To appear in the 6th International Symposium on High-Performance Computer Architecture (HPCA), Jan. 10 - Jan. 12, 2000, Toulouse, France

Improving the Throughput of Synchronization by Insertion of Delays

Ravi Rajwar,[†] Alain Kägi,[‡] and James R. Goodman[†]

[†]*Computer Sciences Department*
University of Wisconsin-Madison
Madison, Wisconsin 53706 USA
{rajwar,goodman}@cs.wisc.edu

[‡]*Microprocessor Research Labs*
Intel Corporation
Hillsboro, Oregon 97124 USA
alain.kagi@intel.com

Abstract

Efficiency of synchronization mechanisms can limit the parallel performance of many shared-memory applications. In addition, the ever increasing performance gap between processor and interprocessor communication may further compromise the scalability of these primitives. Ideally, synchronization primitives should provide high performance under both high and low contention without requiring substantial programmer effort and software support. QOLB has been shown to offer substantial speedups and to outperform other synchronization primitives consistently [17], but at the cost of software support and protocol complexity. This paper proposes the use of speculation and delays to implement a purely hardware-based queueing mechanism called Implicit QOLB. *Making use of the pervasiveness of the Load-Linked/Store-Conditional primitives, we present a series of hardware mechanisms to optimize performance for sharing patterns exhibited by locks and associated data. The mechanisms do not require any change to existing software or instruction sets. IQOLB sits alongside the cache-coherence protocol and guides the decisions the protocol makes with respect to lock (and associated data) transfers. Preliminary evaluations indicate that IQOLB may perform as well as, if not better than, QOLB without the additional software and protocol complexity.*

1. Introduction

There are two key aspects to supporting efficient synchronization in shared-memory multiprocessors: (1) recognizing that an application has issued a synchronization operation, and (2) exploiting this knowledge to maximize performance. Unfortunately, in current memory systems, synchronization requests are often hard to distinguish from other memory requests.

While many papers [1, 2, 4, 9, 10, 11, 12, 21, 24] suggest that maintaining a dichotomy in the cache coherence protocol between regular memory and synchronization requests can improve the execution of parallel applications substantially, no commercial system has embraced these techniques because of their perceived cost.

In this paper we propose to implement efficient synchronization by speculating about the programs' access patterns and by possibly delaying coherence actions. The benefits of this approach are threefold: (1) it does not require requests to be differentiated; instead we speculate about the nature of a request, (2) no special instruction is required, (3) no recompilation is required to benefit from our techniques (even without recompilation our techniques can outperform the best known synchronization methods).

To demonstrate the viability of our approach, we design and implement *Implicit QOLB (IQOLB)*, which provides the benefits of QOLB but without its drawbacks. We demonstrate that IQOLB can be implemented using two instructions commonly found on contemporary processors: Load-Linked and Store-Conditional.

Classic cache coherence protocols are optimized for memory access patterns of regular data. Yet, memory access patterns for locks are different than for normal data accesses. Typically, a lock variable is written (usually involving an atomic read-modify-write operation) by a successful acquisition at the beginning of a critical section. The same processor then releases the lock at the end of the critical section by performing another write to it. During the critical section, different processors waiting to acquire the lock may read the value of the lock variable repeatedly. Such sharing behavior, while common for contended locks, is generally not well supported by cache coherence protocols. The external reads to the lock variable will force the lock holder to relinquish exclusive ownership of the lock variable. Thus, in order to release a contended lock, the holder must re-acquire write permission.

Critical sections are often small, particularly those at the lowest levels. Thus it may be possible to increase throughput and reduce interprocessor communication traffic by delaying the transfer of a cache line containing a lock until after the lock is released. This observation led to the proposal for a *synchronous prefetch* [11], which allowed a processor to request a lock, but provided information so that the processor holding the lock could defer supplying the cache line until after the lock was released. This concept naturally extends to the notion of a queue of waiting processors, each waiting its turn to acquire exclusive access to the lock, and was the basis for QOLB [11], which exploited the sharing pattern of lock variables and provided substantial performance gains.

IQOLB guides the cache coherence protocol to make intelligent—and sometimes delayed—choices. Speculating on whether an operation is a lock acquire or an atomic read-modify-write (such as Fetch&Φ), the appropriate cache coherence action is invoked. This speculation results in a performance improvement of both a lock acquire operation and a traditional Fetch&Φ operation. These mechanisms form the core of this paper and are detailed in section 3.

We use the Load-Linked/Store-Conditional (LL/SC) instructions to demonstrate a possible implementation of IQOLB. These instructions have found widespread acceptance in modern processor architectures [18, 26, 32] under different names. In section 2, we discuss QOLB and LL/SC operations in depth. Our use does not require any change to the architected semantics of the instructions, allowing existing software using these instructions to benefit from our mechanisms without requiring any changes or recompilation.

2. Background and related work

QOLB, originally proposed by Goodman, Vernon, and Woest [11], was the first proposal for a queue-based synchronization primitive. QOLB maintains, in hardware, a queue of processors waiting to acquire a lock. When the processor at the head of the queue releases the lock, it transfers the lock in a single message directly to the next processor in line, if any. When a processor requests a lock, it first allocates local space for the lock variable and then sends a request for the cache line containing the lock or, if it is currently unavailable, to join the QOLB queue of processors waiting to acquire it. It then waits for the lock, spinning locally until the lock is delivered. In this way, QOLB can achieve a very efficient transfer of a lock: the queue-based primitive optimizes lock hand-off and local spinning avoids unnecessary network traffic while processors are waiting for the lock. Since every processor spins on the same address without evicting or downgrading the lock holder's copy, it is possible to allocate protected data in the same cache line as the lock—a strategy referred to as *collocation*. This allows the protected data to be transferred along with the lock and eliminates the read/write overhead in the critical section. Details of an SCI implementation of QOLB can be found elsewhere [15].

Kägi, Burger, and Goodman [17] showed that over a range of benchmarks and processor technologies, queue-based locking and collocation are the most effective aspects of QOLB. Their results also showed that local spinning provides marginal gains. An important result of the work was in demonstrating consistent and large gains of QOLB over all other synchronization primitives studied for a range of benchmarks. These gains were much more impressive than others had predicted, and underscored the magnitude of delay in acquiring data in a critical section, not just the lock.

Over the years, various synchronization mechanisms have been proposed [29, 16, 11, 3, 13, 27, 25, 14]. Most synchronization operations use an atomic read-modify-write primitive. While not all architectures provide a synchronization primitive, they offer some form of instruction that atomically swaps a value (either predefined or contained in a register) with one in memory. Conditional versions of this instruction also exist—with the understanding that the swap occurs only if the value read from memory is the same as a specified value. More complex primitives such as Fetch&Φ provide the ability to perform a simple operation—usually an arithmetic addition or increments to a variable in memory. This provides valuable opportunities for parallel execution if the Φ operation satisfies certain properties because multiple operations can be executed concurrently through a procedure known as combining [12].

Many contemporary microprocessors provide the LL/SC instructions to implement atomic read-modify-write operations to cached memory locations. Fetch&Φ can be implemented using the LL/SC operations. Such implementations are useful for enqueue, dequeue, software barriers, and ticket lock implementations in addition to providing efficient ways to implement software barriers.

The LL/SC instructions were originally proposed by Jensen, Hagensen, and Broughton [16] as Load-Locked/Store-Conditional instructions. These instructions expose the steps involved in performing the atomic read-modify-write operation to the programmer and rely on the cache coherence protocol to ensure correctness. The LL instruction loads a memory location into a processor register. This is followed by an arbitrary sequence of operations involving the register. The SC then attempts to write the same memory location as the previous LL operation. The SC will succeed only if the hardware can guarantee that no other processor has successfully written to the memory location since the most recent LL instruction was executed. Thus, a successful SC operation implies that a read-modify-write operation occurred atomically, completing at the time of the SC. In the case of a failure, the entire sequence may be retried.

The LL/SC paradigm has been adapted for several architectures—Load-Locked/Store-Conditional in the Alpha [32], Load-and-Reserve/Store-Conditional in the IBM PowerPC [26], and Load-Linked/Store-Conditional in the MIPS [18]. In various implementations, a link flag and registers are used to store the LL information. A register typically stores the physical address to which the LL was issued. The link flag is set when the LL is issued. The success of the SC operation can only be determined at the point of coherency, that is, at the time the write operation can be performed on the designated memory location. If, at that point, the link flag is still set, and no incoming invalidate to the address in the locked physical address register is encountered, the SC can successfully complete. Implementation details vary depending on the architecture and the coherence mechanism; the semantic remains the same.

While the basic concept is elegant and simple, in theory permitting the implementation of an arbitrarily complex synchronization primitive, in practice it is difficult to design a system that can reliably guarantee success of the LL/SC sequence. Two obvious problems demonstrate the difficulty: (1) a memory conflict that forces the cache line corresponding to the LL address to be evicted, and (2) an intervening page fault or interrupt resulting in large delays between the LL and SC. The common solution used for (1) is to prohibit memory operations that may result in an eviction. Preventing memory operations may also solve the problem in (2). It is hard to account for all possible actions that may occur in an LL/SC sequence. In order to address this, many architectures provide a set of guidelines and requirements to increase the likelihood of a successful LL/SC sequence.

For example, in the Alpha Architecture Handbook [8], three pages of description are required to explain each of the LL and SC instructions along with many restrictions. Such restrictions include (1) that a write of a different word on the same block (the size of a block being an implementation-dependent constant of some power of two, being no smaller than a cache line, and no larger than a page) as the target address may cause the SC to fail, (2) that numerous system calls or traps will cause the link flag to be reset, (3) that there be no instructions that access memory between the LL and SC instructions, (4) that there be no taken branches between the immediately preceding LL and SC instructions (the processor may execute multiple LL instructions before it attempts the SC), and (5) that a "large number" of instructions not be executed between the LL and SC. The term "large number" is not defined—though they do require a minimum number of instructions every implementation must execute between timer resets.

While the simplicity of these primitives makes them appealing, and seemingly very general, the implementation-aware architectural restrictions imposed on the LL/SC primitives tend to discourage creative use of the instruction pair for operations more complex than traditional Fetch&Φ. Since LL/SC is optimistic, under high contention, the performance of such atomic operations degrades rapidly due to an increase in failed LL/SC sequences. Complex primitives therefore are generally built using LL/SC to implement an underlying lock operation.

In the next section, we propose an implementation of the LL/SC semantics that provides good atomic read-modify-write performance even under high contention. We then extend this mechanism to optimize for specific lock behaviors.

This work is inspired by the work of Mukherjee and Hill [28] who proposed to use prediction to speed up coherence protocols, by the work of Kaxiras and Goodman [19] who discussed instruction- and address-based prediction to improve the performance of multiprocessor systems, and Lai and Falsafi [22] who first designed and analyzed a full parallel system encompassing ideas from the first two papers.

3. Synchronization algorithms

In an LL/SC sequence, the objective is to modify a target location atomically. Thus, it might seem obvious that the initial request for data—initiated by the LL instruction —should read the data for exclusive ownership. We are unaware of any implementations that do this. A problem with this approach is the difficulty in guaranteeing that any processor will ever succeed. For example, consider the sequence of two processors P1 and P2 attempting to perform an atomic operation on the same variable. Assume that both processors successfully complete the LL instruction in short succession (say P1 succeeds first). If the LL operation obtains a writable copy, P2's LL operation may invalidate P1's copy (and reset the link flag) before P1 is able to complete the SC instruction. In turn, P1, on a retry, may destroy P2's copy before P2 has performed its SC. Thus, the two processors may enter a live-lock situation.

In a given case of contention, which processor successfully completes an SC instruction does not matter, except that the system should not be so biased as to permit the same processor to succeed repeatedly. In general, it is sufficient if some processor can be guaranteed to succeed at a non-zero rate (though this guarantee does not by itself assure that other processors will not starve).

This observation can be exploited to enhance the performance of synchronization. If processor P1 is permitted to acquire a writable copy of the lock on a LL instruction, and another processor P2 generates a similar request before P1 executes the SC instruction, the maximum system throughput can nevertheless be reached by allowing P1 to complete its SC instruction before giving up the line to P2. This behavior may appear unfair from the standpoint of the cache coherence protocol, since P2 broadcasts its request for ownership before P1 decides to write. However, if it can be ascertained with a high probability that P1 will soon write the variable, allowing it to hold onto the data long enough to complete the SC operation successfully, the number of external requests generated is reduced since otherwise P1 will have to acquire the cache line again in order to complete the sequence.

A convenient way of visualizing this operation is through the use of relativity arguments: unless P2 can somehow observe the timing of P1's SC request, it has no way of knowing whether the request occurred before or after it made its own conflicting request for the cache line. Effectively, P2's LL request can be said to have occurred after P1's SC request, even though in fact the LL request was received before the SC request. Of course, a strict limit must be maintained regarding the amount of time-warping permitted. P1's cache cannot be allowed to wait indefinitely while delaying P2's request for the cache line. Such delay may also introduce concerns about memory ordering. However, we observe that, while sequential consistency constraints require a global ordering of events, that order need not be the same as the order of requests observed on the bus.

The concept of a delayed response becomes more interesting in the presence of multiple requests. If processor P3 also issues an LL instruction, P1 cannot send writable copies to both P2 and P3. Here, the notion of time-warping can be extended to reason about multiple requests. Assuming that P2's request is observed before P3's, P2 can expect to receive the cache line before P3, and can in fact be made responsible for passing the data on to P3, perhaps after a small additional delay to allow P2 to complete its LL/SC sequence. In this way a queue of outstanding requests is built up, even if every processor concurrently attempts to acquire exclusive access to the cache line, and the line will be passed in a writable state from one processor to the next, in precisely the order in which the original requests occurred.

This section has considered reading for ownership to satisfy an LL instruction. Since the LL will likely be followed shortly by an SC instruction, this solution is intuitively appealing, but under high contention it can provide very poor performance, and even lead to a lack of forward progress. We note, however, that the hardware could selectively choose this option, speculating conservatively but nevertheless saving the additional operation in the cases of uncontended locks. For example, it might choose to request ownership on the first LL instruction encountered after a successful SC instruction. This would prohibit live-lock by ensuring that the failure would only occur once. While we haven't investigated this algorithm in detail, we believe it would always perform as well as the baseline case, and better under most circumstances.

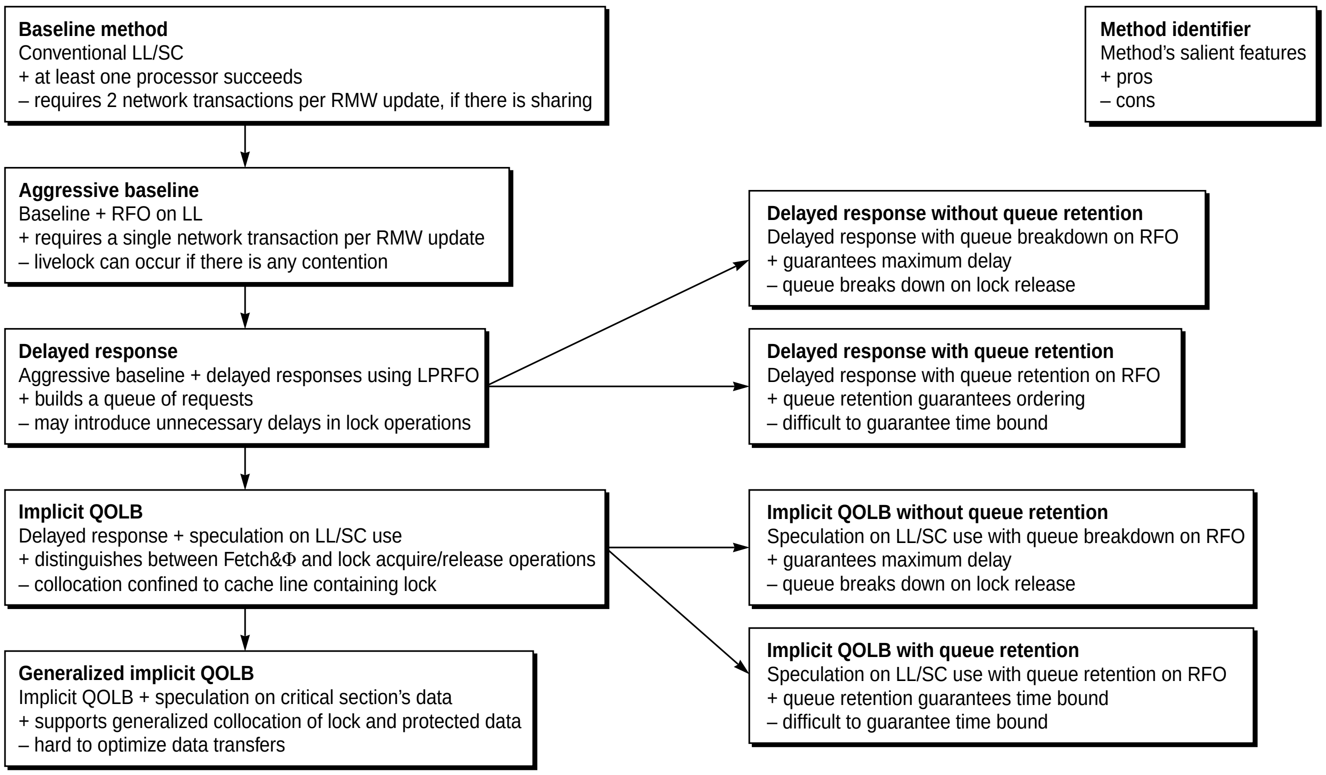
In the following subsections, we explore the above concepts in detail and study a series of models for improving synchronization performance. Figure 1 depicts the progression of these models. It starts with the *Baseline* or *Traditional* LL/SC model in the upper left corner and progresses downward through the different models described in the next sections.

3.1. Baseline LL/SC implementation

This type of implementations corresponds to the default protocol used in many systems we studied, with possibly minor variations. In these implementations, the LL instruction fetches a memory location in a shared state. A successful SC then requires a second network transaction to obtain an exclusive copy of the corresponding cache line. Overall, two network requests are generally required for performing the atomic read-modify-write operation.

Figure 2 depicts a typical LL/SC sequence. Processors P1 and P2 each execute an LL instruction to the same memory location resulting in two read requests issued simultaneously. After receiving a response, each processor attempts the execution of the SC instruction. Necessarily, one of the two processors obtains the exclusive copy of the data first, say P1, forcing P2 to reset its link flag. Thereupon, P2 must retry the sequence. As more processors issue LL instructions to the same address concurrently, the number of such failed sequences increases, causing poor performance.

Though successful execution of a traditional LL/SC sequence requires two network requests, some special cases may require fewer requests. For example, if no cache owns a copy of the targeted memory location, a cache may obtain the cache line in a writable state. Also a well-known programming trick may reduce the number of requests required to perform an atomic memory operation [20]: a program may perform a write to an unused location in the same cache line as the lock before issuing the LL instruction. Consequently, the line will be fetched in an exclusive state. In the absence of contention, the program can perform the atomic operation with a single network transaction.This coding technique requires the programmer's involvement and software modifications. Additionally, the programmer must take great care to ensure that forward progress constraints are met. Finally, this technique performs very poorly in the presence of contention. This technique is shown as the *Aggressive baseline* method in Figure 1 (second frame from the top).



Method identifier
Method's salient features
+ pros
– cons
Baseline method
Conventional LL/SC
+ at least one processor succeeds
– requires 2 network transactions per RMW update, if there is sharing
Aggressive baseline
Baseline + RFO on LL
+ requires a single network transaction per RMW update
– livelock can occur if there is any contention
Delayed response
Aggressive baseline + delayed responses using LPRFO
+ builds a queue of requests
– may introduce unnecessary delays in lock operations
Delayed response without queue retention
Delayed response with queue breakdown on RFO
+ guarantees maximum delay
– queue breaks down on lock release
Delayed response with queue retention
Delayed response with queue retention on RFO
+ queue retention guarantees ordering
– difficult to guarantee time bound
Implicit QOLB
Delayed response + speculation on LL/SC use
+ distinguishes between Fetch&Φ and lock acquire/release operations
– collocation confined to cache line containing lock
Implicit QOLB without queue retention
Speculation on LL/SC use with queue breakdown on RFO
+ guarantees maximum delay
– queue breaks down on lock release
Implicit QOLB with queue retention
Speculation on LL/SC use with queue retention on RFO
+ queue retention guarantees time bound
– difficult to guarantee time bound
Generalized implicit QOLB
Implicit QOLB + speculation on critical section's data
+ supports generalized collocation of lock and protected data
– hard to optimize data transfers

Figure 1. Overview

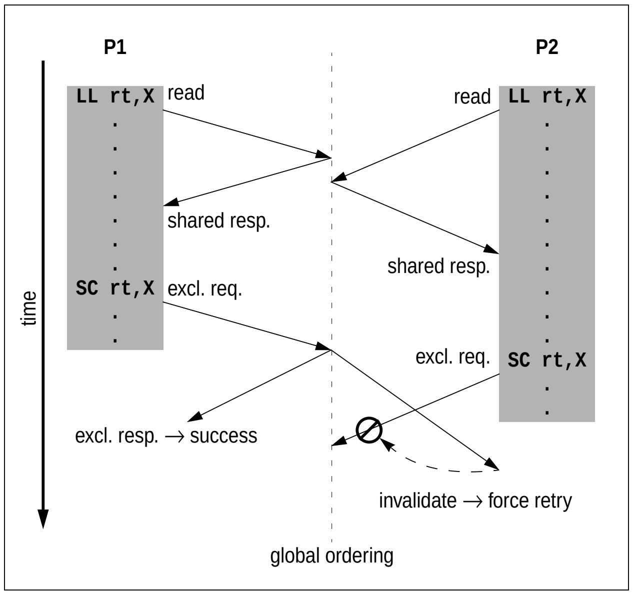


Figure 2. Traditional LL/SC sequence

Note that the hardware could also speculate about the likelihood of success and request initially ownership of the cache line containing the lock. It might, for example, speculate that success is likely at the first opportunity after a successful SC instruction. In the absence of any contention, this would only require a single network request, but would cause some additional failures under heavy contention. We could find no cases where this scheme increases traffic over the base case, and believe that such a scheme can be justified even for conservative implementations.

3.2. LL/SC implementation with delayed response

The previous section discusses requesting data exclusively to satisfy an LL instruction. While at first glance intuitively appealing since it is likely that an SC instruction will soon follow the LL instruction, we observe that this solution can lead to a lack of forward progress and can perform extremely poorly under high contention.

Based on these observations, our second extension not only proposes to request data exclusively upon executing an LL instruction but also to delay processing a request for a copy that a cache controller believes is about to be written. To guarantee correctness, however, this delay must be finite and the cache controller should process other cache requests with circumspection in order not to violate the constraints of memory consistency. If an SC instruction does not occur within a certain time, a time-out mechanism guarantees that the cache controller will eventually forward the cache line to the requesting node.

The success of this scheme relies on the timer to trigger infrequently. We believe that time-outs will indeed be infrequent because architectural specifications typically insist on a very limited amount of instructions between pairs of LL and SC instructions.

In summary, this scheme allows a processor to perform an atomic memory operation in a single network transaction, most of the time.

The scheme works very well for atomic memory operations like Fetch&Φ. However, if the LL/SC sequence is used to implement the primitives for a lock operation, there is a performance problem. Lock operations have two phases: acquire and release. During the acquire phase, when P1 completes the acquire successfully, even if the cache line containing the lock is passed on to the next waiting processor, say P2, that processor will immediately discover that it cannot obtain the lock currently held by P1. At this point P1's cache no longer possesses a copy of the lock and P2 must now wait for P1 to release the lock. For P1 to release the lock, it will have to re-request ownership for the cache line containing the lock. However, P1 is now forced to wait for P2 to time-out (since P2 will never execute the SC instruction it speculated about). This behavior results in potentially poor performance in the case of locks. Furthermore, the situation worsens as more processors attempt to acquire the same lock since the lock must now be passed down the queue, waiting at each node until the time-out forces its release.

To address the performance problem with our current mechanism, we introduce the concept of priorities among ownership requests and refine our protocol.

In the new scheme, instead of a read-for-ownership (RFO) request, the LL instruction issues a "low-priority read-for-ownership" (LPRFO) request. A response to an LPRFO request can be delayed for a substantial, but bounded, time duration determined by the time-out mechanism. On the other hand, a normal read-for-ownership request must be serviced within a small (a few cycles) delay. Note that the two operations can easily be differentiated: restrict LPRFO to LL instructions. Considering the same scenario of a lock hand-off described earlier, a lock release operation will result in a normal read-for-ownership. The requestor obtains write permissions much faster than in the original mechanism described in this sub-section.

We call this improved mechanism the *Delayed response* scheme (third frame from the top in Figure 1).

Figure 2 portrays a typical sequence of operation for the delayed response scheme. It shows three processors issuing concurrent LPRFO requests for the same address and it shows these processors forming a queue, each processor waiting for the previous one to complete the read-modify-write operation. In the figure, P1 obtains an exclusive response first. P1, now being responsible for the line, recognizes P2's request but delays the actual response. P1, upon a successful completion of the SC operation, transfers the corresponding cache line to P2. In effect, our mechanism introduces a delay (Δ) between the time P2's request reaches the network and the time P1 services it. Similarly, P2 delays the response to P3's request, and sends the data after a delay (Δ'). Note that no processor need retry the LL/SC sequence unlike what would typically happen with the traditional LL/SC method.

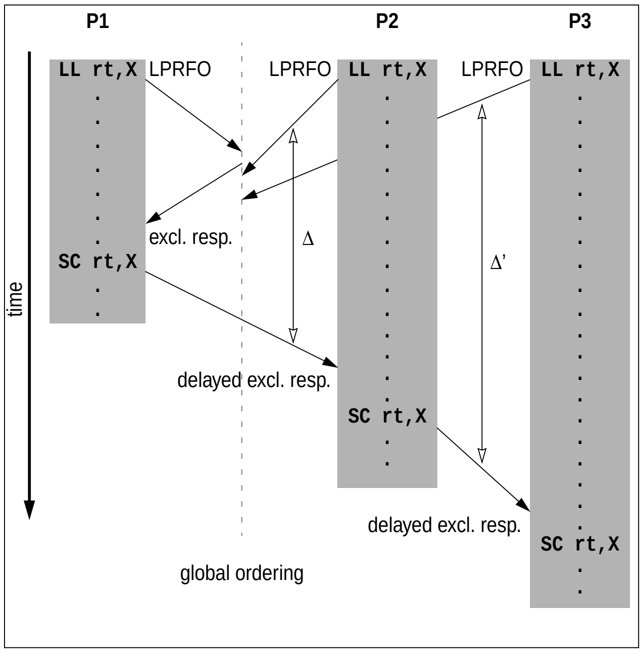


Figure 3. LL/SC sequence with delayed response

Our proposal still occasionally suffers from performance problems when dealing with locks. Thus, when a regular read-for-ownership is issued, the owner of the line responds with little delay. An invalidation implied by the request squashes the outstanding LPRFO requests on all other waiting processors, thus breaking down the queue. In such an event, waiting processors reissue the LPRFO requests and build a new queue, possibly in a different order. This event causes additional network traffic, although we believe less than the baseline protocol.

As an alternative, one can eliminate the problem of a queue breakdown by allowing a node to insert itself at the head of the queue without breaking down the queue. We call these alternatives *with* and *without queue retention*. Figure 1 shows both alternatives on the right-hand side. On observing a regular request for ownership, the head of the queue sends data along with a special marker. The special marker forces the requestor to transfer ownership back immediately once the write completes.

The presence of collocated data exacerbates the performance problems of both alternatives. If the queue is not preserved, reads and writes to collocated data result in frequent queue breakdowns. If the queue is retained, the cache line holding lock and data may traverse the network often going back and forth between the head of the queue and the node issuing regular RFOs. The latter scheme is also more complex and may have problems guaranteeing bounded delays.

In both alternatives, a time-out at the head of the queue forwards the line along to the next processor in the queue. In the event that the owner of the line evicts the lock line, the ownership along with the data is transferred to the next requestor, i.e., an eviction is treated as a time-out.

The notion of a time-out mechanism is not an attractive requirement, and we have resorted to this mechanism only after an extensive search for alternatives. We note, however, that the time-out mechanism can be fairly simple. First, while it is necessary to maintain information about multiple critical sections, it is not necessary to support more than a single outstanding request requiring a time-out. In general, the speculation is aimed at the lowest-level critical sections, and we anticipate only marginal gains for higher-level locks. Therefore, if a second, nested, critical section is entered, the first can generally be discarded with respect to speculation. Furthermore, the speculative delay is optional, and can always be disposed of by responding at once. Thus when a second request arrives, a controller with only a single time-out can still make a choice between the two, servicing one while speculatively delaying the other.

This ability to dispose of a speculative delay becomes critical if the period of delay is significantly increased. In the next section we explore the possibility of delaying beyond the successful SC instruction.

3.3. Speculating on lock operations

So far, we have used the concept of delays to build a queue of processors waiting to perform a read-modify-write operation. We can further improve the performance of synchronization, if we can speculate on the usage of LL/SC sequences present in programs. If a sequence performs a simple Fetch&Φ operation, we would like to apply the protocol detailed in the previous section (the delayed response method). Alternatively, if the sequence acquires a lock to be released some time later, we would like to delay requests until the program releases the lock (i.e., beyond the execution of the SC instruction that acquires the lock). As a result of this extended delay, we form a queue of lock requestors in hardware similar to the queues in queue-based locks (e.g., QOLB). Hence we call this method *Implicit QOLB* or *IQOLB*. This method is shown near the bottom of the overview chart in Figure 1.

The queue of lock requestors improves the lock transfer time, takes full advantage of collocation, and avoids the problem of a cache line going back and forth between two caches as may occur in the previous (delayed response) method.

Sizes of critical sections and network operations introduce delays larger than those with which the previous method must deal. Delaying servicing of an external lock request until that lock is released may substantially increase the time a processor is blocked. To prevent waiting processors from being blocked and to prevent tying up resources unnecessarily, we introduce the notion of a *tear-off copy* [23]. The response is speculative in the sense that a copy of the cache line's current content is given away without giving up ownership. An interesting side effect is that we are able to capture the effect of local spinning on a lock variable and we allow the requestor to use the data at one moment in time, but not retain it for later use. The receipt of a tear-off copy signals a successful insertion into the queue of requestors. The tear-off copy provides the requestor with information it needs: the lock is not currently available, but will arrive as soon as available.

The entire cache line need not be returned as the speculative response. The primary effect of the response is to allow the requesting processor to complete its LL instruction, with the expectation that it will decide not to attempt the SC instruction, but spin on the LL. Thus it is only necessary to return the actual word requested. The requesting processor can in fact use the returned value for multiple reads, subject to constraints of memory ordering. It cannot, for example, read or write any other cache lines, then use read the value again if it is supporting sequential consistency.

The basic sequence of operations involved in the IQOLB method is:

1. Speculate that an LL instruction operates on a lock and issue an LPRFO;
2. Delay ownership transfers to external lock requestors, but service them speculatively; and
3. Identify the instruction releasing the lock and transfer data to the next requestor in line.

Figure 4 illustrates three processors attempting to enter the same critical section concurrently. All processors issue LPRFOs with P1 obtaining the associated lock first, followed by P2 and P3, in that order. There are two kinds of responses: (1) regular exclusive responses, possibly delayed, and (2) tear-off copies. Assuming the lock is initially free, P1 can enter the critical section immediately. While in the critical section, it delays responding to requests, sending speculative responses if it speculates that indeed P1 is holding a lock and not performing a simpler Fetch&Φ. A speculative response frees network resources and allows the receiving processor to speculate on the con-

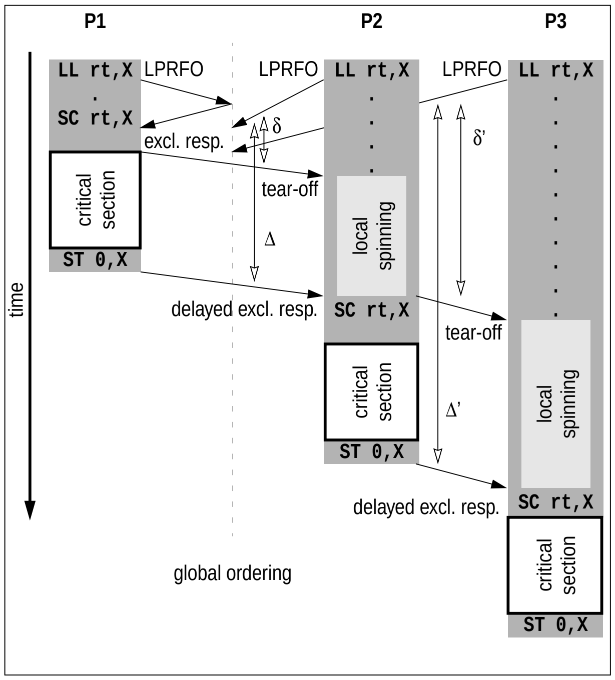


Figure 4. IQOLB sequence

tent of the cache line containing the lock. P2 receives the speculative response and learns that the lock is currently held. P2 then spins locally waiting to receive the lock. When P1 releases the lock (by writing the lock variable), it transfers ownership of the cache line containing the lock to P2. Upon receiving the lock, P2 can enter the critical section.

Cache line eviction transfers ownership and data to the next processor in line. Read requests to locks currently held are speculatively satisfied by issuing tear-off copies. This allows for a processor interested in querying the state of the lock to proceed without being involved in the queue.

As before we refine the IQOLB method into two different alternatives (denoted *with* or *without queue retention*) based on the course of action in the event of a write to a lock line. When the lock owner and holder are the same, the lock unset operation proceeds efficiently; the releaser forwards the lock to the next requestor. The complication occurs when the lock-holder and lock-owner are different. In the first alternative (no queue retention) a write to the lock causes an invalidation of all entries in the queue, forcing all processors to re-acquire the lock. Of course, the order of re-insertion may not match the original order. The other alternative (the queue retention method) preserves the queue. In this method, when a lock-holder requests write permission, the owner responds to the request but requests the cache line back immediately. All other processors ignore this transaction and preserve the state of the queue. This method avoids queue breakdown at the expense of a more complex implementation, of fairness and of forward progress. A similar problem may occur if collocated data is present in the cache line and the line is evicted before an unset.

A point to remember is that performance will be poor under the above scenario even with traditional locks. Cache evictions are events that can degrade performance across all synchronization primitives and need to be handled correctly.

3.4. Prediction of lock acquires and releases

As discussed earlier, the LL instruction can be used as part of a simple Fetch&Φ operation or be used as part of a lock primitive. We have proposed two methods, the delayed response method and IQOLB, which specifically optimize each of these two cases. To decide which mechanism to apply, one can speculate. IQOLB speculates on an LL instruction being used as part of implementing a lock and optimizes accordingly. The delayed response method speculates on the LL instruction implementing a simpler atomic read-modify-write and provides an optimization specific to it. To obtain high performance, it is important to have high prediction accuracies in such speculation. By speculating on the instruction being a basis for a simple Fetch&Φ or a lock primitive, it will most often be the case that the speculation is indeed correct. This is an important basis for the high confidence in our speculation. In our simulations, the benchmarks always used LL/SC to implement locks and so we had perfect behavior. The instruction PC was used to index into a lock-predictor table. Our mechanism for inferring an unset operation was as follows.

A lock operation is inferred by observing the sequence of memory operations to an address. A lock operation has an acquire phase (during which the LL/SC primitive is employed) and a release phase which may be a simple write operation to the lock variable. A predictor table consisting of the instruction PC associated with the acquire (and possibly an address) is used. A lookup determines whether this instruction was a lock operation and a prediction is made accordingly. Once a lock operation is seen, one can predict with high confidence that this will be true for all future executions of the code. The pathological case can be detected by determining the accuracy of prediction and turning the predictor off. A lock operation is speculated by observing a successful LL/SC on a location followed sometime in the future by a write to the same location of the lock—this is necessary to prevent writes to collocated or falsely shared data being interpreted as an unset operation.

In addition to a lock predictor table, we need a table to keep track of locks currently held by the processor. This is required in order to be able to determine quickly when to

Table 1. Baseline system

Processor	
Instruction window	64 entry reorder buffer, 32 entry load/store queue
Issue mechanism	out-of-order issue/commit of ≤ 4 instructions per cycle, speculative loads
Branch predictor	8K-entry gshare, 8-bit history; 2K-entry BTB with 2-bit saturating counters; 32-entry return address stack, static sequential target predicted on a BTB miss
Cache subsystem	
L1 data cache	64-KB, 2-way set associative, write-back, write-allocate, dual-ported, non-blocking., ≤ 8 outstanding misses, 1-cycle hit, MESI
L1 instruction cache	64-KB, 2-way set associative, dual-ported, non-blocking, ≤ 8 outstanding misses, 1-cycle hit
L2 unified cache	512-KB, 4-way set associative, write-back, write-allocated, non-blocking, ≤16-outstanding misses, 6-cycle hit (uncontended), MOESI
L1/L2 bus	≤ 4 concurrent accesses to the L2, runs at processor speed
Line size	64 bytes
Memory bus	split address/data buses, split transactions, ≤ 117 outstanding requests
Address bus	broadcast-based MOESI protocol, 12-cycle access latency
Data network	point-to-point crossbar, 40-cycles latency per cache line transfer
Memory	8-byte wide, 40-cycle access time for first part of cache line, 4-cycle time for subsequent accesses.
Consistency model	sequential consistency

delay a response. The table can be small. On a successful LL/SC sequence, the table is updated. We do not need to store information prior to success—as that is captured by the LL instruction support in the form of link flags and locked physical address registers etc. On a release of a lock, the entry is removed from the table.

Now, the choice of optimizations is restricted to the delayed response method or IQOLB. It can be ascertained with a high degree of confidence that the instruction PC (and additionally in combination with the address being accessed) corresponds uniquely either to the simple Fetch&Φ operation or to a lock primitive. In the rare (and possibly impossible) case of the same PC actually corresponding to two different types of uses, the misspeculation recovery mechanism guarantees correct behavior.

4. Experimental methodology

We implement our proposal IQOLB (from here on we will no longer distinguish between the variants regarding queue retention, since we did not observe queue breakdown in our simulations) and compare its performance against that of QOLB and a simple implementation of the test&test&set algorithm using the LL/SC primitive. The major objective is to determine whether IQOLB and QOLB perform comparably. We are also interested in comparing our mechanism with the baseline LL/SC implementation—a simple LL/SC-based test&test&set implementation. The results do not attempt to take advantage of potential collocation benefits.

4.1. Simulation environment

We use an execution-driven simulator developed for performing detailed studies of modern shared-memory multiprocessor systems. The simulator is based on the SimpleScalar Toolset [6] and allows both detailed and high performance simulations of distributed shared-memory multiprocessor systems. The simulator performs a cycle-by-cycle simulation of an aggressive out-of-order processor and a detailed event-driven simulation of the memory hierarchy. The processor core is similar to the out-of-order core (“sim-outorder”) distributed with the SimpleScalar toolset. However, we model data movement accurately and actually pass data values down the different pipeline stages and memory hierarchy components. Also we model bandwidth and port contention at all levels. While the simulation technique used in SimpleScalar is faster than our more detailed method, SimpleScalar’s method does not lend itself well to simulating multiprocessors and our technique produces somewhat more accurate results.

Our simulator supports parallel applications written to the PARMACS macros [5] and supports both the original SPLASH’s (fork, similar to the Unix fork model) and SPLASH-2’s (sproc, similar to a light-weight thread model) programming models [30, 33]. We use the SimpleScalar supplied toolset to compile libraries and applications for our simulations (the compiler is based on gcc version 2.6.3). SimpleScalar’s ISA is very similar to the MIPS instruction set [18]. Through SimpleScalar’s annotation mechanism, we have add instructions originally not available in the SimpleScalar ISA. These instructions include Swap, Load-Linked, Store-Conditional, EnQOLB, and DeQOLB operating both on 8-bit and 32-bit quantities.

4.2. Target system

We model a bus-based multiprocessor system. An aggressive split broadcast-based snooping address bus and crossbar data bus connect the nodes of our system. Sun's Gigaplane [31] and Gigaplane-XB [7] inspired our bus design. Each node consists of a processor, split first level instruction and data caches, and a unified second level cache. All caches are non-blocking. Each processor can fetch, issue, and commit up to 4 instructions per cycle. We also model a reorder buffer, a load/store queue, a branch predictor, a branch target buffer (BTB), and return address stack.

Table 1 summarizes the parameters of our simulated hardware. We express all latencies in terms of processor cycles.

4.3. Benchmarks

We use benchmarks drawn from SPLASH-2 [33]. These applications are Barnes, Ocean (the contiguous version), Radiosity, Raytrace, and Water-nsquared. Description of these benchmarks appears in the original article. We compile all benchmarks using the SimpleScalar compiler with the option –O3. We list the problems that the benchmarks solve and the inputs that we use in Table 2.

We measure L1 data cache miss rates of <1% for Barnes, 3-10% for Ocean, 1-2% for Radiosity, 1-3% for Raytrace, <1% for Water-nsquared. These rates are very small for all benchmarks and are similar to previous published results [33]. Instruction cache misses were negligible.

Like any small set of benchmarks, little can be concluded about the generality of results. The benchmarks were chosen to demonstrate that the techniques work as well as QOLB without collocation in those applications where QOLB performs well

Table 2. Benchmarks

Benchmark	Type of simulation	Input
Barnes	Barnes-Hut N-body	2,048 bodies, 11 iter.
Ocean contig.	Hydrodynamic	130x130, 2 days
Radiosity	Light distribution	room, batch mode
Raytrace	3-D rendering	car
Water-nsquared	Water molecules	512 mols, 3 iter.

5. Results

We present the results for a 32-processor system in Table 3. We show the results for three synchronization primitives: test&test&set (TTS) implemented with the LL/SC instructions, QOLB (without collocation), and our new method IQOLB (also without collocation). All results correspond to the execution time of the parallel section.

TTS is our base case and for this synchronization primitive we show the absolute speedup in parentheses. We measure absolute speedup as the fraction of the running time on a single node divided by the running time on a 32-node system. All other numbers in Table 3 are speedups relative to the base cases.

We observe that QOLB consistently outperforms TTS. Two benchmarks (Barnes and Water) are relatively insensitive to the performance of synchronization primitives and therefore display only moderate speedups. The other benchmarks are more sensitive to synchronization performance and we measure speedups in excess of 30%.

The key result in Table 3 is that IQOLB tracks the performance of QOLB very well. In particular, although usually slower, IQOLB is never more than 2% slower than QOLB!

Table 3. Results

Synch. primitive	Barnes	Ocean	Radiosity	Raytrace	Water-nsq
TTS w/ LL/SC	(7.5)	(6.0)	(2.5)	(1.5)	(18.1)
QOLB	1.06	1.54	6.37	11.01	1.06
IQOLB	1.06	1.52	6.37	10.75	1.06

6. Concluding remarks

We have demonstrated the potential of two important mechanisms—speculation and the insertion of delays—to improve the performance of synchronization primitives without software support. Preliminary evaluations indicate that these two mechanisms combined can perform as well as, if not better than, other synchronization primitives proposed to date.

This work can be extended in multiple ways. First, we have only evaluated the two above mechanisms in the context of improving the handling of locks. Some primitives, such as QOLB, not only support the transfer of locks efficiently, but also the transfer of data associated with those locks. We believe that we can apply these mechanisms to manage protected data as well locks. In fact, we believe that these mechanisms can handle protected data better than QOLB does. Without evaluating this extension, we refer to it as *Generalized implicit QOLB* (shown at the bottom of Figure 1).

Secondly, we have only studied the use of these mechanisms in the context of bus-based systems and with the Load-Locked/Store-Conditional instructions. We believe that our schemes can be adapted successfully to support other configurations.

Finally, we have not examined attentively the interplay between our methods and weaker form of memory consistency models. Many multiprocessor systems today trade looser memory ordering constraints for performance. It would be interesting to study if weaker memory models could further improve the performance of our mechanisms.

Acknowledgements

We thank Allan Gottlieb, Gil Neiger, Eric Rotenberg, Steve Scott, and Craig Zilles for their comments. Finally, this research was supported in part by an NSF Grant No. CCR-9810114, NSF Grant No. CDA-9623632 (MIDSHIPS), and various support from Intel Corporation.

References

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

[2] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

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

[4] Philip Bitar and Alvin M. Despain. Multiprocessor Cache Synchronization: Issues, Innovations, Evolution. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 424–433, June 1986.

[5] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfield, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, New York, NY, 1987.

[6] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, Computer Sciences Department, University of Wisconsin, Madison, WI, June 1997.

[7] Alan Charlesworth, Andy Phelps, Ricki Williams, and Gary Gilbert. Gigaplane-XB: Extending the Ultra Enterprise Family. In *Proceedings of the Symposium on High Performance Interconnects V*, pages 97–112, August 1997.

[8] Compaq Computer Corporation, Houston, Texas. *Alpha Architecture Handbook, Version 4*, February 1998.

[9] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

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

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

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

[13] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[14] John Heinlein. *Optimized Multiprocessor Communication and Synchronization Using a Programmable Protocol Engine*. PhD thesis, Stanford University, Stanford, CA, March 1998.

[15] Institute of Electrical and Electronics Engineers, New York, NY. *IEEE Standard for the Scalable Coherent Interface (SCI)*, August 1993. ANSI/IEEE Std. 1596-1992.

[16] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, November 1987.

[17] Alain Kägi, Doug Burger, and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 170–180, June 1997.

[18] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Upper Saddle River, NJ, 1992.

[19] Stefanos Kaxiras and James R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 161–170, January 1999.

[20] Sanjeev Kumar, Dongming Jiang, Rohit Chandra, and Jaswinder Pal Singh. Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 23–34, May 1999.

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

[22] An-Chow Lai and Babak Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 172–183, May 1999.

[23] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, June 1995.

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

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

[26] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufman, San Francisco, CA, second edition, May 1994.

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

[28] Shubhendu S. Mukherjee and Mark D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 179–190, June 1998.

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

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

[31] Ashok Singhal, David Broniarczyk, Fred Cerauskis, Jeff Price, Leo Yuan, Chris Cheng, Drew Doblar, Steve Fosth, Nalini Agarwal, Kenneth Harvey, and Erik Hagersten. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of the Symposium on High Performance Interconnects IV*, pages 41–52, August 1996.

[32] Richard L. Sites. Alpha AXP Architecture. *Digital Technical Journal*, 4(4):19–34, 1992.

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