described in this section chose the less expensive approach of maintaining only approximate fairness.

```
/* DECLARATION */

STRUCT _LOCKED_DATA {
     INT LOCK;
     INT DATA[31];                                  /* 128-BYTE CACHE LINE, 4-BYTE WORDS */
};

/* CODE */

VOID
CRITICAL_SECTION(STRUCT _LOCKED_DATA *X) {          /* ASSUMPTION: CONTENT AT X IS CACHE-BLOCK ALIGNED */
     ENQOLB(&X->LOCK);                              /* PREFETCH LOCK AND DATA */
     /* VARIOUS COMPUTATION HERE */
     WHILE (!ENQOLB(&X->LOCK)) { /* DO NOTHING */ ;}   /* SPIN */
          /* CRITICAL SECTION USING X->DATA */
     DEQOLB(&X->LOCK);                              /* RELEASE LOCK */
}
```

Figure 2. QOLB code example assuming an infinite cache

Because of cache line replacements, and assuming a system that only maintains approximate fairness, a processor cannot claim to have successfully obtained a lock solely on the basis of having reached the head of the queue. Indeed, the current lock holder may have seen its line evicted from the cache before it had a chance to release the lock. In such an event the cache replacement policy forces the processor prematurely to leave the queue and to send the corresponding cache line to the next processor in the queue. To prevent a processor from misconstruing getting a lock, one possible solution consists of shadowing the QOLB operations with conventional atomic operations to acquire and to release the lock correctly. The code fragment in Figure 3 illustrates this approach, maintaining correct locking semantics in spite of cache replacement. When the code finds the cache line local, it attempts an atomic operation (such as TEST&SET). If the operation succeeds, the processor has successfully obtained the lock and can enter the critical section. If the operation fails, it implies that some other processor is still holding the lock but had to evict the corresponding cache line. An atomic operation that failed forces a processor to restart the sequence of instructions at the beginning of the outer loop. Attempts at obtaining the lock will fail until the lock holder clears the conventional lock by storing zero in the corresponding address.

```
/* DECLARATION */

STRUCT _LOCKED_DATA {
     INT LOCK;
     INT DATA[31];                                  /* 128-BYTE CACHE LINE, 4-BYTE WORDS */
};

/* CODE */

VOID
CRITICAL_SECTION(STRUCT _LOCKED_DATA *X) {          /* ASSUMPTION: CONTENT AT X IS CACHE-BLOCK ALIGNED */
     ENQOLB(&X->LOCK);                              /* PREFETCH LOCK AND DATA */
     /* VARIOUS COMPUTATION HERE */
     DO {
          WHILE(!ENQOLB(&X->LOCK)) { /* DO NOTHING */ ; }   /* SPIN WAITING FOR THE LOCK TO BECOME LOCAL */
     } WHILE (TEST&SET(&X->LOCK));                  /* SPIN WAITING ON THE LOCK */
          /* CRITICAL SECTION USING X->DATA */
     X->LOCK = 0;                                   /* CLEAR THE LOCK */
     DEQOLB(&X->LOCK);                              /* RELEASE LOCK */
}
```

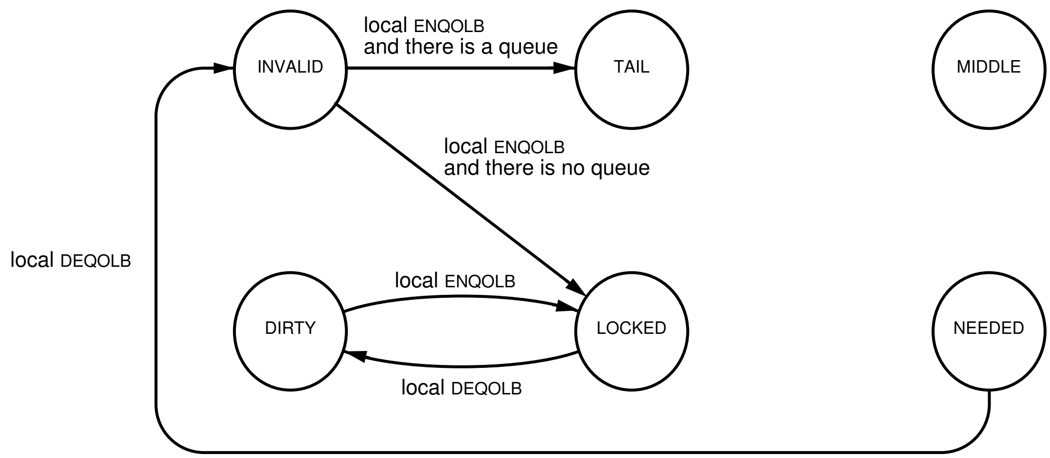
Figure 3. QOLB code example assuming a finite cache

3 SOFTQOLB

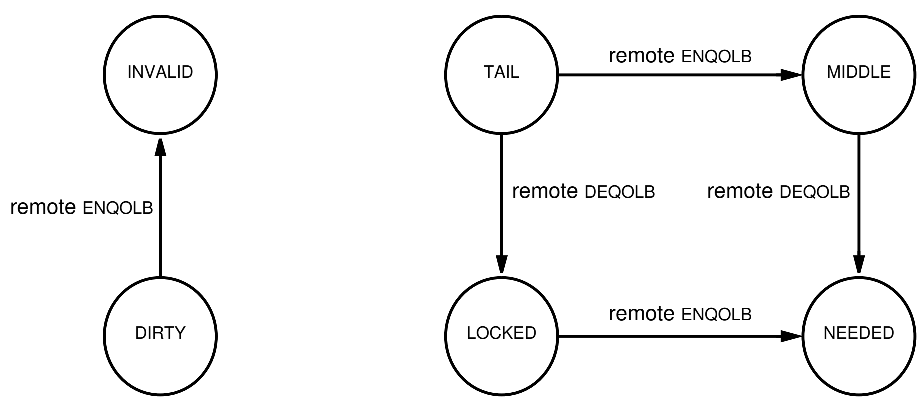
SOFTQOLB is an implementation of QOLB on top of the Tempest interface [19]. Tempest is an architectural interface for distributed parallel systems. It provides a portable platform to write parallel applications. Tempest supports the message passing and shared-memory models, as well as hybrid models constructed by the users.

The QOLB specification in the SCI standard [25] inspired this specification of SOFTQOLB. Unlike the implementation of QOLB in SCI, our SOFTQOLB prototype does not currently support access to data collocated with the lock without the ENQOLB and DEQOLB primitives. The QOLB implementation in SCI supports such "unsynchronized" accesses. A read or a write access fault invokes the standard invalidation-based protocol. If a QOLB queue exists for the corresponding address, the SCI protocol collapses the queue and ships the cache line at the head of the queue (i.e., the most recent data) to the faulted processor.

Figure 4 depicts the QOLB protocol we implemented. Intermediate states are not shown. Figure 4 (a) shows transitions effected by QOLB calls issued by the local processor, Figure 4 (b) shows transitions effected by QOLB calls issued by remote



(a) Transitions effected by the local processor issuing QOLB calls.



(b) Transitions effected by remote processors issuing QOLB calls.

Figure 4. A partial state diagram

processors. A cache line is initially in state INVALID in all caches. If a processor calls ENQOLB, the cache line state at that processor becomes LOCKED. When that processor calls DEQOLB and no other processor has requested the cache line, the state becomes DIRTY, that is, the line is writable but not locked. That processor can issue a sequence of ENQOLB and DEQOLB operations and, while no other processor requests that cache line, the state moves between LOCKED and DIRTY without requiring global communication. If another processor calls ENQOLB and the corresponding request finds the cache line in state DIRTY, the cache line is shipped immediately to the requesting processor. The state at the old lock owner becomes INVALID. Alternatively, if the other processor calling ENQOLB finds the cache line in state LOCKED a queue forms. The cache line in the processor holding the line moves to state NEEDED to reflect the fact that the line has been requested. The requester becomes the "TAIL" of the queue. If additional processors call ENQOLB the queue grows: the latest requester becomes the new "TAIL" and the older "TAIL" processor becomes "MIDDLE." When the processor currently holding a requested cache line calls DEQOLB, its state moves from NEEDED to INVALID and the cache line is sent to the next processor in the queue. The processor receiving the cache line moves to state NEEDED if it was in state MIDDLE, or moves to state LOCKED otherwise.

4 Experimental methodology

To determine if low-cost implementations of QOLB will still outperform other primitives, we compared the performance of SOFTQOLB, MCS, and a message-based centralized queue lock (CQL) [23] implemented on a cluster of commodity workstations. The workstations used the Blizzard run-time system [24] to provide the illusion of shared memory. Blizzard is an implementation of the Tempest interface [19]. MCS and CQL are part of the locally available Blizzard distribution and are implemented directly on top of the Tempest interface. We wrote SOFTQOLB as a set of protocol handlers and bundled the resulting object code in a library that is linked in with applications as desired.

We evaluated these implementations with the microbenchmark shown in Figure 5. This benchmark repeatedly accesses a critical section in a loop (the benchmark accesses the critical section a total of 100,000 times; these accesses are divided evenly among the contending nodes). Once in the critical section, a processor writes a value into a shared variable (all processors access a common shared variable). After release, the releasing processor waits for a random amount of time selected from a uniform distribution. The mean of the distribution is 1,400μs (which corresponds roughly to 20 times the round trip time of a message carrying data). This random time reduces the likelihood of a processor reacquiring the lock

```
/* DECLARATION */

STRUCT {
      LOCKDEC(LOCK);                /* DECLARES A LOCK */
      CHAR PADDING[124];            /* REMOVE PADDING IF COLLOCATION IS DESIRED! */
      INT VAR;                      /* COMMON SHARED VARIABLE */
} GMEM;

/* CODE */

FOR (I = 0; I < 100,000/N_PROC; I++) {
      LOCK(GMEM.LOCK);              /* ENTER CRITICAL SECTION */
      GMEM.VAR = 1;                 /* WRITE A VALUE IN COMMON SHARED VARIABLE */
      UNLOCK(GMEM.LOCK);            /* LEAVE CRITICAL SECTION */
      DELAY(RANDOM() % MAX);        /* WAIT A RANDOM AMOUNT OF TIME */
};
```

Figure 5. Microbenchmark algorithm

immediately. 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. The methodology is similar to that used by both Anderson [2], and Lim and Agarwal [14] to measure raw critical section throughput. However, inside the critical section, instead of writing a value into a shared variable, their microbenchmark waits for fixed amount of time. We made that change to explore the impact of collocation on synchronization throughput. Synchronization primitive permitting, we ran experiments with both the lock and the shared variable residing in different cache lines, and the shared variable collocated with the lock.

Our cluster of workstations consists of 40 unmodified dual processor Sun SPARCStation 20s, each with two 66-MHz HyperSPARC processors [21] and a Myricon Myrinet interface [3], running the manufacturers' Solaris release 5.4 operating system. The network topology is a regular tree of degree six. A tree depth of two is sufficient to connect up to 36 nodes. The router at the top level adds about 0.5μs latency to messages having to switch subtrees [3].

The detection of message arrival is achieved through polling. A binary rewriting tool [12] automatically inserts polling instructions and checks before each shared-memory access in the parallel application. By default Blizzard performs polling through an uncacheable read access to a memory-mapped status register. This method of polling wastes memory bus bandwidth when the incoming message queue is empty. To reduce bus contention we exploit the fact that the memory bus on our workstations supports coherent memory transactions. The polling code in all our experiments checks the status of the network interface through accesses to a cacheable memory-mapped location. The network interface updates this cacheable location using its DMA interface [18]. This optimization lets the polling code complete most of the time without requesting the bus, reducing traffic and bus load. The drawback of this optimization is that the polling latency when there are messages waiting in the queue is slightly higher than if it had been performed using the uncacheable access (an increase of about 1μs).

We measure the time it takes for all processors to complete the microbenchmark iterations. We report the best of three measurements in an attempt to reduce the impact of external uncontrollable events (such as the underlying operating system scheduling other processes). The Blizzard system allows a number of parameters to be configured either at compile- or link-time. Cache line size is a compilation parameter that can take values ranging from 32 bytes to four Kilobytes. For our measurements, we used line sizes of 32 and 128 bytes, representative of values that might be used to run fine-grained applications. In all of our experiments the microbenchmark runs with a single thread per node in the system. The protocol code runs either in shared or dedicated mode. In the shared mode, a single processor per node runs both the microbenchmark and the protocol code (the other processor remains idle). In the dedicated mode, the microbenchmark thread runs on one processor and Blizzard assigns the other processor to running protocol handlers. The shared mode is representative of a very cheap configuration (requiring a single processor per node) that will run computation-bound (low communication-to-computation ratio) applications reasonably well. If applications are communication-bound (high communication-to-computation ratio) it may be more advantageous to run them on the more expensive dual processor system in dedicated mode.

5 Results

We plot the completion time (in seconds) of the microbenchmark loop in Figure 6 for a number of nodes ranging from one to sixteen[1] and for one configuration of Blizzard. This configuration assumes 32-byte cache lines, and application and protocol sharing a single processor. We measure the throughput of MCS, CQL, and QOLB; for QOLB we measure the throughput both with and without collocation of lock and data (the suffix +C denotes experiments with collocation).

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

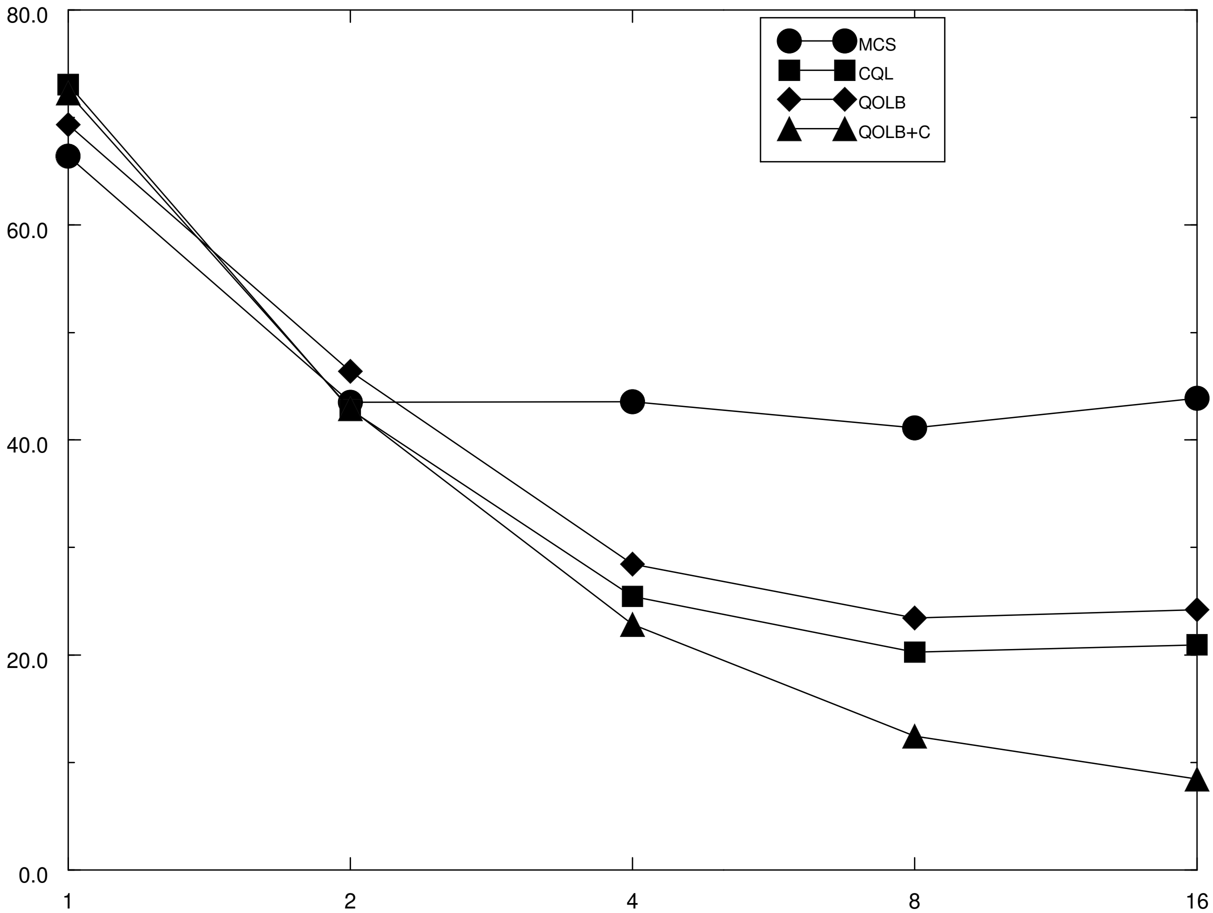


Figure 6. Microbenchmark performance (shared, 32-byte cache line)

When there is no contention, MCS performs better than either CQL or QOLB (10.0% and 8.9% faster respectively). The difference is in large part due to the fact that both CQL and QOLB 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. This advantage of MCS over QOLB and CQL is not inherent, other implementations could reduce, if not eliminate, this performance gap. For instance, the application thread could in limited cases directly process local requests on the behalf of the protocol thread. This off-loading of local protocol processing duties avoids two sources of overheads. First, it eliminates the delays associated with communication between application and protocol threads. Blizzard supports this communication with a queue stored in main memory that is correctly updated with atomic operations. Second, it eliminates the delays associated with thread switches, since whenever the application thread processes a local request itself there is no need to run the protocol thread. Under high contention, QOLB+C performs best, MCS poorest. In the 16-node configuration, QOLB+C completes the loop 5.2 as fast as MCS and 2.4 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, 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 (a single word, 4 bytes) to communicate with the node managing the centralized queue, while QOLB transfers entire cache lines (a message consists of two words plus the 32-byte cache line). On our system, the round trip time of a message carrying a cache line is close to twice the round trip time of a short message. Other implementations could substantially reduce the round trip delay of messages carrying data. For instance, a design could have the network interface copy a message's cache line directly into the processor's cache instead of the node's main memory. The protocol thread would almost certainly then find the data it needs without requesting the memory bus, reducing both the thread's running time and the message round trip delay.

We note that the performance of MCS appears to reach a local minimum near four nodes. Under medium contention it is possible for a node to join the queue as the third entry (updating the "next" field of the second node's data structure) while the current head of the queue is signalling the second entry that it can now enter the critical section (updating the "lock" field of the second node's data structure; the second node is spin-waiting on that field). Since by design these two variables sit in the same cache line, it leads to a longer critical path for medium contention than for higher contention. Under high contention, the update to the "next" field occurs early, therefore this update does not hinder the transfer of the lock. Under medium contention, however, a third party (the new node joining the queue) may be updating the "next" field while the lock holder attempts to clear the "lock" field increasing the time to transfer the lock.

We observe that the performance of MCS, CQL and QOLB worsens by about 5% going from eight to sixteen nodes (the performance of QOLB+C is still improving after eight nodes). Additional network latencies experienced in sixteen-node runs (as compared to eight-node runs) can explain part of that increase.[1] Recall that the network topology is a regular tree of degree six; and the delay through the top-level switch is 0.5µs. For an eight-node run, the top-level switch will see between two (the nodes are ordered in the QOLB queue such that all transfer messages but one remain in the same subtree) and eight (the nodes are ordered in the worst-case scenario) messages per microbenchmark iteration. In this case the top-level switch adds a delay between 1 and 4µs per iteration. Compared to the sixteen-node run, where the top-level switch will see between three and sixteen messages and thus cause an additional network delay between 1.5 and 8µs per iteration. So the increase in latency caused by additional subtree switching going from eight to sixteen nodes is between 0.5 and 4µs. We measured a total increase of 8µs per iteration for QOLB (without collocation) going from eight to sixteen nodes. MCS and CQL will see more additional delay as more messages are exchanged per synchronization access (at least six messages for MCS and two for CQL). For instance, we measured an increase of 28µs per iteration for MCS going from eight to sixteen nodes.

Figure 7 shows the same type of graphs for all the Blizzard configurations we analyzed. The graph in Figure 6 appears again in Figure 7 (inset a). The other configurations are dedicated mode and 32-byte cache line (inset b), shared mode and 128-byte cache line (inset c), and dedicated mode and 128-byte cache line (inset d). All the graphs show similar trends. QOLB+C consistently outperforms MCS by at least a factor of 5 (the speedups range from a factor of 5.2 to 6.4) and QOLB+C is in all cases at least twice as fast as CQL (the speedups range from a factor of 2.4 to 3.2).

There are two striking differences. First, under low contention the throughput is lower for the dedicated mode than for the shared mode. In both modes communication between the application and protocol threads is performed through a queue in shared memory. In dedicated mode this communication involves the bus since the application and the protocol threads run on different processors. In shared mode this communication occurs faster since both threads share the same cache. This difference in performance becomes irrelevant under high contention, where the waiting time for the lock dominates any other latencies.

1. The network contention explains part of, but not all of, the performance loss, we are still investigating the reasons to explain the entire running time increase.

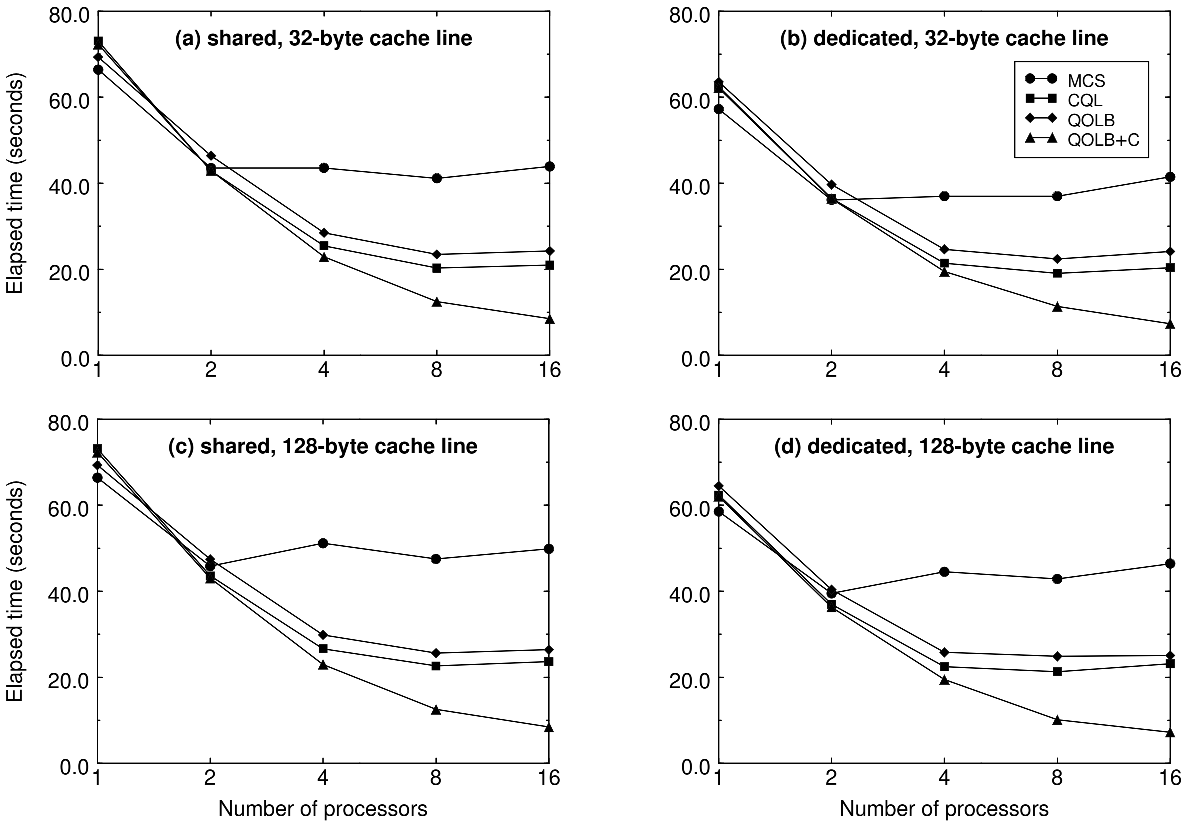


Figure 7. Performance of software QOLB

The second difference is that the local-minimum behavior associated with MCS is more marked for the 128-byte cache line than for 32-byte cache line. The reason is that round trip latency for 128-byte blocks of data is higher than for 32-byte blocks of data increasing the probability of conflict between the releaser and the new requester.

6 Summary and conclusion

Currently available hardware support for synchronization seems to restrict programmers to using primitives that are much less efficient than other primitives proposed in the literature such as QOLB and transactional memory. This restriction stems from the perception that these other primitives require hardware support. This report refutes that notion. With the advent of new fine-grained distributed shared-memory systems that support the shared-memory abstraction in software, it is possible to provide QOLB on systems that do no have special support for this synchronization primitive.

We have described an all-software implementation of QOLB called SOFTQOLB. Using a microbenchmark we have shown that SOFTQOLB outperforms the alternatives under high contention (at least a factor of 2 speedup and up to 3.2) and is only marginally slower when there is no contention (at most a 9% slowdown).

This report shows that efficient synchronization primitives like QOLB are not necessarily bound to the high-performance end of the cost/performance spectrum of synchronization support. A range of implementation alternatives is available to designers. For high-performance implementations designers could provide QOLB support as envisioned when this primitive was first proposed. The original proposal had the protocol tightly integrated with a processor, the instruction set architecture containing the ENQOLB and DEQOLB instructions and the local memory implementing the efficient synchronization protocol directly in hardware. Low-cost implementations would resemble SOFTQOLB: protocol handlers running on a node's processor implement the QOLB protocol.

We believe that relatively minor and inexpensive changes in the design of workstations could have profound impact on the performance of low-cost synchronization support. For instance, we have observed that the performance of SOFTQOLB critically depends on the location of the network interface in a node. A network interface attached to the memory bus and capable of transferring incoming data directly into the processor's cache will almost certainly improve considerably the performance of SOFTQOLB under medium to high contention.

Finally, the emergence of low-cost efficient synchronization primitives may improve the popularity of fine-grained shared-memory applications. The scalability behavior of parallel applications that use locking depends on the overheads associated with synchronization primitives. There is a point where any further division of the computation granularity (i.e., the critical section size) stops leading to commensurate performance improvement. For such applications, this point is reached when the synchronization overheads are dominating the running time of the critical section. Therefore, with lower-overhead synchronization primitives, such as QOLB, programmers can design finer-grained applications that will scale better.

References

[1] Nagi M. Aboulenein, James R. Goodman, Stein Gjessing, and Philip J. Woest. Hardware Support for Synchronization in the Scalable Coherent Interface (SCI). Technical Report 1117, Computer Sciences Department, University of Wisconsin, Madison, WI, November 1992.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[8] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, November 1987.

[9] 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.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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, Computer Sciences Department, University of Wisconsin, Madison, WI, March 1996.

[24] 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.

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

[26] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel & Distributed Technology*, 1(4):58–71, November 1993.