An O(1) Time Complexity Software Barrier

Liqun Cheng and John B. Carter
School of Computing
University of Utah
50 South Central Campus Drive, Room 3190
Salt Lake City, UT 84112
{*legion, retrac*}*@cs.utah.edu*

Abstract

As network latency rapidly approaches thousands of processor cycles and multiprocessors systems become larger and larger, the primary factor in determining a barrier algorithm's performance is the number of serialized network latencies it requires. All existing barrier algorithms require at least $O(\log N)$ round trip message latencies to perform a single barrier operation on an N-node shared memory multiprocessor. In addition, existing barrier algorithms are not well tuned in terms of how they interact with modern shared memory systems, which leads to an excessive number of message exchanges to signal barrier completion.

The contributions of this paper are threefold. First, we identify and quantitatively analyze the performance deficiencies of conventional barrier implementations when they are executed on real (non-idealized) hardware. Second, we propose a queue-based barrier algorithm that has effectively $O(1)$ time complexity as measured in round trip message latencies. Third, by exploiting a hardware write-update (PUT) mechanism for signaling, we demonstrate how carefully matching the barrier implementation to the way that modern shared memory systems operate can improve performance dramatically. The resulting optimized algorithm only costs one round trip message latency to perform a barrier operation across N processors. Using a cycle-accurate execution-driven simulator of a future-generation SGI multiprocessor, we show that the proposed queue-based barrier outperforms conventional barrier implementations based on load-linked/store-conditional instructions by a factor of 5.43 (on 4 processors) to 93.96 (on 256 processors).

Keywords: shared memory multiprocessors, synchronization, barriers, write update, coherence protocols.

1 Introduction

Since 1987, processor performance has improved at a rate of 55% per year due to increasing clock rates and die sizes and decreasing feature sizes. However, DRAM latency has only decreased by 7% per year and interprocessor communication latencies have dropped only slightly in terms of wall clock time As a result, the round trip communication latency between the nodes of a large share memory multiprocessor is rapidly approaching a thousand processor cycles. This growing gap between processor and remote memory access times is impacting the scalability of many shared memory algorithms, and in particular synchronization operations are becoming increasingly expensive. Relatively slow synchronization has become a major obstacle to sustaining high application performance on scalable shared memory multiprocessors [3, 11].

In this paper, we focus on analyzing and improving the performance of *barriers*, a common synchronization operation often used in modern shared memory algorithms [2, 9, 15]. Barriers synchronize a large number of cooperating threads that repeatedly perform some work and then wait until all cooperating threads are ready to move on to the next computation phase, e.g., as follows:

```
for (i = 0; i < MAX; i++) {
    DoWork(thread_id);
    ...
    BarrierWait(barrier);
}
```

Figure1 illustrates the timing information for a single barrier operation. We define the time at which

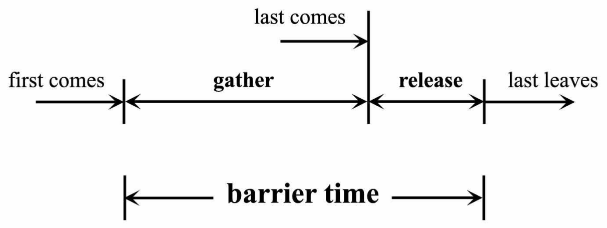

Figure 1. Timing information for barrier synchronization

(a) naive coding

```
atomic_inc( &gather_variable );
spin_until( gather_variable == num_procs );
```

(b) "optimized" version

```
int count = atomic_inc( &gather_variable );
if( count == num_procs-1 )
      release_variable = num_procs;
else
      spin_until( release_variable == num_procs );
```

Figure 2. Traditional barrier pseudo-code

the first processor performs a `BarrierWait()` operation as the barrier start time. We define the time at which the last thread has been signaled that the barrier operation is complete and returns from the `BarrierWait()` operation as the barrier completion time. The total time required to perform a single barrier operation is the difference between the barrier start time and the barrier end time. We divide this time into two components, the time during which each thread signals its arrival at the barrier, which we denote the *gather* phase, and the time it takes to convey to each thread that the barrier operation has completed and it is ok to resume execution, which we denote the *release* phase. Between the time when a thread signals its arrival at the barrier and the time that it is signaled that the barrier operation has completed, it can perform no other computation. To motivate the need to improve barrier performance we measured the time it takes to perform a 32-thread OpenMP barrier operation on a 32-node Origin 3000. We found that in the time it takes to perform a 32-node barrier operation, the system could have executed 5.76 million FLOPS.

Traditionally, barriers have been implemented by having each thread increment one or more barrier count variables located in shared memory, e.g., as illustrated in Figure 2. Barrier completion is signaled via a release flag [6] that each thread checks repeatedly until it indicates that all threads have arrived at the barrier.

Traditional barrier implementations often suffer from contention during both the gather phase, when all the processors must atomically update a count, and during the release stage, when all the processors must read a release flag. For example, Figure 2(a) illustrates a naive barrier implementation where the count and signal variables are the same; threads waiting for the barrier operation to complete spin reading the count variable, which is updated each time a new thread reaches the barrier. This naive algorithm results in $O(N^2)$ coherence protocol messages being sent per barrier to invalidate and reload the shared count.

Figure 3 illustrates the source of these $O(N^2)$ coherence messages in a typical barrier implementation running on a 3-node CC-NUMA (cache coherence non-uniform memory access) multiprocessor system. Solid lines represent request and data messages, dashed lines represent intervention messages (i.e., ones that request data from a remote cache or request that a remote cache invalidate its copy of a shared cache line), and dotted lines represent intervention replies. In the illustrated scenario, we assume that all three processors start with a read-only (shared) copy of the cache line containing `gather_variable` in their cache, and one thread on each node arrives at the barrier at approximately the same time. Each thread attempts to perform the `atomic_inc()` operation, which causes each processor to send a request to the home node of the barrier count variable asking that the other cached copies of the count be invalidated and the requesting node be given write access to the corresponding cache line (messages (1), (2), and (3)). Only the first request to arrive at the barrier variable's home memory controller will be granted write access (message (8)), which occurs only after the other processors have been sent invalidation messages (messages (4) and (5)) that have been acknowledged (messages (6) and (7)). The remaining two processors will again compete for write access to the barrier count variable, which will generate another round of invalidations and acknowledgements. As the figure shows, the simple algorithm barrier requires 18 messages before all three processors can increment the barrier count, plus a few more messages (not shown) for each processor to detect that the barrier count has reached its upper limit. Even worse, relatively few of these messages are *pipelined* (overlapped); each message round trip adds to the overall barrier completion time, which can lead to very poor performance.

Previous optimized barrier algorithms have sought to minimize contention and the number of messages

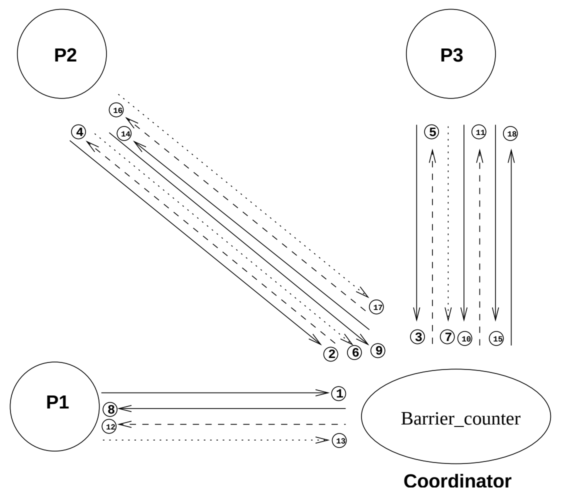

Figure 3. Traditional barrier

```
struct flag_type{
      int flag;
      int padding[31];
      /*make sizeof(flag_type) = L2C line size*/
}

/* In practice, these are carefully allocated*/
/* in physical memory on coordinator node*/
/* and cache line aligned.                    */
struct flag_type gather_flags[N];
struct flag_type release_flag;

Thread(i):
-----------
...
for (;;) {
     ...
     /* Set my completion flag */
     atomic_inc(gather_flags[i]);

     /* Wait for coordinator to signal completion */
     while(!release_flag);
}

Coordinator:
-------------
for (;;) {

      for (z =0; z<N; z++)
         while(!gather_flags[z]);
      atomic_inc(release_flag);
}
```

Figure 4. Queue-based barrier

required to synchronize in a number of ways. For example, Figure 2(b) illustrates a barrier implementation that uses separate count and signal variables for each thread [16]. However, as we show in Sections 2 and 3, previous algorithmic optimizations have not eliminated all sources of contention, and in some cases even lead to reduced performance on real shared memory systems.

In this paper, we present a novel barrier algorithm that is carefully designed to perform well on real shared memory multiprocessors, which typically employ write-invalidate protocols and which can support only a limited number of outstanding cache misses by any particular processor. To eliminate contention for the single global barrier count variable, we employ an array of per-thread *signal flags*. Each thread participating in the barrier synchronization signals its arrival at the barrier by writing to its flag variable in the global flag array, and then spins on a separate *release flag*. A separate coordinator thread continually examines the shared flags until they indicate that all threads have arrived at the barrier. When the coordinator determines that all threads have arrived at the barrier, it updates the *release flag* to signal all threads that they may exit the barrier spin loop. To minimize inter-processor memory contention, the *signal flags* are allocated in separate cache lines (via array padding) in physical memory located on the coordinate node (via careful memory allocation)

Our barrier algorithm, as shown in figure 4 outperforms conventional barrier algorithms for several reasons. First, since each thread updates an independent *signal flag*, no invalidation and reload coherence protocol messages are generated due to inter-thread contention. Second, because each thread updates an independent *signal flag*, their updates, or rather the coherence protocol messages needed to perform the updates, can be pipelined (performed concurrently). If the round trip communication latency is much larger than the time it takes the coordinator's memory controller to respond to a single read or write request, which is increasingly the case in large-scale multiprocessors, O(N) protocol exchanges can be overlapped such that they occur in roughly O(1) time (as measured in round trip message latencies). Third, by using a single (separate) *signal flag*, as opposed to one signal flag per thread as proposed by some researchers [13], we avoid the problem that modern processors can only signal a limited number of processors at a time before the load-store unit runs out of Miss Status Handling Registers (MSHRs) and thus stalls the processor pipeline. As a result of these optimizations, our best queue-based barrier algorithm that assumes no special hardware support for updates outperforms the baseline OpenMP barrier implementation by a factor of 7.9X on 256 processors. On a system that supports write updates (or coherent PUT operations), our optimized barrier implementation performs 94X as well as the baseline OpenMP barrier implementation and 6.5X faster than barriers using SGI's proprietary memory controller-based atomic operations (MAOs).

Looking ahead to architectural features that have been proposed for future multiprocessors, we investigated the extent to which judicious use of write update protocols (or PUT operations) could improve both gather and release operations. A write update protocol can reduce the amount of coherence traffic required to update the value of a shared variable in a remote node by eliminating the invalidations and reloads that occur using a conventional write-invalidate shared memory coherence protocol. Using updates to signal barrier arrival and barrier completion, our optimized barrier algorithm can reduce the time required to synchronize N threads to a single message round trip, assuming round trip message latency dwarfs memory controller overhead, which is the limit of how fast a barrier operation can be performed.

The rest of the paper is organized as follows. We review various hardware and software barrier implementations in Section 2 to provide background for this work. In Section 3 we describe a variety of queue-based barrier implementations in detail. In Section 4 we present our simulation results, and in Section 5 we present our conclusions.

2 Background

A number of barrier implementations have been published over the years. Among them, the hardware barriers of Cray et. al. are the fastest [2, 11, 19, 20]. Pure hardware barrier require a pair of wired-AND lines between every two nodes. Processors signal arrival at the barrier by pulling the input wire voltage high, and then wait for the output wire to be high to signal completion. While provably optimal in terms of performance, this approach is only feasible in a small scale system, where the number of dedicated wires is manageable. The requirement for $N*(N-1)$ unidirectional wires on a N node system is prohibitive when N is large. Besides the high cost of dedicated wires, static hardware approaches cannot synchronize an arbitrary subset of threads and do not perform well when the number and identity of participants in a barrier change over time.

Most barrier implementations are software-based, but exploit whatever hardware synchronization support is provided on the particular platform. Many modern processors, including MIPS$^{\mathrm{TM}}$ [8], Alpha$^{\mathrm{TM}}$ [1], and PowerPC$^{\mathrm{TM}}$ [12] rely on load linked / store conditional (LL/SC) instructions to implement atomic operations. These instructions are used as follows. A thread performs a load linked operation, which causes the thread to start tracking external accesses to the loaded data. If the contents of load linked memory location are changed before the subsequent store conditional to the same address, then the store conditional fails. If a context switch occurs before the subsequent store conditional, then the store conditional also fails. A successful SC implies that the read-write pair is effectively atomic from the point of view of any other process. To implement other synchronization primitives, libraries typically retry the LL/SC pair until the SC succeeds.

In an LL/SC-based barrier implementation[8], each thread loads the barrier count into its local cache before trying to increase it atomically. Only one thread will succeed on the first try while the SCs on all other threads will fail. This process repeats itself as each new thread arrives at the barrier, until all participating threads have atomically increased the barrier count. For this basic barrier algorithm, average barrier latency increases superlinearly with respect to the number of synchronizing threads because (i) round trip network latency increases as system size increases and (ii) contention for the single barrier count variable increases as the number of threads increases. In the worst case, when there is significant contention and thus frequent backoff-and-retry cycles, $O(N^2)$ round trip message latencies are required to complete the gather stage of the barrier operation. When contention is light, updates to the barrier count occur sequentially, resulting in $O(N)$ round trip message latencies to complete the gather stage of each barrier operation. Regardless of load, $O(1)$ round trip message latencies are required for the release phase, because the N threads are invalidated and the barrier count reloaded in parallel. As a result, the best case time complexity of LL/SC-based barriers is $O(N)$ round trip message latencies, while the worst case complexity is $O(N^2)$. The average case is highly application dependent, and depends on the relative ratio of computation to synchronization and the average skew in completion time of each thread – the more skew, the less contention, although large amounts of skew can cause performance problems due to load imbalance.

Note that while all N threads can be invalidated in parallel, and each of the N threads can send a reload request for the cache line containing the barrier count in parallel, the memory controller than is the *home node* for the cache line can only handle one request at a time. For our analysis, we assume that round trip message latency dwarfs the controller occupancy required to handle a single protocol message, which is true for processor configurations up into the low hundreds of nodes. In this case, the predominant performance factor is the $O(1)$ serialized round trip message latencies, not the $O(N)$ protocol operations on the home node's memory controller. Readers interested in the details of what a memory controller does in response to a remote read request can find it discussed in detail elsewhere [6].

Replacing the LL/SC try-retry loop with atomic *fetch-and-incr* instructions can eliminate failed SC attempts, thereby improving the performance. Goodman *et al.*[3] propose *fetch-and-ϕ* as a generic hardware atomic primitive and Michael *et al.* [15] demonstrate how these primitives can be used to reduce synchronization contention. Some modern processors support special instructions that perform a variety of *fetch-and-ϕ* operations, e.g., the Itanium IA-64's semaphore instructions [7]. These types of instructions are often referred to as *processor-side atomic operations*, because the data is loaded into the processor cache and modified there atomically. Data still must be invalidated from remote caches before it is modified, and invalidated threads must reload the data across the interconnect before they can accesses it. Although barriers implemented using processor-side atomic operations induce less serialization and scale better under heavy load, their low contention performance is essentially the same as LL/SC-based barriers. The global shared counter must be updated by every single thread, and every atomic update costs a round trip message latency, so the gather stage still has $O(N)$ time complexity. As a result, barriers implemented using processor-side atomic operations do not significantly outperform LL/SC-based barriers.

The NYU Ultracomputer [4, 9] implements a variety of atomic instructions in its memory controller. Further, it uses a combining network that tries to combine all loads and stores for the same memory location in the routers. Combining is useful only when barriers are global and accessed frequently, because the combining mechanism can slow down other requests in an attempt to induce opportunities to combine. In contrast, the SGI Origin 2000 [10] and Cray T3E [19] implement similar combining functionality, but do so at the barrier variable's home memory controller. This design eliminates the problems associated with combining in the router. In the SGI Origin 2000 and Cray T3E, threads trigger atomic memory operations by issuing requests to special IO addresses on the home node memory controller of atomic variable. The home node MC interprets these requests and performs the update operations atomically. We refer to these mechanisms as *memory-side atomic operations*. Compared to processor-side atomic operations, memory-side atomic operations simplify the design of processor pipeline and save system bus bandwidth. However, each atomic operation still requires a round trip across the network, which needs to be done serially.

Some researches have proposed using barrier trees to reduce synchronization contention and overlap communication in large-scale systems [5, 18, 21]. Barrier trees employ a hierarchy (tree) of barriers. Rather than centralize the barrier implementation through a single global barrier or coordinator, tree-based barrier algorithms divide the participating threads into modest-sized groups of threads that synchronize amongst themselves. When the last thread in a subgroup reaches the barrier, it signals its arrival at the next higher level barrier in the barrier tree. This process continues recursively until the last thread arrives at the barrier at the root of the barrier tree, which initiates a series of cascading signal operations that spread back down the barrier tree. If we assume the maximum fanout in the tree is M, both the gather and release stages can complete in $\lceil \log_M N \rceil * O(M)$ round trip latencies. A tree-based barrier on a large system is essentially a series of smaller barriers. For example, 256 threads could synchronize by employing a four-level barrier tree, with a fanout of four at each level in the tree. Since barrier operations at the same level of the tree can be done in parallel, the time required for this 256-thread barrier is only roughly four times that of a base 4-thread barrier.

The queue-based algorithm presented in the following section requires only $O(1)$ message round trips to synchronize N threads. However, this $O(1)$ result assumes that the time for a given memory controller to perform N protocol operations is less than a single message round trip latency. While this assumption holds true for reasonable sized values of N, it does not hold for arbitrary sizes of N. For large values of N, a hybrid barrier solution employing barrier trees combined with our queue-based barriers for synchronization within a level of the barrier would perform best. Our algorithm improves the performance of individual subtree barrier synchronization, which allows us to increase the fanout in the tree and thereby reduce the height of the barrier tree. Determining what combination of tree- and queue-based barrier provides the best performance for various sizes of N is part of our future work.

Table1 shows the time complexities of existing barrier solutions and our proposed queue-based barrier algorithm as measured in round trip message latencies.

3 Algorithms

In this section, we describe our proposed queue-based barrier mechanism, starting with a simple version in Section 3.1, followed by a series of refinements in the subsequent subsections. We call our algorithms "queue-based" due to their similarity in spirit and data structures to Scott *et al.*'s queue-based spinlocks [14]. However, our queue-based barrier is quite different than simply implementing a barrier using queue-based spinlocks. A barrier can be implemented using two spin locks, one to protect the barrier count variable and another on which threads can block until the barrier operation completes. However, this design requires every thread

Algorithm	Gather stage	Release stage	Total
LL/SC Average case	$O(N)$	$O(1)$	$O(N)$
LL/SC Worst case	$O(N^2)$	$O(1)$	$O(N^2)$
Atomic(Processor side)	$O(N)$	$O(1)$	$O(N)$
Atomic(Memory side)	$O(N)$	$O(1)$	$O(N)$
Barrier Tree	$O(\log N)$	$O(\log N)$	$O(\log N)$
Queue-based	$O(1)$	$O(1)$	$O(1)$

Table 1. Time complexity of various barrier implementation

to acquire and release each lock once per barrier iteration, which would result in a barrier time complexity of $O(N)$.

3.1 Simple Queue-Based Algorithm

In our first queue-based barrier algorithm, we designate one node as the *coordinator* and allocate an array of flags, one per participating thread, in the coordinator's local physical memory. To eliminate false sharing, we pad the flag array such that each thread's flag variable resides in a separate cache line. When a thread arrives at the barrier, it sets its flag to TRUE. This operation involves a single round trip message latency – in response to the attempt to update the flag on the signaling thread, a READ_EXCLUSIVE protocol message would be sent to the flag variable's home node (i.e., the coordinator). Upon receiving this coherence protocol message, the coordinator's memory controller would invalidate the coordinator's processor cache (if necessary) and then supply a writable copy of the appropriate cache line to the signaling thread. The *gather stage* of the barrier operation completes when the last thread sets its flag variable. Figure 5 depicts the coherence protocol messages exchanged during the gather phase of our simple queue-based algorithm. Note that each thread can update its flag variable effectively in parallel, since the protocol request and response messages are independent of one another and controller occupancy will be low for reasonable-sized systems. This is illustrated in Figure 5 by the fact that request messages 1a, 1b, and 1c can occur in parallel (or at least be pipelined). As a result, using an array of flag variables rather than a single global count reduces the effective number of round trip message latencies required for N threads to signal arrival from $O(N)$ to $O(1)$.

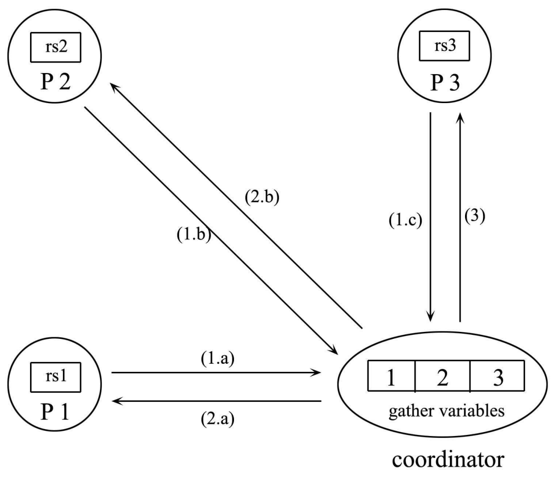

Figure 5. Simple Queue-based Barrier

To determine when the barrier operation is complete, the coordinator sweeps through the flag array until it sees that all of the flags are set. This sweep requires $O(1)$ message round trips, because although the coordinator will load N cache lines containing flags from the corresponding remote processor caches, only the cache line load corresponding to the last thread to signal its arrival at the barrier impacts performance.

To determine when they are allowed to finish the barrier operation, each participating thread spins on a second private flag variable, which the coordinator sets when all threads have arrived at the barrier. The speed at which this release operation can be performed is limited by how fast coordinator can modify all of the private completion flags. At first glance, this might appear to be an $O(1)$ operation, since the writes are all independent and thus can be pipelined. However, modern out-of-order processors can only support a modest number of outstanding memory operations. Processors have a limited number of Miss Handling Status Registers (MSHRs); if the processor accesses a cache line not present in its local cache when no MSHRs are available, the processor pipeline stalls. The impact of this architectural limitation is that the coordinator will only be able to pipeline K updates at a time, if K is the number of MSHRs for that processor.

Referring back to Figure 5, if the processor in question has only 2 MSHRs and thus can support at most 2 outstanding remote writes at a time, then the coordinator cannot issue the write request to P3 until one of the other updates (2a or 2b) completes. As a result, in this case the signaling phase took two round trip latencies (the time for 2a or 2b to complete followed by the time for 3 to complete) In general, on a system that can support k outstanding remote writes, the coordinator needs at least $\lceil N/k \rceil$ round trip times to finish complete the signaling phase. This hardware constraint on performance is discussed in more detail in the following section.

The overall execution time of this queue-based barrier scheme is the sum of the gather stage ($O(1)$) and the release stage ($O(N/k)$), where k is the maximum pending writes the system can support. Consequently, due to the oft-overlooked restriction of the number of outstanding memory operations a single processor can have a time, the overall time complexity of this algorithm is $O(N)$.

3.2 Optimized Queue-Based Algorithm

Since the gather stage of our initial queue-based barrier algorithm is already an $O(1)$ operation, we focus on reducing the time complexity of the release stage. Actually, our baseline algorithm is $O(1)$ in terms of software operations, provided the coordinator can scan the flag array in $O(1)$ time. This holds true as long as the time constant for round trip message latencies dwarfs the time constant of local memory reads, which is true for practical system sizes.

As noted in the previous section, the problem with employing a separate signal flag for each participating thread is that the coordinator processor can only issue in parallel as many invalidation requests as it has MSHRs. After all participating threads have arrived at the barrier in the algorithm described above, the coordinator must modify each thread's private release flag so that the corresponding thread will know that it can quit spinning. Before the coordinator signals a thread, the thread will be sitting in a tight loop repeatedly reading the value of its release flag variable. All modern processors employ a write-invalidate-style MESI cache coherence protocol. When the coordinator thread attempts to modify a particular thread's flag variable, its local cache would detect that the coordinator's copy of the corresponding cache line was in SHARED mode. In response to the write request, the coordinator's cache controller will issue a read-exclusive operation to the corresponding thread's cache controller, asking it to invalidate its local copy of the cache line containing the flag variable. Modern processors have miss status handling registers (MSHRs) to keep track of outstanding misses. When the processor has no free MSHRs, subsequent cache misses stall Without loss of generality, assume a particular processor has 8 MSHR entries. In this case, if the coordinator attempts to write the flag of a ninth thread before any of the first eight invalidates have been acknowledged, it will stall. In this case the coordinator can only pipeline 8 remote writes at the same time and $\lceil N/8 \rceil$ serial round trips are required to finish the release stage.

Increasing the number of MSHRs, and thereby increasing the number of outstanding remote misses that can be tolerated, would clearly help alleviate this problem. However, Palacharla *et al.* [17] note that further increases in windows size, issue queue length, and the number of MSHR entries is not complexity effective.

Given the limited number of MSHRs in real processors, we must reduce the number of remote writes. Using a single global release variable for all participating threads eliminates the MSHR problem. A single write by the coordinator causes the coordinator's memory controller to issue N pipelined invalidation requests before the coordinator can modify the flag, which are quickly followed by N pipelined read requests as each participants re-reads the (now modified) flag variable.

Figure 6 illustrates how the optimized algorithm works. After all participating threads have arrived at the barrier, they spin on local copies of a shared global release variable contained in their local caches. All the copies are invalidated before coordinator

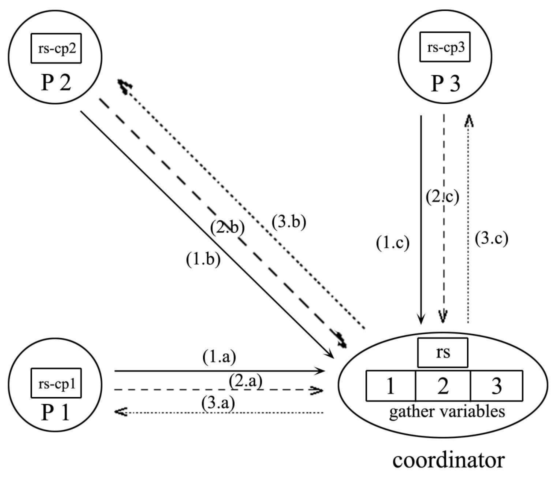

Figure 6. Optimized Queue-based Barrier

updates the value of the release variable. Then each thread reloads a copy of the release variable after the update has completed. Since all round trips related to read contention for the single global release variable are automatically pipelined by modern MESI protocols, the time complexity of the release stage drops to $O(1)$ in our optimized algorithm, again assuming protocol handling time on a given node is dwarfed by message latency.

Combined with the algorithm $O(1)$ for the gather stage described in Section 3.1, the resulting barrier time is reduced to $O(1)$ round trip message latencies.

3.3 Simple Update Algorithm

The algorithm described in Section 3.2 provides an $O(1)$ round trip latency solution to the barrier problem, but its constant coefficients are not as low as (say) a hardware wired-AND barrier. In particular, during the gather phase, each participating thread suffers a remote write miss when it attempts to update its flag variable (resulting in N read- -exclusive messages and N corresponding acknowledgements). During the release phase, the coordinator invalidates all N copies of the shared release flag variable (resulting in N invalidation messages and N acknowledgement messages), and then each of the N participating threads suffers a read miss (resulting in N read-shared messages and N data-return messages). Even though these coherence protocol messages are pipelined so that they are performed as three sets of N concurrent (pipelined) messages, the protocol described in Section 3.2 requires a minimum of three round trip message latencies per barrier synchronization. As round trip latencies approach 1000 processor cycles, this overhead limits barrier performance, especially for small configurations where the benefit of protocol message pipelining is small. In Sections 3.1 and 3.2 we developed algorithms that improved barrier latency by aggressively overlapping (pipelining) coherence message traffic. The problem we address in this and the following section is how to reduce the number of non-overlapped round trip message latencies that remain to signal arrival at the barrier and/or completion of the barrier operation and how to reduce the number of coherence operations performed by the coordinator's memory controller.

We first consider how to reduce the performance impact of the two sets of round trip message latencies required for the coordinator to signal barrier completion. Recall that these two sets of round trip messages are for (i) the coordinator to invalidate all N shared copies of the barrier completion flag and then (ii) for each of the N participating threads to reload a shared copy of the flag. In conjunction with researchers at SGI, we are investigating the value of *write update* coherence protocols in scalable shared memory multiprocessors. The write update protocol we are investigating allows threads to perform either explicit coherent GET/PUT operations on cacheable shared data or to designate particular regions of memory as ones that should be kept coherent using a hardware write update coherence protocol. Our proposed directory controller tracks coherence at the cache line level, as is the norm in existing scalable shared memory systems. When the home memory controller of a particular variable receives a GET request, it returns a coherent value of the target variable loaded either from local memory or a remote processor cache, depending on the state of the cache line. In response to a GET operation, the requesting node is not added to the cache line's list of sharers, and hence will not be informed of future modifications of the cache line. In response to a PUT operation, the home memory controller sends an update request to local memory and every processor that has a copy of line containing the target variable, where the modification is applied[1]. In our simple update algorithm, we have the coordinator use PUTs to update the global release variable after the last thread reaches the barrier. This eliminates one of the two round trips required to perform a release in the optimized queue-based algorithm described above. A secondary benefit of the use of PUTs is that they require smaller protocol messages (e.g., 8-bytes versus 32-128 bytes plus overhead to send a full cache line). As we report in Section 4.3 this bandwidth reduction provides additional performance benefits.

3.4 Optimized Update Algorithm

In this section we consider how to improve the performance of the gather phase of our queue-based barrier implementation using PUTs. Recall that during the gather phase, each participating thread will suffer a remote write miss when it attempts to update its flag variable, which will result in the corresponding cache line being invalidated from the coordinator's processor cache. When the coordinator subsequently tests the flag to determine if that thread has arrived at the barrier, it will suffer a read miss and be forced to reload the cache line across the network. As described in Section 3.2, the round trip overhead of independent invalidate requests can be pipelined, but invalidation requests can queue up at the coordinator's directory controller. Since coordinator is in the critical path of our centralized barrier implementation, every delay it experiences impacts barrier performance. As such, high controller occupancy on the coordinator node induced by the N invalidations and subsequent N reloads performed during the gather phase can increase barrier latency by a non-negligible constant factor, especially in large configurations. This overhead can be reduced by having participating threads use PUTs to update the value of their private flag on the coordinator node, thereby eliminating an invalidation and reload cycle. As we report in Section 4 the judicious use of cache-coherent PUTs for signaling reduces the number of protocol messages that pass through the coordinator's network interface by 70%, which results in an additional 7.3X speedup compared even to our optimized queue-based barrier implementation.

[1]Details of how coherent GET and PUT are physically implemented are beyond the scope of this paper.

3.5 Summary

Figure 2(a) shows a naive barrier implementation, where `num_procs` is the number of participating threads. This implementation is inefficient because it spins on the the gather variable directly, which as discussed in Section 1 can lead to significant contention problems and even in the best case requires $O(N)$ round trip message latencies to complete because the updates are effectively serialized.

A common optimization to this barrier implementation is to use a separate release flag, as shown in Figure 2(b). Instead of spinning on the barrier count variable, the optimized version loop spins on a separate release variable that is only updated when the last thread arrives at the barrier. Programmers need make sure the gather variable and release variable do not reside in the same cache line to avoid contention due to false sharing. This implementation performs one more write per barrier operation than the naive implementation. This extra write to the release flag causes copies of the shared flag to be invalidated from all N nodes, who in turn issue N read requests to load the updated flag variable. Nikolopoulos *et al.* [16] report that spinning on a separate variable improves barrier performance by 25% over an implementation that spins directly on the barrier count for a 64-node barrier.

In our two basic queue-based barriers, pseudocode for which appear in Figure 7, an array of cache-line-aligned flags (one per thread) is allocated in the physical memory of a coordinator node. To signal arrival at the barrier, participating threads update their private slot in this array. The coordinator repeatedly sweeps through the array to determine when all participating threads have arrived at the barrier, which induces N remote read misses as it reloads the updated flag variables. However, in practice these misses and the induced message traffic are effectively pipelined. Our two queue-based algorithms differ only in how the coordinator signals barrier completion to the waiting threads,

Participating processors:

```
atomic_inc( &gather_variable );
spin_until( release_variable);
```

Coordinator :

```
spin_until(Forall i: gather_variable[i]==TRUE);
Set_Update_Protocol(release_variable);
atomic_inc(release_variable);
```

Figure 7. Queue barrier pseudo-code

either via private signal flags (S-Queue) or via a shared signal flag (O-Queue).

Finally, we discussed how the judicious use of a proposed update protocol could improve the constant coefficients of our queue-based barrier algorithm. To support this optimization, we assume a system where software can specify on a per-page basis whether the shared memory coherence hardware should employ a write invalidate or write update protocol. If software wishes to employ an update protocol for a particular data structure, it invokes the `Set_Update_Protocol` system call, which is supported by our simulator [22] and shown in Figure 7.

4 Evaluation

In this section we present details of our experimental methodology and results. We describe the simulation environment we employ for all experiments in Section 4.1 and compare the results of the various barrier implementations described earlier in section 4.2

4.1 Simulator Environment

We use a cycle-accurate execution-driven simulator, UVSIM, in our performance study. UVSIM models a hypothetical future-generation Origin 3000 architecture that we are investigating along with researchers from SGI. The simulated architecture supports a directory-based coherence protocol that supports both write invalidate and write update coherence protocols. The hardware write update protocol is implemented using implicit "GET/PUT" operations as described in Section 3.3. Each simulated node contains two next-generation MIPS microprocessors connected to a future-generation system bus. Also connected to the bus is a next-generation HUB chip, which contains the processor interface, memory controller, directory controller, network interface and IO interface.

Parameter	Value
Processor	4-issue, 48-entry active list, 2GHz
L1 I-cache	2-way, 32KB, 64B lines, 1-cycle lat.
L1 D-cache	2-way, 32KB, 32B lines, 2-cycle lat.
L2 cache	4-way, 2MB, 128B lines, 10-cycle lat.
System bus	16B CPU to system, 8B system to CPU
	max 16 outstanding L2C misses, 1GHZ
DRAM	16 16-bit-data DDR channels
Hub clock	500 MHz
DRAM	60 processor cycles latency
Network	100 processor cycles latency per hop

Table 2. System configuration.

Table 2 summarizes the major parameters of our simulated system. The DRAM back end has 16 20-bit channels connected to DDR DRAMs, which enables us to read an 80-bit burst every two cycles, 64 bits of which are data. The remaining 16 bits are a mix of ECC bits and partial directory state. The simulated interconnect is based on SGI's NUMALink-4, which uses a fat-tree structure with eight children on each non-leaf router. The minimum-sized network packet is 32 bytes and we model a network hop latency of 50nsecs (100 cpu cycles). We do not model contention within the routers, but do model port contention on the hub network interfaces. We have validated our simulator by configuring it with parameters that match an Origin 3000 system and found that all predicted results for a wide set of tuning benchmarks are within 20% of the real machine, most within 5%. Key performance metrics, e.g., elapsed cycle counts and cache miss rates, are typically within 1%.

4.2 Results

In this section we report the relative performance of seven barrier implementations for 4-256 processors. All the programs in our study are compiled using the MIPSpro compiler version 7.3 with an optimization level of -O2. We use OpenMP's barrier implementation for the SGI Origin 3000 as the baseline against which we compare all other barrier implementations. OpenMP's barrier is implemented using LL/SC instructions on the Origin 2000 system.

We compare the performance of six other barrier implementations against the baseline OpenMP implementation. The first two alternative implementations replace the LL/SC instructions with conventional processor-side atomic fetch-and-inc instructions (*Atomic*) and SGI-specific memory-side atomic instructions (*MAO*). The Atomic version simply replaces the LL/SC instructions with more efficient atomic instructions, whereas the MAO version exploits the optimization proposed by Nikolopoulos *et al.* [16] for MAOs.

In addition to these three conventional implementations of barriers, we consider four queue-based barriers: our basic queue-based barrier that uses separate flags for both signaling arrival at the barrier and completion of the barrier operation (*S-Queue*), a version that uses a single variable to signal barrier completion (*O-Queue*), and versions of both algorithms that employ updates (*S-Update* and *O-Update*, respectively).

All results reported herein correspond to 1001 barrier operations per thread. The first barrier is used simply to synchronize the start time of our measurements. The barrier time is measured as the time from when the first barrier operation completes until the last thread leaves the 1000^{th} barrier.

Table 3 presents the normalized speedups of the six optimized barrier implementations compared to the baseline OpenMP LL/SC-based barrier implementation. We vary the number of threads that synchronize from 4 (i.e., 2 nodes) to 256 (i.e., 128 nodes). Columns 2 through 7 of Table 3 present the speedups of various optimized barrier implementations.

All the implementation show noticeable improvement compared to the baseline version. Of the algorithms that require only the basic hardware support that could be expected on any scalable shared memory multiprocessor (baseline, atomic, S-Queue, and O-Queue), our optimized queue-based algorithm (O-Queue) performs 7.85 times faster than the baseline and approximately 5 times faster than any other algorithm for 256 processors. This demonstrates the importance of being aware of every source of protocol overhead and the problem of MSHR-induced stalls when designing a scalable barrier mechanism.

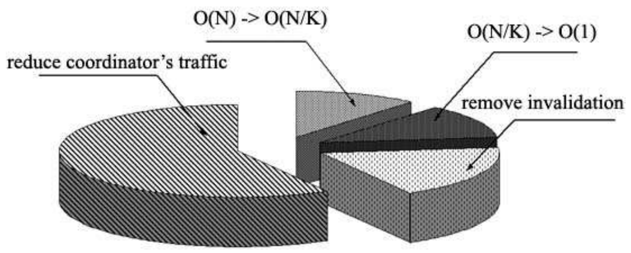

Figure 8. Sources of Performance Improvement for 32P Barrier

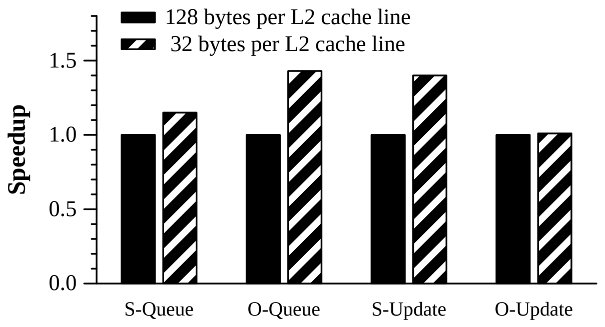

Figure 9. Bandwidth effect on Algorithm

All three of the algorithms that require special hardware support (MAO, S-Update, and O-Update) perform well, although the value of avoiding MSHR-induced stalls (S-Update vs O-Update) is particularly important when protocol overhead is reduced by the use of update protocols. Overall, the O-Update algorithm significantly outperforms all other algorithms, outperforming the baseline by almost a factor of 94 and its next closed competitor (MAO) by a factor of 6.5.

Figure 8 breaks down the source of performance improvement for the O-Update algorithm. Over half of the improvement comes from using updates. The remainder comes from having participating threads signal independent variables $O(N) \rightarrow O(N/K)$), using a single variable to signal barrier completion ($O(N/K) \rightarrow O(1)$), and eliminating extraneous invalidates.

CPUs	Speedup over Baseline					
	Atomic	MAO	S-Queue	O-Queue	S-Update	O-Update
4	1.15	1.21	0.58	0.65	0.95	**5.43**
8	1.06	2.70	1.23	2.02	2.73	**18.04**
16	1.20	3.61	1.11	2.56	3.98	**25.04**
32	1.36	4.20	1.16	3.14	4.44	**31.71**
64	1.37	5.14	1.01	4.23	5.92	**43.59**
128	1.24	8.02	1.13	5.06	6.54	**44.39**
256	1.23	14.70	1.58	7.85	9.61	**93.96**

Table 3. Speedup of various barrier implementations versus the OpenMP barrier baseline

4.3 Sensitivity to Bandwidth On/Off the Coordinator Node

As described in Section 3.4, controller occupancy at the coordinator can limit the performance of the queue-based barrier implementations. Essentially, when occupancy is high enough, the assumption that the overhead of handling individual protocol operations is negligible compared to a round trip message latency is not completely accurate. When this happens, the $O(N)$ protocol operations performed at the coordinator begin to lose their insignificance compared to the $O(1)$ round trip message latencies, resulting in a less scalable algorithm.

When the fraction of time spent handling protocol messages becomes significant compared to the inter-node communication overhead, a more precise formula for the performance of the various algorithms is $\tau + \frac{km}{\phi}$, where τ is the round trip message latency, m is size of a single request packet, k is the number of packets handled by the coordinator node, and ϕ is the network bandwidth on/off the coordinator node. We use bandwidth as our metric for coordinator controller occupancy because in practice it is is the delimiting factor for the rate at which the controller can handle the simple protocol messages induced by our barrier algorithms.

To investigate the extent to which controller occupancy impacted performance, we tested the sensitivity of our queue-based algorithm to cache line size, which in our experiments was the primary factor in determining how many bytes were sent on/off the coordinator node. For this experiment, we simply compared how the performance of the various algorithms changed when we reduced the L2 cache line size from 128 bytes to 32 bytes, which effectively decreases m by a factor of 4. As can be seen in Figure 9, reducing the L2 cache line size improves the performance of S-Queue, O-Queue, and S-Update by factors of 1.15, 1.43, and 1.40, respectively. The performance of O-Update algorithm was essentially unchanged. These results tell us that network bandwidth, and thus controller occupancy, was not a major factor in performance even for the 256-processors barrier case.

5 Conclusions and Future Work

Efficient synchronization is crucial to the performance and scalability of applications running on large scale share memory multiprocessors. In this paper, we analyze a variety of existing barrier solutions and identify sources of unnecessary performance overhead when run on real shared memory hardware.

We propose a family of novel *queue-based barrier algorithms* that eliminate most sources of serialization in existing barrier implementations. By aggressively exploiting pipelining and being aware of the limitations of modern processors (especially in terms of the limited number of misses that can be outstanding from any given processor), the resulting algorithms can perform an N-thread barrier operation in $O(1)$ message round trip latencies. For practical machine configurations, e.g., up to 256 processors, the $O(1)$ message round trip latencies dominate the $O(N)$ protocol operations performed by the memory controller on the coordinator node. For systems large enough for the $O(N)$ factor to be significant, the queue-based barrier algorithms presented herein can be used as the base barrier algorithm in an $O(log(N))$ barrier tree.

On a 256-processor system, our O-Update algorithm demonstrates a 94X speedup compared to the baseline LL/SC-based OpenMP barrier algorithm. Compared to other algorithms that exploit specialized machine features, O-Update outperforms algorithms that use memory-side atomic ops (MAO) by a factor of 6.5X and ones that employ processor-side atomic operations (Atomic) by a factor of 75X. The best queue-based algorithm that does not exploit special hardware features (O-Queue) outperforms the baseline OpenMP barrier implementation by a factor of 7.9X on 256 processors.

As part of our future work, we plan to determine the extent to which queue-based barrier algorithms can be combined with MAOs and barrier tree. Also, we plan to test the sensitivity of our algorithm to network latency and investigate what minimal set of hardware primitives is ideal to support efficient synchronization.

References

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

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

[3] J. R. Goodman, M. K. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessor. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Apr. 1989.

[4] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU multicomputer - designing a MIMD shared-memory parallel machine. *IEEE Transactions on Programming Languages and Systems*, 5(2):164–189, Apr. 1983.

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

[6] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

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

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

[9] C. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):570–601, October 1988.

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

[11] S. Lundstrom. Application considerations in the system design of highly concurrent multiprocessors. *IEEE Transactions on Computers*, C-36(11):1292–1390, Nov. 1987.

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

[13] J. Mellor-Crummey and M. Scott. Synchronization without contention. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, Apr. 1991.

[14] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, Feb. 1991.

[15] M. M. Michael and M. L. Scott. Implementation of atomic primitives on distributed shared memory multiprocessors. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 222–231, Jan. 1995.

[16] D. S. Nikolopoulos and T. A. Papatheodorou. The architecture and operating system implications on the performance of synchronization on ccNUMA multiprocessors. *International Journal of Parallel Programming*, 29(3):249–282, June 2001.

[17] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA*, pages 206–218, 1997.

[18] M. Scott and J. Mellor-Crummey. Fast, contention-free combining tree barriers for shared memory multiprocessors. *International Journal of Parallel Programming*, 22(4), 1994.

[19] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

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

[21] P. Yew, N. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, Apr. 1987.

[22] L. Zhang. UVSIM reference manual. Technical Report UUCS-03-011, University of Utah, May 2003.