AN ANALYSIS OF SYNCHRONIZATION MECHANISMS IN SHARED-MEMORY MULTIPROCESSORS

by

Philip J. Woest and James R. Goodman

Computer Sciences Technical Report #1005

February 1991

An Analysis of Synchronization Mechanisms in Shared-Memory Multiprocessors

Philip J. Woest and James R. Goodman

Computer Sciences Department
University of Wisconsin-Madison
Madison, Wisconsin 53706

Abstract— The granularity of computation achievable in a shared-memory multiprocessor is limited by the time required for process communication, that is, synchronization and data sharing. Therefore, reducing delays associated with process communication directly affects the minimum parcel of computation that can be efficiently executed. QOLB (formerly called QOSB) is a recently proposed hardware primitive for minimizing communication overhead by overlapping communication delays with computation. We investigate QOLB and a QOLB-inspired software synchronization method, comparing their effectiveness in supporting critical sections (*i.e.* locks). We show that substantial reductions in communication latency may be achieved over other hardware and software mechanisms by using QOLB's ability to simultaneously synchronize on and prefetch shared data, allowing these communication latencies to be overlapped with computation. We also analyze the performance of additional hardware and software techniques for implementing locks, counters, and barriers. Some observations are reported on the efficiency of various methods of notifying processes of an event, and on the effects of locality on the performance of barrier synchronization methods.

1. Introduction

Parallel computing is not an end in itself. It is necessary in order to achieve the high level of performance required to solve the biggest problems, and even to provide cost-effective solutions to moderately large problems. The advent of ever-faster processors at moderate costs focuses the drive for high performance on parallel computing. Thus parallel computing is confronted with the need to coordinate increasing numbers of processors, while at the same time the individual processors are becoming more powerful.

One might hope that arbitrarily powerful systems could be built simply by connecting together a sufficiently large number of processors, limited only by cost. Experience has shown that this is not the case. Any parallel algorithm requires processors to communicate periodically, that is, to synchronize and to share data. While some algorithms allow large amounts of computation between points of communication, many others do not, or at least complete more rapidly if communication is more frequent.

Conceptually, one might add processors incrementally, thereby reducing the granularity of the computation between points of communication. But because communication also involves some delay, the benefit of adding more processors is diminished until, when most of the time is spent on communication, little further improvement can be achieved. In fact, as the number of processors grows, the communication time will almost certainly increase as the interconnection network and memory system becomes more complicated. Thus, there is an optimal number of processors beyond which the performance actually declines.

Intuitively, a balanced system is attained when approximately equal time is spent in computation and communication. Adding more (or faster) processors does not provide major speedups — infinitely fast processors would only speed up the application by a factor of two. Neither does increasing the granularity of the computation — the application will only run a factor of two faster if communication time is reduce to zero. At such a point, significant additional speedup can be achieved only by increasing computational power *and* reducing communication delays.

While increasing computational power is straightforward, if not easy, reducing communication time is neither. Communication delays are fundamentally limited by speed-of-light delays, and it is probably not reasonable to expect a significant reduction in physical size for large systems. Thus the absolute time for communicating information is not likely to be reduced greatly.[1]

The above discussion indicates that in order to achieve higher performance, hardware and software techniques must be used to reduce communication delays. This paper explores the performance of such mechanisms in the context of large-scale shared-memory multiprocessors. Section 2 describes hardware and software for synchronization. Section 3 discusses the reduction of communication latency in systems that employs directory-based cache coherence. A number of synchronization algorithms are described and evaluated using simulation and simple analytical techniques in Section 4. Conclusions are presented in Section 5.

---

[1]There is, of course, the opportunity to select algorithms that exhibit geographical locality and to exploit this property to minimize the average communication distance. This does not improve the worst-case delay, and places an additional burden on the programmer to select algorithms that can exploit such locality. It is, nevertheless, part of the solution.

2. Hardware and Software for Synchronization

In parallel processing, process communication can be thought of as having two distinct goals: process synchronization and data sharing. The send and receive primitives of non-shared memory multiprocessors implement both of these functions simultaneously. However, in shared-memory systems these functions are generally carried out by separate mechanisms. Read and write operations are used to move data around a system — in accordance with a coherence protocol if cache consistency is maintained by the hardware — while special read-modify-write primitives are generally provided for process synchronization. We take a brief look at some synchronization mechanisms below and address the problem of reducing latency for both synchronization and data sharing in the next section.

A number of hardware primitives have been proposed as a basis for process synchronization in shared-memory multiprocessors. These include traditional Test&Set and Unset, Test&Test&Set [RuSe84], the full/empty bits of the HEP multiprocessor [Smit81], and Fetch&Add [GoLR83], as well as generalizations, known collectively as Fetch&Φ operations [GGKM83, BrMW85, ZhYe87]. Unfortunately, most synchronization algorithms which rely on these primitives require spinning over the interconnect or, as in the case of using Test&Test&Set to acquire a lock, cause excessive bus traffic.

The QOLB[2] primitive, was proposed [GoVW89] to alleviate these problems, as well as to provide additional benefits by reducing memory latency. It is a shared-memory operation that adds a processor to a hardware queue of waiters for a line. QOLB allows a process to spin on a locally-cached *shadow* copy of a line. When the actual line is released by the process at the head of the queue it is automatically transferred to the cache where the next waiting process resides. The software algorithm for maintaining QOLB queues is discussed in the section on performance.

The QOLB mechanism allows for efficient process synchronization by providing a direct implementation of binary semaphores. As a non-blocking operation, QOLB can prefetch (*i.e.* make local) a line of data while a process performs useful work. Combining these two operations along with a simple software convention, QOLB becomes a *synchronizing prefetch* operation. That is, QOLB can be used to prefetch a line of data, allowing local tests to determine when the data has become available. If a line is prefetched sufficiently in advance, a process could synchronize on and obtain (make local) the shared data without experiencing interconnect delay.

The QOLB primitive was proposed independently of any architecture, and is feasible for any shared-memory multiprocessor, including ones that do not provide caches at all, and those that provide caches but do not include hardware cache coherence. QOLB is largely independent of the communication medium; however, a natural way to implement it efficiently is to extend a cache coherence protocol. QOLB provides its greatest advantages in a network that allows multiple outstanding requests and has a high latency/bandwidth product. It is currently being investigated in two contexts: (1) the Multicube topology [GoWo88] employing a broadcast-based cache coherence protocol, and (2) the Scalable Coherent Interface (SCI) [AGGW90], a proposed IEEE standard multiprocessor interface supporting point-to-point links and a non-broadcast-

[2] QOLB (Queue On Lock Bit) is pronounced "Colby". It was originally was called QOSB in [GoVW89] but was changed to be more precise.

based cache coherence protocol.

Two software algorithms inspired by QOLB have been developed. Each of these implements queues as software-maintained data structures. Anderson [Ande90] presents a scheme to implement a queue using Fetch&Add an array of flags, and a counter. Mellor-Crummey and Scott [MeSc90] present a second algorithm that implements a software queue using Compare&Swap and pointers. Although these algorithms provide queues for locks in software, they introduce substantial synchronization and data acquisition delays. Discussion and analysis of these algorithms are postponed until the section on performance.

As a final note, Fetch&Φ primitives may be implemented in a shared memory environment in two different ways. First, they may be implemented as cache operations, performed locally by locking the appropriate cache word to insure that the read-modify-write operation is performed atomically. In this case the primitive is treated as a write operation by the underlying cache coherence protocol, which responds by supplying an exclusive copy of the line containing the desired word to the local cache. Second, Fetch&Φ primitives may be implemented as remote memory operations, performed on the desired word by the controller at the appropriate memory. The first implementation will be denoted as "coherent" operations and the second as "at-memory operations. QOLB is always treated as a coherent operation.

3. Reducing Communication Latency

A shared-memory programming model provides a powerful, yet conceptually simple framework for constructing parallel applications. In addition, shared-memory hardware, such as a cache coherence protocol, typically provides a finer granularity of sharing than that of message-based systems. The finer the granularity the more processors that can be applied to a problem. Thus, shared-memory systems would seem to have a performance advantage over message-based systems as well.

Unfortunately, higher performance is difficult to achieve with shared-memory multiprocessors due to the overhead of synchronization and sharing. For example, cache coherence constraints typically require three or four traversals of the interconnect, each followed by an access to some type of memory, in order to acquire write access to shared data. Synchronization operations usually must follow the same protocol, with additional constraints to insure that data accesses occur in a sequentially consistent manner.

As an example, suppose that process B executes a simple critical section that is guarded by a lock currently held by process A. A typical "efficient" method of synchronization in a shared-memory system requires process B to spin locally on a cached copy of the lock. For reasons of efficiency it is assumed that the line containing the lock is an exclusive copy. To release mutual exclusion on the critical section, process A writes the lock. This write operation requires a minimum of three traversals of the interconnect: (1) to interrogate the directory, (2) to forward the request from the directory to the cache with the exclusive copy of the lock, and (3) to forward the line containing the lock to the process performing the write. Process B, upon losing its cached copy of the lock attempts to reacquire it by performing a write operation (*i.e.* a Test&Set on the lock. Again three traversals of the interconnect are required. Because process B is spinning on the lock, the operation from that process to interrogate the cache is issued while the lock is being forwarded to process A. This permits the accesses of process A and process B to overlap slightly, resulting in a total synchronization latency of as little as five traversals of the interconnect.

Once the lock has changed hands, process B typically attempts to acquire the shared data necessary to execute the code in the critical section. Assuming that this data fits into a single cache line, another three traversals of the interconnect are needed. Process B then completes the critical section by performing the desired computation.

This situation is depicted in the first time line of Figure 1. Each short segment is a traversal of the interconnect, and is labelled as either A or B, depending on which process issued the request which that traversal is part of. The latency for executing a critical section is divided into three parts: (1) the delay due to synchronization (acquiring the lock), (2) the delay due to accessing shared data, and (3) the time to perform the desired computation. Note that any delay waiting for the previous process to finish its critical section is not included. Eliminating or reducing such a delay can only be handled by the programmer, not the synchronization hardware.

From the above discussion it is not difficult to see why message-based systems may hold a performance advantage over shared-memory systems. First, the send/receive mechanism

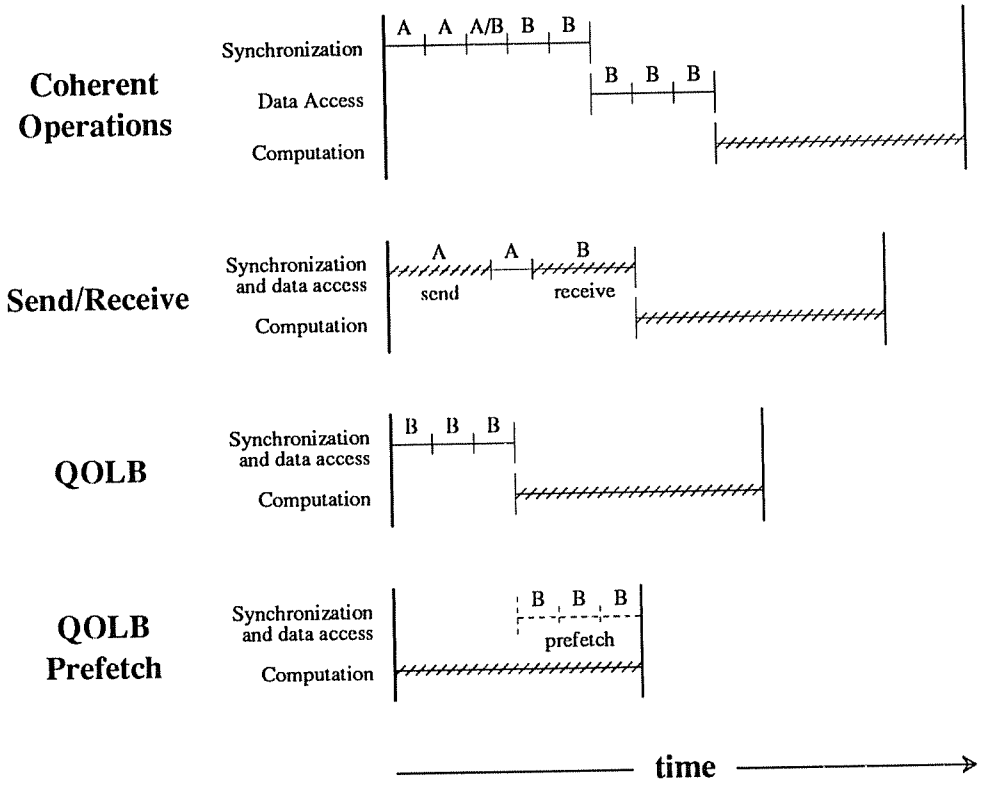

Figure 1. Access Times for Critical Sections. Shown is the time to access a critical section for different synchronization models and levels of support. Each time line is broken down into the times for synchronization (excluding waiting for the data to become available), access to shared data, and computation. Interconnect delays are shown as simple straight lines, while computational delays are depicted as lines with cross slashes.

overlaps synchronization with the acquisition of shared data. Intuitively, the send primitive supplies data as soon as it becomes available to the process that will request it next (assuming the identity of that process is known). Thus, the combined synchronization and data latency is made up of three parts: (1) the execution of the send primitive, (2) a single interconnect traversal to supply the data, and (3) the execution of the receive primitive. This section execution profile is shown as the second time line in Figure 1. Note that the middle portion of the synchronization and data access latency, the interconnect latency, could potentially be overlapped with useful computation, further improving the efficiency of message-based multiprocessors.

Unfortunately, implementations of send and receive result in primitives that require a significant amount of processor time to execute, although they can be sped up by the use of special hardware [RaSV90]. This situation is exacerbated if the operating system is invoked, which requires context switches. Thus message passing systems experience substantial latencies for process communication, much longer than those of a typical shared-memory operation.

What is needed is a mechanism with the latency of a shared-memory primitive that can acquire shared data as soon as it becomes available. Such a primitive is QOLB. QOLB overlaps synchronization (*i.e.* the acquisition of a lock) with a request for exclusive access to the data, actually eliminating the interconnect latency of following reads and writes by allowing them to occur locally once the lock is transferred to the waiting process. This overlap occurs explicitly because the shared data and the lock guarding that data can be placed in the same line when QOLB is used along with Test&Set. Thus, the total latency is that for a single remote access, namely, three interconnect traversals. This situation is depicted by the third time line of Figure 1.

Because QOLB is a non-blocking synchronizing operation, it can be used to prefetch a line containing a lock and shared data. A line may be prefetched as soon as a process knows what data it needs, and that the lock guarding the data has been acquired by the process producing the needed data. If prefetching is performed sufficiently in advance then the latency of the first two traversals used to locate the data may be avoided. If the data itself becomes available soon enough, then the entire communication latency, both synchronization and data acquisition, may be overlapped with useful computation. This situation is shown as the last time line of Figure 1.

The relative performance of QOLB to send/receive depends primarily on two factors. First, the type and efficiency of support that a message-passing system provides can have a dramatic effect on the time required by a processor to execute the send and receive primitives. Second, the structure of the application must allow prefetch QOLB operations to be performed early enough to overlap the computation of earlier critical sections.

Figure 1 is useful for a final observation. If the granularity of the critical section computations was sufficiently coarse to allow a balance of latencies for the case of typical cache coherent synchronization, then obviously using QOLB and prefetching will support a finer granularity to achieve that same balance. Thus, the time to complete a unit of work may be substantially reduced by increasing the number of processors. A finer granularity can be used to split the work of the critical section into a number of smaller critical sections which can then be distributed among the processors.

4. Performance

In evaluating hardware memory primitives for large-scale shared-memory multiprocessors one must consider those scenarios that require efficient synchronization and data sharing among

large numbers of processes. The synchronization scenarios chosen for this study consist of locks (especially for pairwise sharing), counters, barriers, and global event notification. These scenarios are the same as those proposed by Goodman, Vernon, and Woest [GoVW89] with one exception. The work queue scenario has been replaced by incrementing a counter. The provision of fast counters represents the crux of the problem of implementing highly parallel queues, and is much more general. We believe the above scenarios represent, or are typical of, the most critical problems in providing efficient support for parallel applications.

First analytical and simulation methods used to perform the evaluations are discussed. Then the synchronization scenarios are presented, each followed by the description of several algorithms either constructed by the authors or chosen from previous studies, and a discussion of their performance.

4.1. Performance Methods

Both analytical techniques and simulation were used to provide performance results for the chosen synchronization scenarios. Specifically, simple counting techniques were used for evaluating lock algorithms, and verified using a simulator. Simulation was used to analyze counter and barrier methods. The advantage of using simulation for these scenarios are two-fold. First, a simulator can be used to test the correctness of algorithms, as well as assist in pointing out where algorithms may be modified to improve performance. It is especially important to realize that synchronization algorithms contain complex interactions that may be overlooked when attempting to estimate performance using analytical techniques. Second, an appropriate simulator can evaluate arbitrary synchronization algorithms without major modifications.

Simulation results were obtained using a simulator that models the execution of a single synchronization algorithm by a set of processors. An algorithm is represented by a stored program consisting of simple arithmetic operations, program control, and memory operations. The primary requirements of the simulator are that it is capable (1) of evaluating arbitrary synchronization algorithms and (2) of accurately modeling the delays associated with accesses to shared variables.

To assist in the analysis, a model of a shared-memory cache-coherent multiprocessor was constructed. This model is very similar to the the Multicube topology [GoWo88], especially in its implementation of the interconnect, with a directory-based cache coherence protocol. The model consists of a multidimensional grid of buses (the global interconnect), with processing nodes located at the intersections. A node consists of a processor, a two-level cache hierarchy, and a portion of main memory. For purposes of this study each processor executes a single process. It should be made clear that the term "cache" in the context of this paper refers to the second-level cache. Processors are allowed multiple outstanding requests, but only the software combining counter algorithm actually makes use of this feature. The lock, barrier, and other counter algorithms do not require such a capability. For this study, the processors must wait for acknowledgements for write operations before proceeding.[3]

[3] The QOLB primitive is complete in the sense that no other hardware support is needed to assure correct synchronization, *i.e.*, even fences [BrMW85] are unnecessary. Sequential consistency [Lamp79] deals with the correct observable ordering of memory accesses. Several models of "weaker" forms of consistency have been developed, for example [DuSB86, Good89, AdHi90, GLLG90]. Relaxing the restrictions on sequential consistency using QOLB is the subject of a separate study [WoGo91].

A number of simplifying assumptions were made in the model in order to allow the analysis of competing algorithms to focus on shared variable latency issues. These assumption are that (1) caches are infinite in size, (2) processors are infinitely fast, and (3) message queues are "large".

Infinite caches ensure that misses will occur only for shared data that has been invalidated due to a request from another cache.[4] However, since none of the synchronization algorithms require much data, either private or shared, this should cause little perturbation in the results. Likewise, infinitely fast processors (and infinitely fast first-level caches) need have little effect on the results, since processors can be arbitrarily delayed at any point in there execution of an algorithm. In addition the second level cache will only allow non-local memory requests to be issued, or replies to be received, at a specifiable rate. Finally, the most significant impact of large queues is likely to be the reduction of hot-spot effects by mitigating tree saturation effects[5]. Fortunately, few algorithms in this study exhibit hot spots, and the ones mostly likely to benefit from large queues are those that already perform poorly in comparison to other algorithms.

At-memory requests always bypass the cache and are performed on a single word at the appropriate memory. Coherent line requests are routed to the appropriate memory and either obtain the line from the memory or are forwarded to the cache that currently has the exclusive copy of the line. At-memory and coherent line operations are never performed on the same cache line. In fact, they are generally not used in the same algorithm, except for some of the lock examples where the choice of synchronization method involves at-memory primitives and the shared data is exchanged using coherent line read and write operations.

The service times for the system we envision are based on an assumption of approximately 10 ns per cycle, but some scaling of these numbers either direction is appropriate. The following specific timings were used. A memory read requires 5 cycles, although updating the directory or the line itself (usually the case) requires 10 cycles. Directory and cache memory have the same characteristics as those of main memory.

Bus arbitration and synchronization requires 4 cycles. Requests may be address only (2 cycles), address plus a line of data (8 cycles), or address plus a word for at-memory operations (generally 3 cycles). These are in addition to the initial 4 cycles for arbitration. A request can be forwarded as soon as the address portion is transmitted (6 cycles total). Reliability constraints dictate that an entire request be received and error correction performed (2 additional cycles) before service can be completed. Buses service requests in a round robin fashion.

4.2. Locks (Critical Sections)

Locks are used to restrict access to shared data to a single process at a time, so that reading and writing of the data may occur without interference from competing accesses. This type of synchronization is called a critical section. If the critical section is empty, then the lock is basically being used as a timing mechanism. Such a situation may exist, for example, in barrier synchronization where one process may wish to let another know that it has completed its appointed computation.

---

[4]Cold start misses are also ignored.

[5] See [PfNo85] for a discussion of hot-spot contention and tree saturation.

Two situations are common. These correspond to the two states which a lock may be found in, namely, idle or busy. Idle locks are typical of situations where locks are precautionary, that is, where contention is possible but infrequent. Busy locks are commonly found in many situations, especially those involving pairwise sharing, whenever one process is waiting for data being written by another. However, if a lock is released before it is requested then the lock is considered to be idle.

It should be pointed out that serialized access to data implies that, in a large-scale shared-memory multiprocessor, high contention for a lock will cause a severe bottleneck. In addition, with high contention, starvation of some processes may be much more likely, depending on the specific mechanism used for granting the lock. While parallel applications may use locks which change hands frequently, these algorithms are unlikely to scale as the number of contending processes increases. Thus, the behavior of locking mechanisms under conditions of high contention does not appear to be a good indicator of performance. In addition, since most of the mechanisms analyzed are "contentionless", a simple analysis of the time required for a (busy) lock to change hands is likely to provide at least a lower bound on a situation where there is high contention.

Results are provided for three algorithms: spinlocks, QOLB locks, and a software queue-based locking mechanism (MCS locks [MeSc90]). Both idle and busy lock performance are studied. Locks are compared for two types of computation, the first where the critical section is empty, and the second where the critical section performs some simple function, for example, incrementing a counter. The null critical section provides an estimate of the overhead incurred by the lock mechanism itself. The counter critical section is more indicative of true performance since, realistically, a critical section will generally contain at least one read and one write to shared data. For the non-null critical section, coherent line primitives are always used to read and write the line.

Critical sections requiring access to multiple lines of data are not specifically considered. Given a system which supports a model of consistency that allows multiple outstanding requests from a single processor, techniques are possible which allow the acquisition of multiple lines of data to be overlapped with the latency for acquiring the first line. Details concerning appropriate hardware and software (*e.g.* lockup free caches) is considered beyond the scope of this study.

Lock Algorithms

The first lock algorithm is a simple spinlock using Test&Set and Unset. For coherent line primitives, on each lock access the corresponding line is first transferred to the cache of the requesting process before being accessed. For at-memory primitives, these operations are performed at the memory where the word normally resides. A simple spinlock is shown as Algorithm 1.

```
while(test&set(lock))
     /* wait */
"critical work"
unset(lock)
```

Algorithm 1. Spinlocks.

QOLB locks are very similar to Test&Set spinlocks with the exception that a QOLB lock is issued before the spin loop. The QOLB operation allocates a hardware queue entry in the local cache which allows the process to spin locally using Test&Set. Through the hardware queuing

mechanism the desired line is eventually written into the cache when the lock is free and the process is at the head of the queue. After the line becomes local the next Test&Set operation will succeed.

Since a QOLB request is for an entire line, transfer of the first line of data is overlapped with the acquisition of the lock. Also, since QOLB is non-blocking, multiple locks can be acquired simultaneously and/or computation can be performed before the process actually attempts to set the lock. Both of these possibilities point to the use of QOLB as a synchronizing prefetch operation. The acquisition of locks, as well as shared data, can be overlapped with useful work. Thus, it is possible to reduce the latency of a critical section to the processor time for executing the critical section code. The QOLB lock scheme is shown as Algorithm 2. An extended discussion is given elsewhere [GoVW89].

```
procedure acquire(lock)
begin
     QOLB(lock)
     while(test&set(lock))
          QOLB(lock)
end

acquire(lock)
"critical work"
unset(lock)
```

Algorithm 2. QOLB Locks.

The MCS algorithm [MeSc90] implements queue-based locks in software, rather than in hardware, as QOLB does. It improves on Anderson's lock algorithm [Ande90] by forcing all spinning to be performed locally. An atomic Swap operation and an atomic Compare&Swap operation are provided, along with both Fetch and Store, as at-memory primitives. The Compare&Swap operation is not strictly required, however replacing it by other at-memory operations would only serve to increase the latency of the algorithm. The MCS lock scheme is described briefly below and is shown as Algorithm 3.

In the MCS algorithm a process adds itself to a queue of waiting processes, where each queue entry contains a pointer to another queue entry and a boolean flag. This is accomplished by swapping a pointer to the process's locally allocated queue entry with the tail pointer of the queue. The value returned from the Swap operation is a pointer to the queue entry of the process's predecessor in the queue. If there is no predecessor the process has acquired the lock. Otherwise, the process writes a pointer to itself into the predecessor's queue entry and waits on its own local flag. The write informs the predecessor as to which process to unblock when it releases the lock.

The release operation simply requires that a process unblock its successor by writing the successor's local flag. If there is no known successor the process performs a Compare&Swap operation to guarantee that the queue is empty. If another process (successor) has recently joined the queue then the process holding the lock must wait until the pointer in its queue entry is written with the identity of its successor in order to perform the unblock operation.

```
struct entry                    /* queue entry  */
     integer next               /* next process */
     boolean locked             /* local flag   */
end

shared integer lockid           /* queue tail   */
shared entry lock[nprocs]       /* lock entries */
integer pred                    /* predecessor  */
integer succ                    /* successor    */
integer myid                    /* my process id*/

  /* allocate lock[x] local to process x */

lock[myid].next = NIL
pred = swap(lockid,myid)
if (pred != NIL)
     lock[myid].locked = 1
     lock[pred].next = myid
     while (lock[myid].locked)
          /* wait */

"critical work"

if (lock[myid].next == NIL)
     if (!compare&swap(lockid,myid,NIL))
          while (lock[myid].next == NIL)
               /* wait */
          succ = lock[myid].next
          lock[succ].locked = 0
else
     succ = lock[myid].next
     lock[succ].locked = 0
```

Algorithm 3. MCS Locks.

Lock Performance

Table 1 summarizes the performance of the various lock algorithms for a two-dimensional, unloaded system. A coherent line operation requires 59 cycles in which to to interrogate the directory, find the exclusive copy of the line, and move that copy to the requesting processor's cache. Since three interconnect traversals and three memory access are involved, this indicates that about 20 cycles are required per traversal/memory access pair.

A spinlock requires exactly one coherent operation (*i.e.* a Test&Set operation) to acquire an idle lock, and two such operations if a line of data is needed as well. If the lock is busy then additional time is required to release the lock which, according to Figure 1, is about 5 traversals total, or 100 cycles. This agrees very well with the analytical results.

Lock Algorithm	Type of Primitive	Critical Section	Latency (Cycles) Idle	Busy
Spinlock	coherent	null	59	103
		counter	118	162
Spinlock	at-memory	null	62	48
		counter	121	107
MCS lock	at-memory	null	79	57
		counter	138	116
QOLB lock	coherent	null/counter	59	25
		prefetched	0	25

Table 1. Idealized Critical Section Times for Locks Mechanisms. Given is the latency in bus cycles of various lock mechanisms (spinlocks, MCS locks, QOLB locks) for two types of critical sections (null, counter) and two types of memory primitives (coherent, at-memory).

Spinlocks can be made more efficient for acquiring busy locks by using at-memory operations instead of coherent operations to synchronize. At-memory operations require only about 15 cycles per traversal, since a line of data need not be transferred, and do not cause a ping-pong effect when a lock is released. Thus a spinlock using at-memory primitives requires two traversals to set an idle lock and two to release it (about 60 cycles total). Likewise it can be shown that on average about three traversals are needed if the lock is busy (about 45 cycles), although the timing is more complicated. The exact numbers, verified by the simulator, agree well with these estimates. Again, requiring a line of data using a coherent write operation adds another 59 cycles to the latency.

The latency of the MCS lock algorithm is just slightly longer than that for spinlocks, which is unexpected given that it requires more memory accesses. The key is that many of these accesses are to variables allocated in local main memory (not cache). While MCS locks have slightly higher latency, it must be remembered that they cause no spinning over the interconnect. In a real system this translates into less bandwidth consumption and the elimination of the synchronization hot spot that spinlocks cause when more than a single process waits on the same lock. Thus, MCS locks can be used when spinlocks are inappropriate.

The fastest mechanism, however, is the QOLB algorithm. For busy locks, that is for cases where the QOLB queue is set up while the critical section is in use, only a single traversal of the interconnect is necessary to transfer the lock to the next process in the queue once the lock is released. This is the same efficiency as send/receive, which also requires only a single traversal. However, QOLB still has lower latency, since shared-memory operations do not require as much processing. The result is a factor of four speedup over other shared-memory algorithms, for a non-null critical section.

For idle locks QOLB requires a single coherent operation. This results in a factor of two speedup over any of the other schemes. However, if the lock can be prefetched, that is, if the process knows in advance that the lock is needed, the QOLB operation can completely overlap

useful computation, and perform the critical section with zero synchronization and data sharing overhead. With prefetching of a busy lock QOLB still requires a single traversal of the interconnect. It is exactly in these prefetching situations where QOLB can improve performance the most.

4.3. Counters

Unlike general locks, incrementing a counter is an example of a critical section where high contention must be handled efficiently. Counters have a number of uses, from obtaining a unique number, to determining which loop iteration to perform, to obtaining an index into a work queue. Algorithms for barriers generally are not appropriate for counters. Although a barrier essentially counts the arrival of processes, counter applications do not necessarily assume that all processes will participate, or at least not necessarily within a reasonable time frame.

There are four major classes of hardware/software support for counters: (1) serial, (2) pipelined (serial only at memory), (3) software combining, and (4) hardware combining. Assuming that the interconnect involves $O(\log n)$ latency for a request to traverse it, the time required for n processes to acquire a unique number can be computed for each class. For serial techniques $O(n \log n)$ time is required since each request requires a traversal of the interconnect. For pipelined techniques, memory may complete an increment operation every t cycles for a total latency of $O(n)$. Using software combining, a tree requires $O(\log n)$ operations, each requiring a memory request for a total latency of $O(\log^2 n)$. Finally, hardware combining requires time proportional to a single request over the interconnect. Thus, its time is $O(\log n)$. Below, examples of each of the first three classes of counters are examined. The performance of hardware combining is represented by the most ideal situation, which for this model is the completion of an increment operation for every process every 32 cycles.

Of all of the serial algorithms studied in the lock section, QOLB locks performed the best. The fastest pipelined algorithm is a Fetch&Increment operation performed at memory. The choice of a software combining method is that developed in a previous work [GoVW89]. Since the QOLB algorithm has already been described, and the lock algorithm is a simple Fetch&Increment instruction, we will turn to the software combining algorithm.

The software combining algorithm, simplified to use a binary tree, is shown as Algorithm 4. The algorithm consists of four parts. In part one a process proceeds up the tree acquiring nodes, marking them as visited, and unlocking them. The process stops when it either reaches the root or finds a node already visited. In part two the process reacquires the nodes it has visited, combining any results from other processes participating in incrementing the counter. These nodes are then left locked so further combining is prohibited. In part three, the process performs the increment if it has reached the root. Otherwise, the process places its combined increment in the topmost node it has reached, unlocks that node so that it may be combined in higher stages of the tree, and waits for a result. In part four, the process distributes the result to all nodes that it had performed the combining for. New combining is now allowed to continue in a following wave. The interested reader is directed elsewhere [GoVW89] for a more detailed description.

```
/* PART 1: Mark free nodes going up tree */

     value = 1
     i = leaf(pid)
     loop
          acquire(tree[i].lock)
          while (tree[i].state == RESULT)  /* wait */
               unlock(tree[i].lock)
               acquire(tree[i].lock)
          if (tree[i].state != FREE)
               break
          tree[i].state = MARKED
          unset(tree[i].lock)
          i = up(pid,i)
     end
     highest = i

/* PART 2: Reacquire nodes and combine increments */

     i = leaf(pid)
     while (i != highest)      /* prefetch first */
          qolb(tree[i].lock)
          i = up(pid,i)
     i = leaf(pid)
     while (i != highest)
          acquire(tree[i].lock)
          if (tree[i].state == COMBINE)
               value = value + tree[i].value
               tree[i] = value - tree[i].value
          i = up(pid,i)

/* PART 3: Enter increment and wait for result */

     if (tree[i].state == ROOT)
          tree[i].value = tree[i].value + value
          v = tree[i].value - value
     else
          tree[i].value = value
          while (tree[i].state == COMBINE) /* wait */
               unset(tree[i].lock)
               acquire(tree[i].lock)
          value = tree[i].value
          tree[i].state = FREE
     unset(tree[i].lock)

/* PART 4: Distribute results back down tree */

     while (i != leaf(pid))
          i = down(pid,i)
          if (tree[i].state == COMBINE)
               tree[i].state = RESULT
               tree[i].value = tree[i].value + value
          else
               tree[i].state = FREE
          unset(tree[i].lock)
```

Algorithm 4. Software Combining of Increment Operations.

Figure 2 shows the latency of the counter algorithms, for a fixed size system, as the number of participating processes increase. In this graph the curve for hardware combining is indistinguishable from the baseline. The performance for the algorithms is as one would predict. To obtain a better view of the relative performance of the algorithms for fewer numbers of processes, the latency may be divided by the number of competing processes to yield the time per between increment operations. From this second graph some additional performance details become clear. Even with a small constant for the pipelined Fetch&Increment operation and a rather large constant for software combining, the crossover point between the two schemes is at only 64 processes. However, there are two factors that may decrease the performance of software combining. First the degree of combining is artificially high in that all processes are participating all of the time. Second, no computational delays were included in the simulation which, although small in comparison to memory and interconnect latencies, may slow down the algorithm a bit. These factors would tend to move the curve for software combining farther to the right in the second graph.

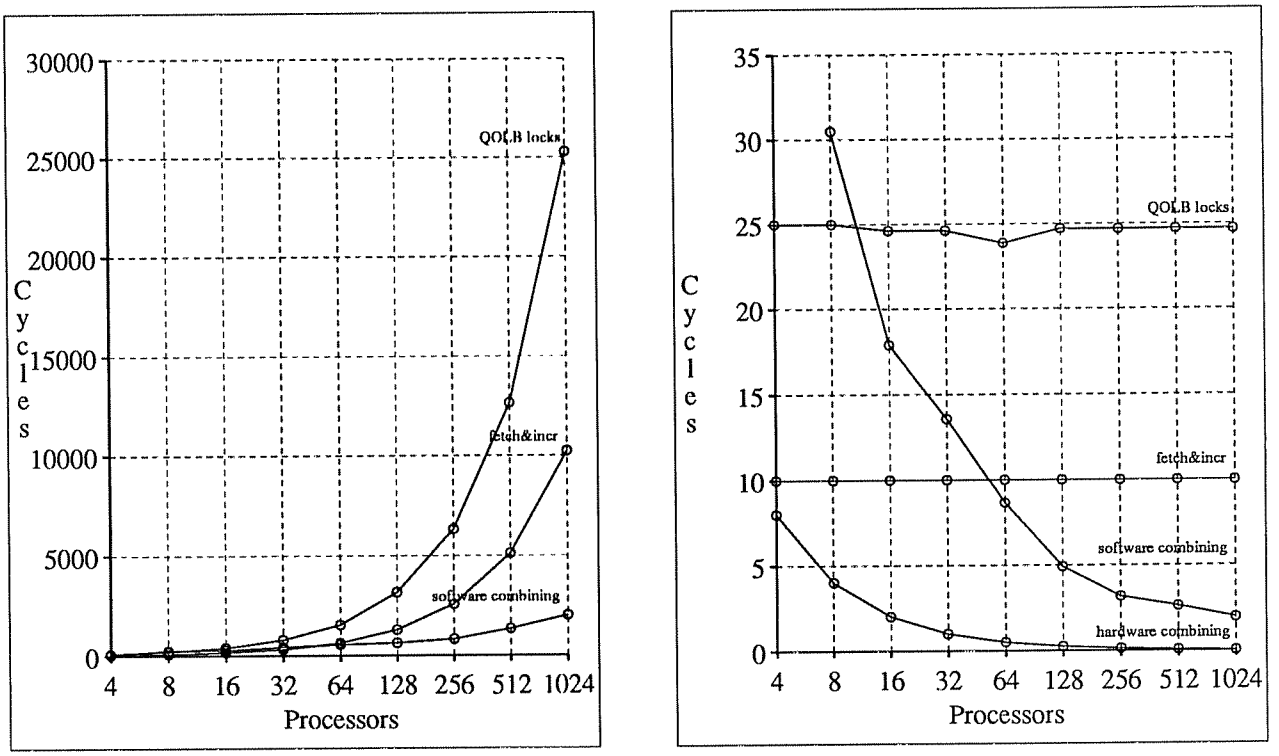

Figure 2. Time per Increment Operation: System and Process Views. Shown on the left is the average latency for an increment operation to complete. On the right is the same information, but with latency divided by the number of competing processors. This yields the time between increment operations, as seen by the system, and gives a better view of the relative performance of the algorithms for fewer numbers of processes. The system size is 1024 processors, with the actual number of competing processors given by the x-axis.

4.4. Barrier Synchronization

Barrier synchronization is a mechanism which guarantees that all processes have reached a specific point in their execution before any are allowed to proceed. It is used to synchronize the exchange of shared data so that updates to that data have completed before the next phase of computation. A typical case is synchronizing between iterations of a loop.

A barrier consists of two functions: (1) insuring that all processes have reached the barrier and (2) notifying all of the processes once that point is reached. Since barriers need to handle large numbers of processes efficiently, tree-based combining algorithms have been proposed to allow processes to declare themselves as having arrived. Such an algorithm, employing Fetch&Increment and a tree of counters, was analyzed by Yew, Tzeng, and Lawrie [YeTL87].

The second part, notifying processes that the barrier has been reached, is a useful technique in any instance requiring notification of a global event, such as in majority (or k out of n) decisions, or for indicating the successful termination of a parallel search. Three methods of notification are studied: (1) a simple write to a global flag, (2) a broadcast write to a global flag, and (3) a tree-based method of distributing the notification. The difference between a simple write and a broadcast write is that the former causes an invalidation with subsequent traffic to re-read the flag, while the latter does not.

In this section, the performance of several barrier algorithms are analyzed in terms of latency for large numbers of processes. The relative performance of the three notification schemes are also analyzed. Finally, estimates of the advantage of locality optimizations are presented.

Barrier Algorithms

The first algorithm is a combination of the fastest serial counter, Fetch&Increment, with the fastest method of notification, NOTIFY. The last process to increment the shared counter performs a broadcast write allowing all processes to proceed past the barrier. The sense of the the notification flag is reversed each iteration in order to speed up execution. This algorithm is shown as Algorithm 5.

```
if (fetch&incr(count) == n)
     count = 1
     NOTIFY(flag,sense)
else
     while (flag != sense)
          /* wait */
sense = ! sense
```

Algorithm 5. Serial Barrier Using Fetch&Increment.

While it is natural to separate the two functions of counting arrivals and global notification, it is also possible to combine them, as in the case of the butterfly barrier [Broo86]. This algorithm requires a process to perform $\log n$ pairwise synchronizations, the pairs being chosen in such a way that a process can complete all of the synchronizations only when all processes have reached the barrier. An improved algorithm, the dissemination barrier, has also been developed and studied [HeFM88]. This algorithm reduces the cost of synchronizations by making them one-way and employing a different pattern of processor-to-memory writes. Two copies of the synchronization variables are used to ensure that there is no interference between consecutively

executed barriers. The dissemination barrier is given as Algorithm 6.

Restricting the number of processors per bus to a power of two, an appropriate partner function for this study is given by $partner(i,k)=i \oplus 2^k$. If processes are assigned to processors in a consecutive manner then each synchronization traverses only a single stage of the interconnect, independent of the number of stages, potentially reducing the latency of the barrier by a significant factor.

```
i = 0
while (i < log(n))
     p = partner(i,pid)
     flag[set,i,p] = sense
     while (flag[set,i,pid] != sense)
          /* wait */
     i = i + 1
set = 1 - set
if (set == 0)
     sense = ! sense
```

Algorithm 6. Dissemination (Butterfly) Barrier.

Hensgen, Finkel, and Manber propose a second algorithm called a tournament barrier [HeFM88]. In this scheme processes race up a binary tree. Since the winner is statically determined at each level, the loser simply writes a flag to let the winner know that it has reached the barrier. The winner at the root node initiates the notification process, which is passed down the tree until it reaches the leaves.

The tournament algorithm presented here is very similar. Each process waits on a single shared variable that is initialized to the total number of winning rounds for that process. Losing processes simply decrement the appropriate variable by one, allowing a winner to stop spinning when its shared variable becomes zero. The reason for the different algorithm is in the choice of an appropriate parent function which is given by $parent(i)=i-2^{\log n-k-1}$. If processes are assigned to processors in a consecutive manner then each synchronization traverses only a single stage of the interconnect. Wakeup may be performed using any of the notification schemes described earlier. The tournament barrier using a separate tree of flags to perform the wakeup operation is shown as Algorithm 7. This algorithm can be easily modified to use a broadcast NOTIFY operation along the lines of the Fetch&Increment algorithm.

Performance results for a tournament barrier modified to use QOLB and NOTIFY are also given. This algorithm is quite naive, employing no prefetching techniques. The purpose of its inclusion is simply to show barrier performance for an algorithm using coherent line primitives. Prefetching may be added to any of the Fetch&Φ based algorithms by using multiple copies of the barrier structure and appropriately locking and prefetching synchronization variables in advance. In the case of tournament and dissemination barriers, QOLB requires the *same* traversal of the interconnect for each variable as does the Fetch&Φ based algorithms. Thus, it should be clear that QOLB prefetching will yield essentially the same level of performance. QOLB has no significant advantage in performance since no data is associated with the lock.

```
if (level(pid) != LEAF)
     while (tree[pid] != sense)
          /* wait */

if (level(pid) != ROOT)
     p = parent(pid)
     tree[p] = sense

p = right_child(pid)
if (p != nil)
     flag[p] = sense
p = left_child(pid)
if (p != nil)
     flag[p] = sense

sense = ! sense
```

Algorithm 7. Tournament Barrier with Tree Wakeup.

Barrier Performance

Figure 3 shows the average time to complete a barrier for each of the various algorithms. The 8 processor case uses a single bus, while each 8-fold increase in the number of processes requires an additional stage in the interconnect. Each process is delayed 180 to 220 cycles between barriers according to a uniform distribution. This delay randomizes the arrival of processes to the barrier. In addition, processes are assigned randomly to processors. Although a process is allowed to spin locally on a wakeup flag, non-local accesses to synchronization variables require, on average, latency proportional to the number of stages in the interconnect.

The first graph demonstrates that using schemes that create hot-spots, such as Fetch&Increment yields performance which is unacceptable beyond a small number of processors. The second graph gives a more detailed view of the performance of the lower latency barrier algorithms. All of the plots in Figure 3 curve upwards since the latency of a barrier is at least $O(\log^2 n)$, one factor of $\log n$ for the number of synchronization levels in the barrier, and the other for the latency of the interconnect. The tournament barrier, using a binary arrival tree and NOTIFY, has the lowest latency, closely followed by the dissemination barrier. The difference is due to the larger number of synchronizations required by the dissemination barrier as compared to the the tournament barrier. The tournament barrier using tree notification and the QOLB barrier using NOTIFY have roughly twice the expected latency of the other two schemes.

These performance results have several implications for global event notification. First, read-sharing[6] generally causes unacceptable performance, such as in the case of a write invalidation of a globally shared wakeup flag. Furthermore, the use of the NOTIFY primitive is only acceptable if the initial traffic to read the shared notification variable is amortized over a large number of iterations of the barrier. This number is on the order of 1000 cycles for 4K

[6] Read-sharing refers to the situation where a large number of processors simultaneously attempt to read a shared variable [KMRS88].

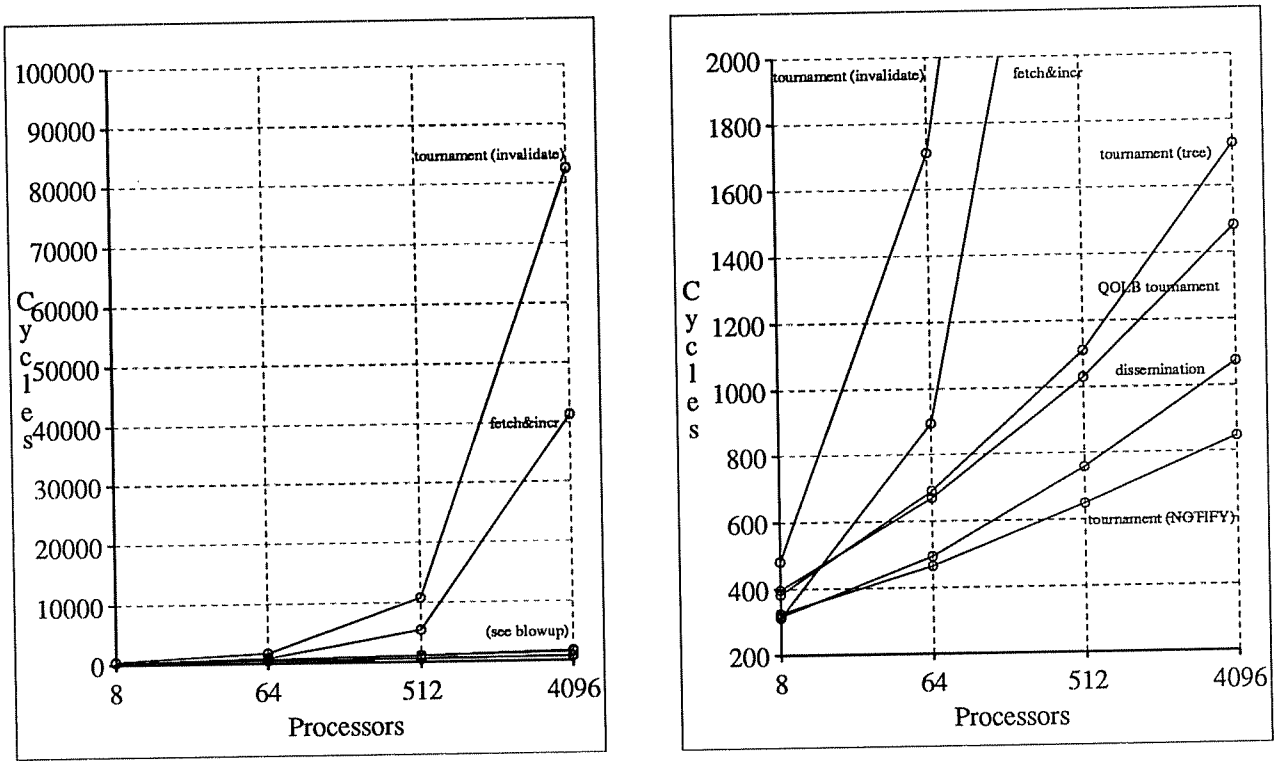

Figure 3. Performance of Barrier Algorithms. Shown is the average time to complete a barrier. Each process is delayed 180 to 220 cycles between barriers according to a uniform distribution. The 8 processor case uses a single bus, with another stage (dimension) added to the interconnect for each 8-fold increase in the number of processors. The graph on the right is a blowup of that on the left detailing the performance of the lower latency algorithms. Processes are assigned randomly to processors.

processors. An alternative is to provide hardware support for combining read-sharing traffic. Thus, notification through a tree of shared variables may perform better over a wide range of situations, even though its latency is approximately twice as long. The best alternative may be to use a dissemination barrier, which provides almost the speed of a tournament barrier using NOTIFY, but does not incur the read-sharing overhead.

Barrier performance can generally be improved if processes are assigned to processors in a specific order and there is sufficient symmetry in the interconnect. For the Multicube architecture and the previously described *partner* and *parent* functions of the dissemination and tournament barriers, the latency of the interconnect is substantially reduced. Synchronization operations are required to traverse only a single stage of the interconnect, independent of the number of stages. This results in $O(\log n)$ latency for the two barrier algorithms, as demonstrated by the straight curves for the optimized algorithms in Figure 4. The locality optimizations are more important for the dissemination barrier, which requires $O(\log n)$ times the traffic of the tournament barrier, and thus experiences more queuing delays in the interconnect. The savings due to the locality optimizations are approximately 36 and 26 percent for the dissemination and tournament barriers, respectively, for 4K processors.

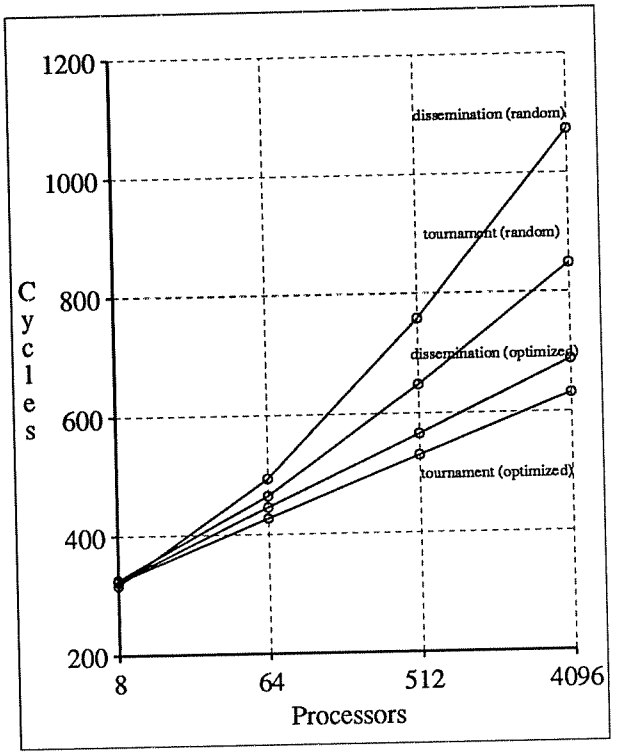

Figure 4. Effect of Random Placement of Processes on Barrier Performance Shown is the average time to complete tournament and dissemination barriers for both optimized and random assignments of processes to processors. For both sets of curves spinning is performed locally. Parameters are the same as in Figure 3.

5. Conclusions and Future Work

The QOLB hardware provides a very efficient mechanism for both process synchronization and data sharing. First, QOLB directly implements the queuing mechanism of binary semaphores, making a single non-local operation sufficient to acquire a lock, independent of whether the critical section is idle or busy. Second, by placing locks with data, both can be acquired in a single access. This technique effectively cuts in half the latency required to enter a critical section and acquire the first line of data. Specifically, from Table 1 it is clear that QOLB reduces the latency to enter a critical section and start computation by a factor of four over software schemes when a lock (*i.e.* critical section) is busy. For idle locks the improvement due to using QOLB is only a factor of two, which is due to the additional latency of software algorithms to acquire shared data.

However, QOLB has an additional advantage in the case of idle locks. Since QOLB is both synchronizing and non-blocking, it can be used to prefetch shared data. Thus communication latency can effectively be reduced to zero if the shared data is prefetched sufficiently in advance to allow the latency of the QOLB operation to overlap useful computation. Through smaller communication latency, finer grained parallelism can be supported and, thus, applications can effectively use more processors than otherwise possible. Even if Fetch&Φ primitives could be issued as non-blocking operations, it would be much more difficult to structure the computation

of an application to issue the multiple steps of an appropriate prefetching algorithm correctly.

The study of counter algorithms demonstrates the effectiveness of the software combining algorithm presented elsewhere [GoVW89]. Even with only 64 participating processes software combining performs as well as a pipelined Fetch&Increment operation. Thus, it holds much promise for implementing efficient counters without resorting to hardware combining.

The barrier synchronization study demonstrates that software algorithms can be constructed that experience a logarithmic number of interconnect latencies. With sufficient interconnect symmetry it is even possible to make the traversal time of the interconnect relatively independent of the size of the system. Overall, the dissemination barrier seems to perform the best over a wide range of assumptions. QOLB does not show a performance improvement over software algorithms since at-memory Fetch&Φ operations are particularly efficient for the dataless, pairwise synchronizations of barriers. In general, at-memory operations seem to be more effective than their cache coherent counterparts for all types synchronization.

The barrier studies also yield insight into methods for performing global event notification. It is clear that efficient (*i.e.* combinable) read sharing is a necessity for broadcast invalidation protocols. Even algorithms employing a broadcast write operation like NOTIFY may suffer from hot-spot contention when the line is initially distributed to all of the participating processes. Thus, tree algorithms may be necessary. For cases other than barriers, event notification of an arbitrary subset of processes requires software combining, such as that demonstrated for counters.

QOLB's performance advantage extends to systems that enforce weaker forms of consistency. These machines allow the issue of multiple outstanding memory operations from a single processor. However, synchronization operations are still required to enforce various logical constraints on the order of operations, and shared data cannot be fetched before such constraints are satisfied. Thus, QOLB can be used to reduce these communication latencies where other memory operations cannot.

The benefit of relaxed consistency constraints is that, once synchronization is performed, multiple accesses to shared data can be handled concurrently. This is true of systems using QOLB as well. However, the QOLB hardware can be augmented to provides a framework for enforcing an even weaker form of consistency, *QOLB consistency*, where the consistency is restricted only to the variables for which there is a data race. It is expected that implementations of QOLB consistency, if not faster than other forms of weak consistency, will at least be easier to implement. This work is part of an on-going study.

6. Acknowledgements

We wish to acknowledge Mary Vernon and Guri Sohi, as well as the referees, for their comments on an earlier version of this paper. This work was supported by NSF grant CCR-892766.

7. References

[AGGW90] Aboulenein, N. M., S. Gjessing, J. R. Goodman, and P. J. Woest, "Hardware Support for Synchronization in the Scalable Coherent Interface (SCI)," University of Wisconsin-Madison, Computer Science Technical Report #984, December 1990.

[AdHi90] Adve, S. V., and M. D. Hill, "Weak Ordering - A New Definition," *Proceedings of the 17th Annual International Symposium on Computer Architecture,* May 1990, pp. 2-14.

[Ande90] Anderson, T. E., "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems,* January 1990, pp. 6-16.

[BrMW85] Brantley, W. C., K. P. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *Proceedings of the 1985 International Conference on Parallel Processing,* August 1985, pp 782-789.

[Broo86] Brooks, E. D., "The Butterfly Barrier," *International Journal of Parallel Programming,* August 1986, pp 295-307.

[DuSB86] Dubois, M., C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessor," *Proceedings of the 13th Annual International Symposium on Computer Architecture,* June 1986, pp. 434-442.

[GLLG90] Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture,* May 1990, pp. 15-26.

[GoVW89] Goodman, J. R., M. K. Vernon, and P. J. Woest, "A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III),* April 1989, pp. 64-75.

[Good89] Goodman, J. R., "Cache Consistency and Sequential Consistency," Scalable Coherent Interface (SCI), IEEE P1596 Standards Group, Technical Report #61, March 1989.

[GoWo88] Goodman, J. R., and P. J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *Proceedings of the 15th Annual International Symposium on Computer Architecture,* June 1988, pp. 422-431.

[GoLR83] Gottlieb, A., B. D. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM Transactions on Programming Languages and Systems,* April 1983, pp. 164-189.

[GGKM83] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, And M. Snir, "The NYU Ultracomputer -- Designing an MIMD, Shared Memory Parallel Machine," *IEEE Transactions on Computers,* February 1983, pp. 175-189.

[HeFM88] Hensgen, D., R. Finkel, and U. Manber, "Two Algorithms for Barrier Synchronization," *International Journal of Parallel Programming,* 1988, pp. 1-17.

[KMRS88] Karlin, A. R., M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive Snoopy Caching," *Algorithmica,* **3**, 1988, pp. 79-119.

[Lamp79] Lamport, L., "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers,* September 1979, pp. 690-691.

[MeSc90] Mellor-Crummey, J. M., and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," University of Rochester, Computer Science Technical Report #342, April 1990.

[PfNo85] Pfister, G. A., and V. A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *Proceedings of the 1985 International Conference on Parallel Processing,* August 1985, pp. 790-797.

[RaSV90] Ramachandran, U., M. Solomon, and M. K. Vernon, "Hardware Support for Interprocess Communication," *IEEE Transactions on Parallel and Distributed Systems,* June 1990, pp. 318-329.

[RuSe84] Rudolph, L., and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proceedings of the 11th Annual International Symposium on Computer Architecture,* June 1984, pp. 340-347.

[Smit81] Smith, B. J., "Architecture and Applications of the HEP Multiprocessor Computer System" *SPIE Vol. 298 Real-Time Signal Processing IV (1981),* pp. 241-248. Also appears in *Supercomputers: Design and Applications*, IEEE Catalog Number EHO219-6.

[WoGo91] Woest, P. J., and J. R. Goodman, "QOLB Consistency," *in preparation.*

[YeTL87] Yew, P. C., N. F. Tzeng, and D. H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers,* April 1987, pp 388-395.

[ZhYe87] Zhu, C. Q., and P. C. Yew, "A Scheme to Enforce Data Dependence on Large Multiprocessor Systems," *IEEE Transactions on Software Engineering,* June 1987, pp. 726-739.