International Journal of Parallel Programming, Vol. 32, No. 3, June 2004 (© 2004)

Inferential Queueing and Speculative Push

Ravi Rajwar,[1] Alain Kägi,[2] and James R. Goodman[3]

Received December 2, 2003; revised January 13, 2004; accepted February 26, 2004

Communication latencies within critical sections constitute a major bottleneck in some classes of emerging parallel workloads. In this paper, we argue for the use of two mechanisms to reduce these communication latencies: Inferentially Queued locks (IQLs) and Speculative Push (SP). With IQLs, the processor infers the existence, and limits, of a critical section from the use of synchronization instructions and joins a queue of lock requestors, reducing synchronization delay. The SP mechanism extracts information about program structure by observing IQLs. SP allows the cache controller, responding to a request for a cache line that likely includes a lock variable, to predict the data sets the requestor will modify within the associated critical section. The controller then pushes these lines from its own cache to the target cache, as well as writing them to memory. Overlapping the protected data transfer with that of the lock can substantially reduce the communication latencies within critical sections. By pushing data in exclusive state, the mechanism can collapse a read-modify-write sequences within a critical section into a single local cache access. The write-back to memory allows the receiving cache to ignore the push. Neither mechanism requires any programmer or compiler support nor any instruction set changes. Our experiments demonstrate that IQLs and SP can improve performance of applications employing frequent synchronization.

KEY WORDS: Synchronization; data forwarding; inferential queueing; critical sections; migratory sharing.

[1]Microarchitecture Research Lab, Intel Corporation, Hillsboro, Oregon 97124, USA. E-mail: ravi.rajwar@intel.com
[2]Department of Computer Sciences, University of Wisconsin-Madison, Madison, Wisconsin 53706, USA. E-mail: alain@cs.wisc.edu
[3]Computer Science Department, University of Auckland, Private Bag 92019, Auckland, New Zealand. E-mail: goodman@cs.wisc.edu

0885-7458/04/0600-0225/0

1. INTRODUCTION

The shared-memory programming model is now widely established as a leading paradigm for parallel computing. The shared-memory abstraction is particularly attractive for irregular applications, where reasoning about program behavior and predicting performance may be difficult. Under the shared-memory model, in addition to holding values, memory also provides the means for synchronization and coordination of activities among processors. When multiple processors attempt to access a set of variables simultaneously and atleast one processor updates at least one of the variables, a *data race* may occur wherein the execution outcome depends on the relative speed of the operations and the result of memory accesses becomes unpredictable. The most common method used to resolve data races and to enforce mutually exclusive accesses to regions of code, known as *critical sections*, is through the use of a lock. A lock is simply a shared-memory location accessed using a software convention for implementing mutual exclusion.

Optimizing lock accesses associated with an actively shared critical section is both crucial and subtle: crucial because nave locking algorithms can lead to disastrous performance,[1–4] and subtle because multiple processors may access the lock even while that lock guarantees exclusive access to the data it protects. Since the protected data is often modified, efficient lock handling often exposes subsequent delays in accessing protected data. Numerous synchronization mechanisms have been proposed,[2,5–9] and while no common mechanism is available in all architectures, virtually all architectures provide a hardware means for acquiring a lock atomically. To date, very few implemented multiprocessor systems have incorporated hardware mechanisms for efficient locking.

A common argument against specialized hardware support claims that re-structuring of software may be an easier solution. This statement may indeed be true for many structured parallel scientific applications where critical sections can often be restructured to minimize synchronization since these applications are computationally intensive, highly regular, and display easily exploitable parallelism. However, emerging classes of parallel programs, such as online transaction processing workloads, display radically different behavior from traditional scientific applications: they are characterized by high communication miss rates.[10–12] A study of such workloads showed that a large fraction of misses are generated within critical sections; for a 4-way system running the Oracle database engine, 20% of execution time was consumed on critical section data modified in remote caches. Most of these misses targeted only a small fraction of the total number of cache lines experiencing misses.[12] A commercial

server study also noted a large portion of execution latencies spent in critical sections.[13] These emerging workloads are characterized by fine-grain updates of control data and frequent synchronization protecting such data. The protected data sets migrate among processors with the passing of the lock and contribute to a large portion of the access latencies to dirty data in remote caches.

With larger numbers of processors, faster processor speeds, and relatively increasing remote access latencies, processor stalls induced by communication misses within critical sections will only increase and processors will be unable to generate misses early enough so as to hide memory access latencies to actively shared data.

In this article, we address the problems outlined above by targeting lock operations in conjunction with data accesses protected by these locks. Thus, in addition to optimizing lock accesses, our proposal also optimizes data transfer associated with these locks. Two mechanisms discussed in this paper are Inferentially Queued Locks (IQL) and Speculative Push (SP).[14,15] Neither mechanism requires any programmer or compiler support nor any instruction set changes.

1. **Inferentially Queued Locks** By devoting hardware to build an orderly queue of lock contenders, IQLs can transfer a contended lock in a single transaction, sending the cache line containing the lock (i.e., the lock line) from the processor currently holding the lock directly to the processor requesting the lock without involving any other processors. IQLs aim to increase system throughput and reduce interprocessor communication traffic by using hardware and the cache coherence protocol to delay transfer of the lock line until after the lock is released. Such a delay allows the processor holding the lock to complete its critical section and without losing exclusive ownership of the lock line until the processor explicitly releases the lock. The selective use of delays in processing incoming coherence requests helps in automatically constructing a queue of waiting processors, each waiting its turn to acquire exclusive access to the lock. The queue is speculative because the processor infers the existence, and limits, of a critical section from the use of synchronization instructions. We believe IQLs are the first proposal to convert, automatically and transparently, software-based spin locks into hardware-based queued locks without requiring software or programmer support.
2. **Speculative Push** Speculative Push works in conjunction with IQLs to reduce the miss latency associated with data accesses within critical sections; such latency gets exposed once the locking mechanism

is optimized. SP allows the cache controller of a processor currently holding a lock not only to defer momentarily its response to a request for the lock line, but also provide additional modified cache lines by anticipating misses likely to occur immediately after the requestor has acquired the lock. Overlapping the protected data transfer with the lock transfer reduces the communication latency experienced within a critical section. To our knowledge, SP is the first hardware technique to convert data misses in a critical section resolved through multi-hop transactions into local accesses.

In Section 2 we discuss IQLs, their intuition, and show how IQLs can be efficiently supported naturally in modern systems by using existing cache coherence protocols. We then extend the coherence protocols to incorporate mechanisms for SP in Section 3. Section 4 discusses applying the IQL notion of delays to atomic read-modify-write operations in addition to locks. Results are presented in Sections 5 and 6. Related work is discussed in Section 7, while Section 8 summarizes the paper.

2. INFERENTIALLY QUEUED LOCKS

In this section, we introduce and discuss IQLs. We start by providing an intuition motivating IQLs in Section 2.1. The core mechanisms involved in IQLs are discussed in Section 2.2. In Sections 2.3 and 2.4, we discuss the implementation of IQLs for snoop- and directory-based protocols respectively.

2.1. Intuition and Motivation

Two observations about performance loss due to lock interference motivate IQLs.

1. A processor requesting a lock for purposes of acquiring the lock will likely spin-wait upon discovering that the lock is already held. The spinning action polls the lock location repeatedly waiting for the lock to be released. Briefly delaying a response to the initial request for the lock only increases the probability of finding the lock free rather than spin-waiting, thereby likely reducing total communication. The latency to acquire a held lock is optimal if such a request is serviced immediately after the lock is released.
2. Immediately servicing a lock request while the lock is held, delays the release of the lock because the processor releasing the lock must re-acquire the cache line with the lock in a writable state. This process may generate additional communication among processors

adding to the total delay. Again, delaying a response to the initial request until after the lock is released is likely to improve performance, not only by reducing total communication, but also by avoiding a delay in releasing the lock.

Modern processors with non-blocking caches support multiple outstanding requests to the memory system. Such processors use special buffers such as miss status holding registers (MSHRs) [16] to track pending memory requests. Multiprocessor systems also use such structures to buffer requests from other processors to caches lines that are in a pending state.

IQLs extend the notion of buffering external requests by applying it to cache lines inferred to contain a synchronization variable. By delaying the service for a small and bounded period, and servicing the deferred request as soon as the lock is inferred to be released, many critical sections can be fully executed and the lock released without interference from other processors. In addition, the transfer of the lock occurs directly between the two nodes involved without the coherence network being in the critical path. With frequent synchronization and migratory locks, optimizing lock accesses reduces network contention, thus having a positive effect on memory system performance.

In contrast, a conventional system without IQLs may require many additional network transactions to transfer a lock: a directory-based system may not be able to transfer the lock directly from producer to consumer without passing through an intermediate node (home), a lock release may require upgrade permissions from the directory, or a snoop-based system may see repeated lock requests. This additional traffic may artificially dilate the time a lock is held, increasing the critical path of the overall computation.

The IQL method is speculative and inferential because hardware lacks sufficient information to know how a variable that might be a lock is being employed by the software. This information is inferred from program behavior, and IQLs specifically infer acquire and release points delimiting a critical section. Since certain coherence requests are merely delayed, but eventually serviced, correct behavior can be guaranteed, and the only possible penalty for an incorrect inference is in performance.

2.2. Basic Mechanism

We begin by specifying terminology. A processor can *have* a lock in two different ways: (1) a process running on the processor has acquired a logical entity, a lock, and (2) the processor's cache has a shared or exclusive copy

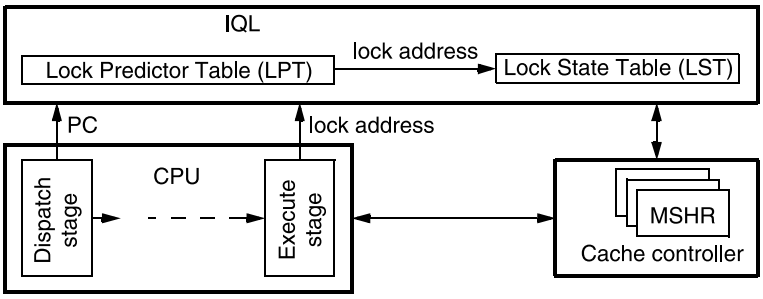

Fig. 1. IQL organization. Lock Predictor Table (LPT) and Lock State Table (LST) are the two new structures.

of a cache line containing the lock. In the first case we refer to a lock as being *acquired*, *held*, or *released*. In the second case we refer to a lock-line or lock variable as being *present*, *requested*, *sent*, or *received*. A requestor is a processor that has requested a lock-line with an inferred attempt to acquire the lock. A responder is a processor holding a writable copy of a lock-line sought by a requestor. Predicting synchronization events, and distinguishing between simple atomic read-modify-write operations and acquisition of a lock have been studied elsewhere.[14] We assume the processor, using a Lock Predictor Table (LPT) and based on previous executions of the code sequence, has predicted the event to be a lock acquire.

Figure 1 outlines the basic mechanisms of IQLs. Events involving cache lines predicted to contain lock variables invoke the IQL protocol. In addition to tracking the presence of a lock-line in its cache, the local processor must also track when it acquires and releases an inferred lock and initiate any actions triggered by these events.

A Lock State Table (LST) is used to track local information regarding inferred locks. The LST is indexed by the PC address of the synchronizing instruction identifying the critical section (Alternatively, the lock address may also be used to perform a lookup). All inferred locks known to the cache controller through the LST are in one of four states: (1) INVALID: the lock-line is not present in the cache and the local processor (probably) does not hold the lock, (2) PRESENT: the lock-line is present in the local cache but the lock is not held by the local processor, (3) HELD: the local processor holds the lock, and (4) REQUESTED: the local processor has requested the lock but does not have the lock.

To differentiate between deferrable and normal requests, a read-exclusive-ownership request (rd_X) that is deferrable is annotated as lower priority (rd_X_lp). The deferrable rd_X_lp request is used for inferred locks and ensures that the IQL protocol is invoked only for inferred

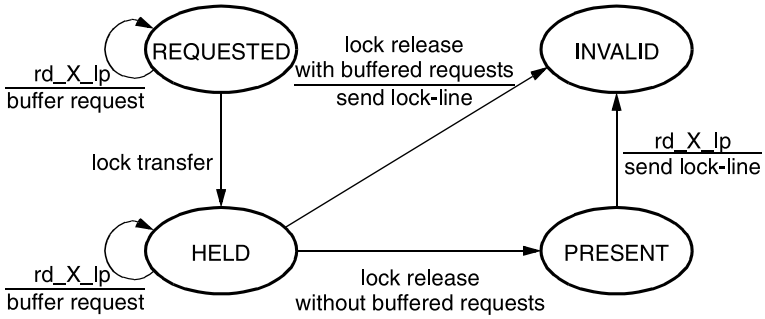

Fig. 2. LST state transition diagram. Only a subset of the transitions is shown. Arcs between states represent transitions labeled with event/action pairs. A horizontal line separates event (above) and action (below). If there is no action, the line is omitted.

synchronization operations. The rd_X_lp request can be deferred for a brief but bounded time. Otherwise it is identical to rd_X.

Figure 2 shows a simplified state transition diagram for the LST. When a processor predicts a lock-acquire, a lock-line already present in the local cache is marked HELD in the LST; otherwise space is allocated in the cache, a rd_X_lp is issued, and the corresponding LST entry is marked REQUESTED. When the request is eventually serviced, the LST entry transitions to HELD. A subsequent write to the byte address inferred to be a lock (and thus predicted to be a lock release) results in the LST entry being set to PRESENT. A lock-line's eviction or invalidation from the cache results in its corresponding LST entry being marked INVALID. The LST is consulted on any incoming rd_X_lp request. If the LST entry for the requested cache line is in state PRESENT, the request is handled as any other read-for-exclusive (rd_X) request. If the cache line is in state HELD or REQUESTED, the line is marked for special action upon the release of the inferred lock, and the request is buffered in an MSHR. A subsequent inferred release of the lock triggers the servicing of the buffered rd_X_lp.

2.2.1. Queue Formation

Since cache coherence protocols already serialize rd_X requests on a per-block basis, a queue of requesting processors is easily constructed with minor support from the coherence protocol. The first rd_X_lp request transfers the apparent owner of the lock-line to the requestor, thus making it the recipient of any subsequent request. By this means, a queue of requesting processors is formed, with each processor receiving the lock-line sequentially in the order of original requests.

2.2.2. Queue Breakdown

Regular priority reads, whether for exclusive or shared, are handled separately. Interactions between regular- and low-priority requests only occur in rare cases where the lock protocol is being violated, or due to false sharing, or where the hardware has otherwise misspeculated. A regular request may be serviced by a cache line owner with little delay. Depending upon the cache coherence protocol, such a request may result in squashing other outstanding rd_X_lp requests thus breaking the queue down. Such a breakdown can be prevented if the coherence protocol allows a requestor to insert itself at the head of the queue without breaking the queue down. When this occurs, the head of the queue sends data along with a special marker indicating that the requesting processor is temporarily at the head of the queue and the requestor must transfer ownership back immediately once the write completes. Such an action also allows for a regular read request to be serviced by temporarily making the reader the head of the queue and then immediately revoking permissions from the reader once the read is completed. The idea of temporarily providing data to a requestor with the understanding the data will be discarded after one use is referred to as providing a *tear-off* copy of the data.[17] Not all coherence protocols may support such actions. We do not believe such interactions between regular and deferrable requests to be common. Further, the unavoidable breakdowns may result in additional traffic but most probably much less than a baseline scheme without such implicit queue formation support.

2.2.3. Time-Outs

A time-out at the head of the queue forwards the line 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. An eviction is treated as a time-out. While the notion of time-outs is not attractive, the time-out mechanism is fairly simple and easy to apply. The processor which has deferred a request longer than the time-out value, will simply service the request, and this time-out helps in bounding any delays and ensuring processors do not wait too long.

While multiple critical sections can be tracked, applying deferrals to more than one cache line need not be supported. Our proposal is aimed at the lowest-level critical section. Thus, if a second nested critical section is encountered, the first can generally be discarded with respect to speculation. Thus, only a single time-out needs to be maintained. Of course, multiple line deferrals may be supported if necessary.

2.3. IQLs and Snoop-Based Systems

Traditional snoop-based systems use a logical serialization point for all coherence requests, and all processors are assumed to observe the requests in the same order. The logical serialization point is a bus. While traditional snoop systems have implemented buses by simply using physically shared wires, modern systems emulate the logical property of a bus but use more complex scalable interconnects that are not physically a bus. For our discussion, we assume a snoop design similar to the Sun Gigaplane.[18,19]

The protocol uses a split-transaction, pipelined address bus with support for a large number of outstanding transactions and out-of-order responses. The bus implements an invalidation-based three-state (Owned, Shared, Invalid) snooping cache coherence protocol and the caches implement a MOESI protocol. The cache with the requested line in Owned state (as seen by the bus) will respond to the next request for that line. IQLs can be supported naturally with the above protocol as the protocol already has an implicit notion of queues. Any processor placing a rd_X_lp on the bus will respond to the next processor which places a rd_X_lp for the same cache line on the bus. With IQLs, the request is not serviced until a lock release. By placing a rd_X_lp on the bus, the processor joins the queue of lock requestors at the tail and is serviced when the preceding requestor releases the lock.

2.4. IQLs and Directory-Based Systems

Directory protocols allow cache coherence to scale to many processors. Directory protocols store a directory state for each memory block currently cached in any node. Commonly, directory protocols implement an invalidation-based scheme. With directory protocols, state information for a line is obtained from a directory through network transactions, and communication with various cached copies is performed by explicit messages using an arbitrary network. Typically, information for a line can be found in a fixed directory location known at the time of the request.

Two popular approaches distribute directories either with memory or with caches. Memory-based schemes store directory information for a cache line at the home node of the lines.[13,20,21] In cache-based schemes, the sharing information is distributed among the various copies (rather than at the home node). Each cache line contains a pointer to the node with the next cached copy of the line in a distributed linked-list organization.[22–24] For cache-based directories, IQLs can be supported relatively easily since there already exists a notion of queues for cache lines. We focus our discussion on memory-based directories.

We use the SGI Origin-2000 coherence protocol [20] to discuss our IQL implementation. The protocol supports the MESI (Modified, Exclusive, Shared, Invalid) states and is non-blocking: memory does not buffer requests while waiting for other messages to arrive. The protocol also supports request forwarding for three party transactions and silent evictions of clean-exclusive lines. The protocol does not rely on an ordered network. Two virtual channels are provided and deadlock in the request network (due to request forwarding) is broken using back-off messages. Memory is the owner for all clean lines in the system; thus memory services any request for clean data immediately. In addition, rd_X requests cause transfer of exclusive ownership to the requestor and sending of invalidations to other holders of cached copies. The holders of cached copies subsequently send invalidation acknowledgments to the requestor. Requests to lines not owned by memory are forwarded (as an intervention) to the owner (and in the case of a rd_X request, the requestor becomes the owner). The directory enters a transitional Busy state for the particular memory block until it receives a revision message from the previous owner. All requests received by the directory for a memory block in Busy state receive a Nack suggesting they retry later. A sharing bit-vector associated with each cache line identifies the processors holding shared copies of the line.

Under the base protocol, the directory enters a transitional Busy state while tracking down an exclusive copy in the system. For IQLs, the directory, instead of sending a Nack to a new request while in Busy state, forwards the request to the previous requestor. If the previous requestor's request is still pending, the intervention is buffered. Forwarding a request to the last requestor guarantees a processor receives at most one intervention for a given memory block. An additional bit, synch_bit, is used per directory entry to determine whether the IQL protocol should be invoked. The owner pointer is overloaded to store the last requestor when the synch_bit is set. The sharing bit-vector is also used to track members of the inferred lock requestor queue. Alternate ways to capture such information include using unused state bits in the encoding.

To understand how queues are created we step through a simple example in Section 2.4.1 and then discuss how the directory detects and handles queue breakdowns in the case of write-backs in Section 2.4.2.

2.4.1. Constructing Queues

Consider three processors, P1, P2, and P3 attempting to enter a critical section. P1 issues a rd_X_lp to the directory. Since the directory exclusively owns the line, it responds to the request with data and enters an Exclusive state. P1 is now the owner of the line. Subsequently, P2 issues a

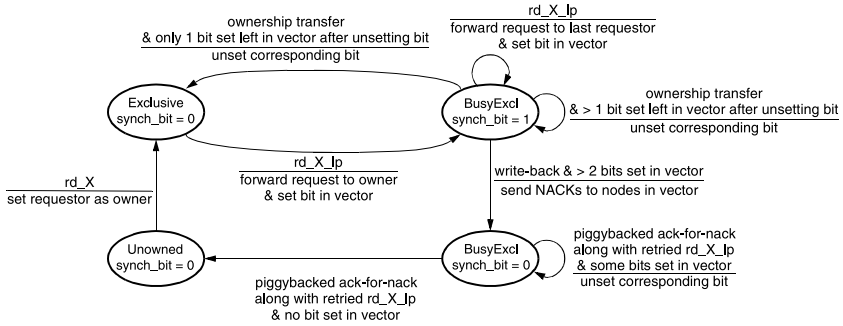

Fig. 3. Protocol transitions for IQLs in a directory-based system. Only a subset of the transitions is shown, this subset focuses on the process of forming a queue. Arcs between states represent transitions labeled with event/action pairs. A horizontal line separates event (above) and action (below). The queue-breakdown sequence is shown when a write-back occurs.

rd_X_lp request to the directory. The directory forwards P2's request to P1, marks P2 as the last requestor, and enters the Busy state. The bit vector now has 2 bits set: P1 and P2. In addition, synch_bit is also set. Now, P3 sends a rd_X_lp request to the directory. The directory state is Busy. However, since synch_bit is set, instead of sending a Nack to the new requestor, the directory forwards it to the last requestor, P2. Also, P3's bit is set in the bit vector. Thus, we have P3 waiting for P2's response, which is waiting for P1's response. A processor buffers an intervention until the lock is released. At the time when the intervention is serviced, a revision message, which is already part of the base protocol, is sent to the directory. On receiving the revision message, the directory unsets the processor's bit in the bit vector. If only one bit is set in the vector, the last requestor automatically becomes the owner. Under this situation, the directory unsets the synch_bit and leaves the Busy state, entering the Exclusive state. A revision message exists for every rd_X_lp request serviced, thus ensuring the directory will eventually transition into an Exclusive or any other stable state. A simplified state transition diagram for the IQL protocol is shown in Fig. 3.

2.4.2. Handling Queue Breakdown Due to Coherence Protocol Events

The directory does not track the order in which requests are received but only marks who has requested a cache line. The actual order of the queue is maintained in a distributed manner among the various caches involved. The distributed nature of the IQL queue may cause the queue to

break-down due to write-backs. For the example in the previous section, suppose P1 is writing back the line to memory. Under the base protocol, P1 can ignore P2's intervention since the directory can recognize this race condition and detect P2 will not be serviced by P1. The directory does this as it keeps track of the owner P2. However, under IQL, if there are multiple bits set in the bit-vector (the synch_bit is set and a queue exists), the directory cannot determine the identity of the processor which will not receive a response from P1; the directory does not remember it forwarded P2's request to P1. While this race condition is indeed rare, it must be handled correctly, if not efficiently.

We adopt a simple approach to handle queue breakdowns. On receiving a write-back to a line with synch_bit set and more than two bits set in the bit-vector (if only two bits are set, the directory can uniquely determine the processor that will not receive a response), the directory unsets synch_bit—the directory is breaking down the queue. Doing so, the protocol behaves like the base protocol with an additional side effect: Nacks are sent to all processors in the bit-vector. When the processors retry on receiving the Nack, the directory can detect this by a bit in the retried message. At that point, the directory unsets the bit in the bit-vector corresponding to the requestor. When the bit-vector has no more bits set, the directory entry enters an Unowned state. Such a mechanism guarantees a directory will eventually transition into an Unowned state.

With conceptually simple changes to the way the directory protocol works for certain types of requests, and some additional bits in the directory entry, IQLs can be efficiently supported in an SGI Origin 2000-style protocol.

3. SPECULATIVE PUSH

This section discusses SP. Section 3.1 presents the intuition behind SP and Section 3.2 outlines the three basic mechanisms for SP: predicting lock and data pairings, assigning confidence to this prediction, and the actual SP protocol. These mechanisms are then discussed in Sections 3.3, 3.4, and 3.5 respectively.

3.1. Intuition and Motivation

The performance of any scheme optimizing data transfer within critical sections depends on the accuracy of correctly predicting which processor will acquire the lock and use the data. IQLs have the advantage of early, accurate knowledge of the next owner of a lock. Since IQLs allow inference of the presence and extent of critical sections in programs,

the data set information associated with these critical sections can be tracked. Using this information, SP forwards the actively shared data to the requesting processor, along with the lock. On subsequently acquiring the lock, the requestor finds in its cache the data it was unable to prefetch. SP forwards data in an exclusive state, thus allowing the processor to modify the cache line without experiencing a further delay it would otherwise suffer if it had initially retrieved the line for reading.

For critical section accesses, SP has inherent advantages over more traditional approaches for latency reduction such as prefetching and compiler-assisted data forwarding. In situations of contended locks, prefetching is not sufficient as the processor would spin-waiting for the lock and would generate data requests only once the lock has been acquired. SP transfers data as soon as the data is ready to be forwarded and does not interfere with the execution of the processor performing the push. The speculative mechanism can also adapt to run-time behavior. SP provides additional performance gain for data that is read before being written to: its initial access is overlapped with the lock transfer, and it does not have to be upgraded for writing.

We restrict SP to modified lines because, while some shared lines may also result in misses, such shared lines likely would already be present in the requestor's cache—data in cache lines previously read but not modified by the requestor in a previous execution of the same critical section would probably still be in the cache. Indeed, our experiments suggest that much of the observed benefit is derived simply from pushing cache lines into caches where they were present and modified in an earlier execution.

3.2. Basic Mechanisms

Speculative Push (SP) aims at constructing a link between critical sections and the data they protect. Once the link is established, whenever a request for a lock-line is received, SP also forwards any predicted data to the requesting processor along with the lock-line. This allows the lock requestor to find both the lock and any protected data in writable state in its local cache upon entering the critical section. The requestor experiences minimal delay in executing its critical section, virtually eliminating latency associated with access to data protected by migratory locks.

SP may initiate data transfer of predicted critical section data even before the requestor has acquired the lock, and the data is forwarded in Exclusive (as opposed to Modified) state. The pushed data is also written back to memory, providing two key benefits: (1) if the target node does not reference the pushed data, the data can be silently evicted, thus doing

no harm, and (2) the target of the push now has the important option of ignoring the push if no local buffer space is available.

In our experiments, we were conservative in accepting the push—a pushed cache line never evicted a valid cache line, that is, a push was accepted only if an invalid line was available. This approach is surprisingly effective, in part because migratory lines that are frequently written are frequently invalidated, leaving available invalid lines in the corresponding cache sets. Even if the push is rejected, benefits accrue for some systems, particularly directory-based ones. Specifically, when the rejected cache line is subsequently accessed, it can be supplied from the directory, avoiding the three-hop latency that would have occurred without the attempted push.

As shown earlier in Fig. 1, IQL has two structures: the LPT to infer critical sections, and the LST to invoke the IQL protocol partly based on information received from the LPT. Figure 4 shows the SP hardware. SP extends the LST by recording data accesses while the inferred critical section is executed and by deciding if predicted data is to be forwarded along with a released lock.

The basic steps of SP are:

1. Establish and record the association between critical section data and a lock.
2. Enable (or disable) the optimization by assigning a confidence level to pairings determined in Step 1.
3. Perform the Speculative Push.

The following three subsections discuss the above steps in detail.

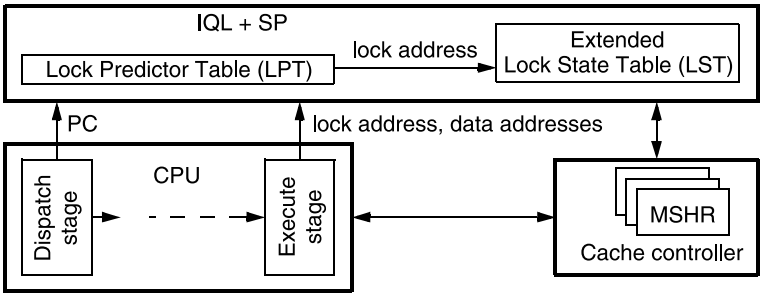

Fig. 4. SP hardware organization.

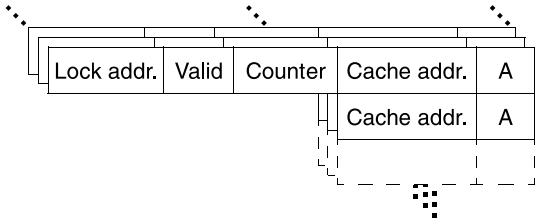

Fig. 5. LST entry extended for SP.

3.3. Prediction Lock and Data Association

When IQL infers a critical section, the SP hardware starts recording addresses of accesses performed while the processor holds the lock. These addresses become candidates for forwarding and are stored in an extended LST entry shown in Fig. 5. Besides the lock address, each extended LST entry stores a valid bit, a saturating counter to establish a confidence level, and data addresses with associated access bits (A).

The LST stores cache-aligned addresses to match SP with existing controllers and existing hardware manipulates data at the granularity of a cache line. All data in a cache line are assumed to be either related or unused since false sharing with critical data is considered poor programming practice. SP may record either individual addresses or address ranges. In our experiments at most two cache lines generally caused a write-miss during the execution of a critical section. Therefore, our LST stores individual addresses.

When an access results in a write fault, the SP hardware allocates a new entry for the address. If no free entry is available, the entry with the lowest confidence is evicted (discussed in Section 3.4). Ties are broken arbitrarily. When a lock request is received, if the lock is predicted to be currently held by the local processor, the push occurs when the processor releases the lock, else if the lock is predicted to be present in the cache but not held, the push occurs immediately.

Two actions identify candidates for future pushes: lines that caused write misses during a critical section and lines accessed during the critical section that have previously been pushed into the cache (and therefore do not cause write misses). While not all data written in a critical section is migratory, write misses capture data written within the critical section and obtained from the memory system. If such misses occur repeatedly, one can speculate the data is migratory. Local private data may also cause a write miss but is more likely to remain in the cache, since it will not be

invalidated by another processor. Of course, various critical sections may touch many or few cache lines, but the first lines touched after acquisition of the lock seem especially important.

3.4. Prediction Confidence

A saturating counter is added to each LST entry data address to assign confidence in the data for forwarding. Also associated with each candidate address is an access bit (A) that is set each time the critical section accesses this address and cleared each time after the execution of a presumed critical section ends. Before the access bits are cleared, SP inspects them and increases the counter for each set bit or decreases it for each cleared bit. A counter reaching the maximum value enables SP for the data address. This scheme avoids a cache line being repeatedly pushed but never written to.

Repeated evictions of a candidate address also causes the counter to decrement. Repeated evictions is a sign that the addresses accessed inside a critical section vary from one execution to the next, preventing effective data forwarding. A counter reaching the minimum value disables the optimization.

More sophisticated predictors, for example using information about remote requests for modified data in the local cache, or combining collective information from multiple nodes about migratory patterns, may improve the effectiveness of SP.

3.5. Speculative Push Protocol

Once the data association has been identified, the SP protocol is initiated. The SP sequence involves two separate actions to maintain memory ordering requirements and achieve high performance:

1. Determine the forwarding data path for the push.
2. Insert the Speculative Push in the global memory order, thereby granting coherence permissions to the target.

We discuss these two actions in Sections 3.5.1 and 3.5.2. In Section 3.5.3 we discuss an interesting approach to match up speculatively pushed data messages with their appropriate coherence permission messages correctly. The issue arises because no ordering guarantees are assumed from the network or coherence protocol and we demonstrate a simple solution to handle the absence of any ordering guarantees.

3.5.1. Determining the Forwarding Data Path

The predicted data, present in the local cache in modified state, can be pushed by the initiating processor either directly to the target processor (with a write back also sent to memory) or pushed via memory (the data is written back to the directory which will forward the data to the target node). If the data is pushed via memory, the initiating processor sends address hints to the target node informing the target to expect unrequested data. The target processor in this case pre-allocates data buffers in anticipation of the pushed data. Doing so prevents the target processor from generating unnecessary requests for data addresses which are going to be pushed along with the lock. If data is pushed directly to the target processor, the target processor receives the actual data rather than hints and allocates cache lines to sink the data, if possible.

When data is directly pushed to the target node, the recipient cannot commit the use of the data until the push has been ordered because the SP is initiated on a network separate from the one used for enforcing serialization (at the snoop network or directory). Early access may still be beneficial, for example, if data value speculation is being performed in the critical section—data is present in the cache and most probably will be a valid copy.

The choice of the forwarding data path implementation depends, among others, on the coherence protocol. For example, for snoop-based systems, pushing data directly to the requestor (instead of via memory) may be beneficial. Modern systems are designed to make snoop bandwidth the performance limiter rather than the data network. Thus, the data network can be utilized to overlap the data transfer with the latency of ordering the SP. On the other hand, for directory systems, address hints may be sufficient since the data has to first go to the directory for ordering.

For our experiments, the data is sent directly to the target node for the snoop-based protocol, while it arrives along with the coherence permission in the directory-based scheme (with hints being sent when the lock request is received).

3.5.2. Ordering the Speculative Push

The precise implementation for correctly ordering (i.e., inserting into the global memory order for correct memory consistency and coherence) the SP depends upon the base coherence protocol. We discuss two coherence protocol classes here and show how a Speculative Push is ordered.

Snoop-based systems. For a typical bus-based protocol, the pushed data could be broadcast once on the bus, with a special annotation allowing the target node to capture the data as it was being written back to memory. In our example bus system, since data is transmitted point-to-point, a write-back operation requires some care to ensure that the data is received, and ordered correctly, at the target node.

The data would normally be pushed immediately after the response providing the lock. To serialize the push, an annotated write-back (the annotation identifying the target of the Speculative Push) is sent to memory and serialized in the global memory order. When the annotated write-back appears on the bus the target receives coherence permissions implicitly. The data may or may not have reached the target node at this point. Since the bus provides an implicit ordering, in many cases, the latency can be overlapped to a larger extent than in a directory system.

Directory-based system. Since the data is also being written to memory (to allow the receiving node to ignore the push if necessary), this 3-way communication presents ordering challenges in a directory-based system. In a directory-based protocol, the directory node must be involved if, as usual, it is the point of serialization for operations. Responding to the annotated write-back, the directory node communicates with the target node, granting exclusive coherence permission or sending a Nack to the target node if necessary.

While the directory node must be involved in all data forwarding to guarantee proper memory semantic, this perceived disadvantage has the benefit of solving all the race conditions that might occur. In effect, having to include the directory node in all transactions has the property of not adding any new race condition that the directory-based protocol must not already handle. Thus, receiving an annotated write-back is really no different than receiving data evicted from a node's cache. The only difference is that SP requests the directory to forward a copy of the data to the target, an action that the directory can decline to do if necessary.

For directory systems, latency is not completely overlapped since the coherence permissions need to come via the directory. However, the write-back to the directory is overlapped with the lock transfer to the target node. In addition, we are sending address hints to the target node. Doing so allows the target node to pre-allocate local buffer space for sinking the push. The only exposed latency left is the directory lookup and transfer of coherence permissions to the requestor. In our experiments, while a three-hop read took about 360 ns, with the Speculative Push, the latency observed by the target node substantially reduces to about 60 ns. In addition, the upgrade traffic that follows after the read of the data is also eliminated since the data is being sent to the target in exclusive state.

3.5.3. Automatically Matching Multiply Pushed Data with Coherence Permissions

The two actions of speculatively pushing data, and ordering the speculative push in the global memory order can occur in any order and different networks may be used for them. In addition, multiple pushes of a cache line to a given processor may occur since no ordering guarantees are assumed from the network. To handle such situations, we treat the two actions symmetrically. If a push is rejected, the corresponding coherence permission must also be rejected and viceversa. Bookkeeping is necessary to track multiple cache lines to ensure consistent responses to both actions. For generality and without assumptions about the network, we require a requestor to include in a rd_X_lp request an indication of the number of lines it can track (but not necessarily sink) at any given time. This number could be quite small.

The push/coherence permission information is stored in a small table at the cache controller. Both messages in the pair will occur exactly once, so every push (irrespective of address) received is tracked until its corresponding coherence permission is received, and vice versa. An entry is removed when the pair is matched up. Any push reject can be matched with any coherence reject and so on. As the mechanism is speculative, the push and coherence permissions may arrive as a result of two different attempted pushes. Nevertheless, it is not difficult to guarantee that the processor will have the latest data if the two actions are matched correctly.

A third processor may attempt to read data in a cache line while the line is being pushed, either due to data races or due to misspeculated pushes. The situation is handled efficiently by providing the third processor with the data and conservatively cancelling the push to the target node (which, for directory systems, may require an additional message to be sent to the target node).

4. IQLs AND ATOMIC READ-MODIFY-WRITE OPERATIONS

IQL discussion so far has focused on lock-based synchronization mechanisms. Another class of synchronization primitives commonly used include atomic read-modify-write operations such as Fetch & Φ. These primitives provide the ability to perform a simple operation—usually an arithmetic addition or an increment to a variable in a 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.[25] While

most modern processors do not support combining, many contemporary processors provide instructions to perform atomic read-modify-write operations to cached memory locations. These instructions include load-linked/store-conditional, compare&swap. Such instructions are also useful for enqueues, dequeues, software barriers, and ticket lock implementations in addition to providing efficient ways to implement software barriers. This section demonstrates the application of inferential delays to atomic read-modify-write operations using the load-linked/store-conditional (LL/SC) primitives.[14]

4.1. Load-Linked/Store-Conditional Instructions

The LL/SC instructions were originally proposed by Jensen, Hagensen, and Broughton.[26] 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 the value from a memory location into a processor register. This instruction is followed by an arbitrary sequence of operations involving the register. The SC then attempts to write to the same memory location as the previous LL operation. The SC succeeds 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 case of failure, the entire sequence may be retried.

The LL instruction itself may read data for exclusive ownership. However, 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 (thus forcing P1's SC to fail) 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.

4.2. Optimizing LL/SC Sequences by Using Inferentially Delayed Responses

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 ensure 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 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 on to 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.

An extension thus involves requesting 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. This scheme allows a processor to perform an atomic memory operation in a single network transaction most of the time.

The above should provide an intuition as to why the principle of delayed responses can also be applied to atomic read-modify-write sequences, similar to the applications discussed earlier for lock operations. Further details of the treatment of atomic read-modify-write operations, and implementation details can be found elsewhere.[14] Importantly, a distinction between LL/SC use for implementing locks, and LL/SC use for implementing only atomic read-modify-write operations must be made and we discuss the issues arising from such distinction and potential solutions elsewhere.[14]

5. EXPERIMENTAL METHODOLOGY

We use an execution-driven simulator to perform cycle-by-cycle simulation of an out-of-order processor and a detailed event driven simulation of the memory hierarchy. The simulator models all data movements accurately (in the pipeline as well as in the memory hierarchy) and models port contention at all levels. The processor implements a release consistency memory model [27] similar to the Compaq Alpha 21264.[28] The processor retires stores to a coalescing write buffer.

Not all benchmarks display the behavior of high communication miss rates we are targeting. We specifically select four benchmarks (Table I) that frequently experience communication misses within critical sections. A locking version of *mp3d* is used to demonstrate the effectiveness for applications with frequent synchronization. The test&test&set synchronization primitive is used and is implemented with LL/SC instructions.[25]

Table I. Benchmarks

Applications	Application type	Input	Procs
Cholesky (Splash)	Sparse matrix factorization	tk14.0	16
MP3D (Splash)	Rarefied fluid flow simulation	24,000 mols, 25 iter.	4
Raytrace (Splash-2)	3-D rendering	teapot	8
Water-Nsq (Splash-2)	N-body molecular dynamics	512 mols, 3 iter.	16

5.1. Target Systems

We simulate two systems: a snoop-based symmetric multiprocessor (SMP) and a directory-based distributed shared-memory (DSM) systems. The SMP system is modeled after the Sun Gigaplane:[18] coherence and data traffic is split onto two separate networks. Address requests and associated coherence operations take place on a high bandwidth snoop bus; while a high-speed point-to-point crossbar transfers data among the nodes. The SMP employs a coherence protocol similar to the one used in the Sun Enterprise 10,000 system.[19] The DSM implements a MESI cache coherence protocol similar to the SGI Origin 2000 system.[20] We assume a fully connected, point-to-point network in which the messages take a constant latency to traverse one hop. However, port contention is accurately modeled. Table II list the parameters of the integrated processor and cache

Table II. Integrated Processor and Cache Subsystem

Processor	
Processor speed	1GHz (1 ns clock)
Reorder buffer	64 entry with a 32 entry load/store queue
Issue mechanism	out-of-order issue/commit of 4 ops/cycle, 64 entry return address stack, aggressively issue load (∼MIPS R10,000)
Branch predictor	8-K entry combining predictor, 8K entry, 4-way BTB
Cache	
L1 instruction cache	64-KByte, 2-way associative, 1-cycle access, 8 outstanding misses
L1 data cache	128-KByte, 2-way associative, write-back, 2 ports, 1-cycle access, 8 outstanding misses
L2 unified cache	2-MByte, 4-way associative, write-back, 10-cycle access, 16 outstanding misses
L1/L2 bus	Runs at processor clock
Line size	64 bytes

Table III. External Network and Memory Configuration

DRAM memory module	8-byte wide, ~70 ns access time for 64-byte line
Snoop-based configuration	OSI protocol on address bus modeled after the Sun Enterprise 10000, MOESI at snoop cache
Address bus	split-transaction, out-of-order responses, 120 outstanding requests, 22 ns snoop cycle (including 2 ns arbitration)
Data network	pipelined, point-to-point crossbar, 64-bit wide, 80 ns transfer latency
Some uncontended latencies	(pin to pin) read miss to memory: ~172 ns, read miss to another cache: ~125 ns
Directory-based configuration	SGI Origin-2000 based MESI protocol
Directory access	70 ns (overlapped with memory access)
Processor and local directory	30 ns (directory is integrated with memory and network controllers, point to point)
Directory and remote router	50 ns (point to point)
Some uncontended latencies	(pin to pin) read miss to local memory: ~130 ns read miss to remote memory: ~230 ns, read miss to remote dirty cache: ~360 ns
Network configuration (DSM only)	pipelined point-to-point network
Network width	64 bits
Read latency from network to cache	1 ns per word
Setup latency for header packet	5 ns
Setup latency for data packet	5 ns + 1 ns per word

subsystem used in both SMP and DSM systems and Table III lists the parameters for the SMP and DSM memory systems.

5.2. Explanation of Metrics

Speedup is measured as the ratio of the parallel execution time of the base case to the execution time of the optimized case. Therefore, a speedup greater than one implies a performance gain. Attributing stall cycles to specific components is a complex task for multiprocessor systems with out-of-order processors where many events occur concurrently. We use an approximate approach. For every cycle, we compute the ratio of instructions committed that cycle to the maximum commit rate. This fraction of cycle time is attributed to the busy time for the processor. The remaining fraction is attributed as stall time to the first instruction that

could not be committed that cycle. The fractions of stall cycles are normalized to the running time of IQL-only case. The stall categories are:

- *WMB*: stall at write-memory barrier
- *MB*: stall at memory barrier
- *C SECTION*: stall associated with data accesses within a critical section
- *MEM ACC*: stall associated with shared-memory accesses outside critical sections
- *CPU*: the remainder of useful cycles while the processor is busy computing

The shared-memory accesses do not include lock variable accesses. This is done specifically to avoid any bias in latency accounting due to differences in synchronization primitives.

6. RESULTS

Section 6.1 presents performance of IQLs and SP. Section 6.2 presents stall characteristics of SP and Section 6.3 provides statistics regarding push behavior. We compare SP to another technique for reducing communication latencies—flushing data to memory—in Section 6.4.

6.1. Performance

Table IV presents the main results. The numbers in parentheses show the running time in millions of processor cycles. All the other numbers in the table represent speedups. SP does not hurt performance for most of our benchmarks. The exception is for `cholesky` running on a DSM system where its performance drops by only 1%. SP helps `mp3d` and `raytrace` substantially compared to the IQL-only case, speeding up their performance by a factor of up to 1.51. The results for the SMP system show the same trends but have different magnitudes. In particular, SP speeds up `mp3d` by 21%, but slows down `cholesky` by 2%. The exception is `raytrace`, which displays an improvement of only 3%. The latencies in the SMP system are such that the locking behavior of `raytrace` does not affect its performance much. `Water-nsq` communicates comparatively far more infrequently than the other benchmarks do (see Figs. 6 and 7), but nevertheless still suffers some migratory data-related stalls and benefits, if little, from SP.

Table IV also compares the performance of one system implementing IQL only and the other one without it (i.e., programs use test&test&set built with load-linked and store-conditional instructions but otherwise run

Table IV. IQL and SP Performance for a DSM System

	Cholesky	MP3D	Raytrace	Water-Nsq
Without IQL (base)	(11)	(373)	(121)	(18)
IQL	1.13	1.21	2.75	1.04
IQL+SP	1.12	1.60	4.15	1.11

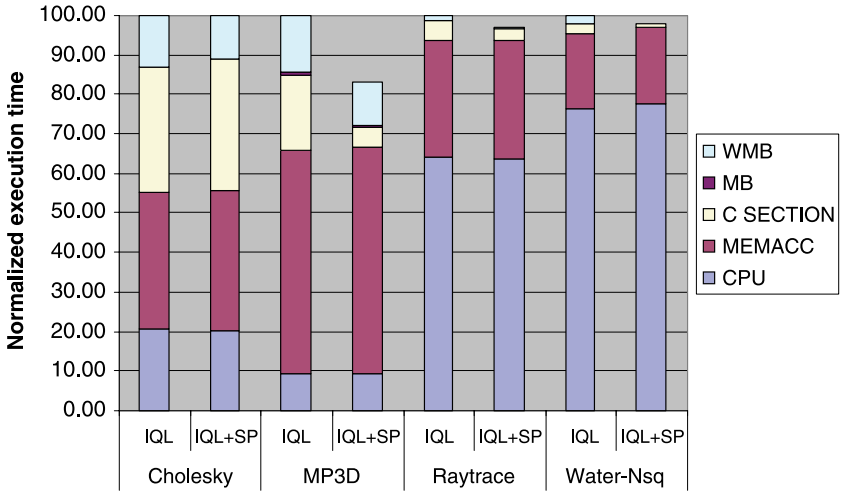

Fig. 6. SMP stall contributions. Normalized stall contributions are expressed as percentage of the IQL running time for the specific contribution.

without hardware support). The results are impressive: IQL improves the performance of all our benchmarks and, in particular, reduces the running time of `raytrace` by more than half. These results follow a trend similar to numbers published elsewhere using Queue-On-Lock-Bit as the primitive and SCI as the coherence protocol.[3]

6.2. Speculative Push Stall Cycles Breakdown

Figures 6 and 7 shows how the stall cycles are attributed to the different components of the system. These figures show that most of the performance gains stem from the reduction of the following two components: C SECTION and WMB. SP is able to eliminate nearly all latencies associated with loading shared-memory locations. This observed behavior reduces the stall cycles associated with critical section execution

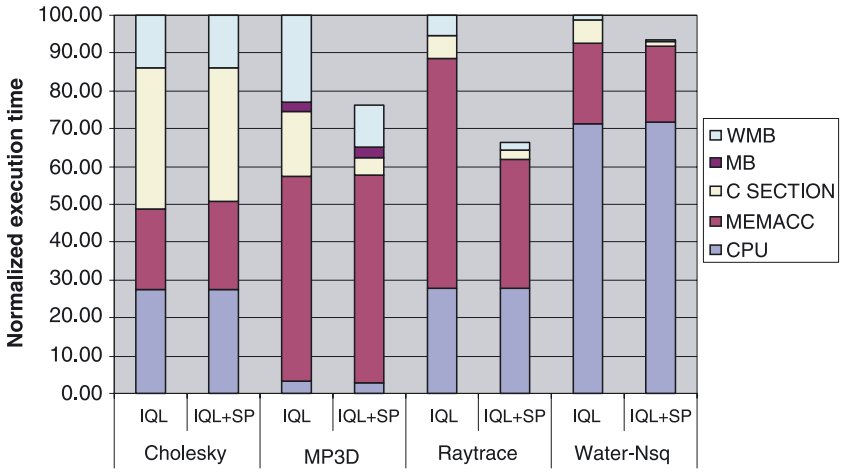

Fig. 7. DSM stall contributions. Normalized stall contributions are expressed as percentage of the IQL running time for the specific contribution.

(C SECTION). SP is also able to reduce considerably the wait time at write memory barriers (WMB) located at the end of each critical section. In our experiments, write memory barriers are almost always strictly confined to the end of a critical section and are there to ensure all the memory operations effected inside the critical section have performed globally before releasing the lock.

6.3. Speculative Push Characteristics

Table V breakdown Speculative Pushes into four categories: (1) Used: the pushed data was accessed, (2) Evicted: the pushed data was evicted before being used, (3) Rejected: the pushed data was rejected, and (4) Invalidated: the pushed data was invalidated by another processor before being used. For the most part SP performs well: more than 70% of all pushes are useful for the execution of `mp3d`, `raytrace`, and `water-nsq`. The exception to this trend being `cholesky`. `Cholesky` is the control benchmark and its critical section behavior does not lend itself to the type of optimization we are studying in this paper. Indeed, `cholesky` rarely suffers write-faults on the same address on successive executions of the same critical section. For this benchmark, little correlation exists between a lock address and data addresses accessed while the lock is held, resulting in little benefit for this benchmark (Table IV) and many evicted pushes (Table V). Our predictor detects this patterns and turns off the SP optimization, minimizing the degradation.

Table V. Breakdown of Push Characteristics (Shown as Percentage of Pushes)

	Used		Evicted		Rejected		Invalidated	
	SMP	DSM	SMP	DSM	SMP	DSM	SMP	DSM
Cholesky	8.00%	12.02%	92.00%	88.98%	0.00%	0.00%	0.00%	0.00%
MP3D	99.46%	99.56%	0.43%	0.40%	0.09%	0.04%	0.02%	0.00%
Raytrace	80.80%	71.64%	16.55%	28.33%	2.64%	0.00%	0.01%	0.03%
Water	93.80%	99.61%	6.20%	0.08%	0.00%	0.31%	0.00%	0.00%

We also varied the number of cache lines the SP mechanism could forward with a lock. For our benchmarks, we found most of the benefits could be had with a single cache line. Adding a second cache line improved the performance of SP by no more than 2% and more than two cache lines lead to insignificant improvements. Obviously, this observation might vary significantly for other benchmarks.

6.4. Speculative Push Vs. Flush

We compare the performance of SP against a technique that consists of flushing data back to memory at the end of a critical section. The basic idea is to avoid the penalty of accessing remote dirty data and instead to attempt finding the desired data at memory directly. To implement the flushing mechanism, we rely again on our predictors to identify critical section boundaries and the data set associated with a lock. Upon receiving a lock request, we write the data back to memory only unless the lock is still held. If the lock is still held, we wait until the lock is released to write the data back to memory.

We ran experiments both with the SMP and DSM configurations. We found the flush technique to be highly ineffective in our simulated SMP system (and do not show the numbers) because our assumptions are such that the transfer latency to and from memory is much larger than the transfer latency between two caches. Thus flush techniques and self-invalidation techniques will degrade performance of SMP systems.

Table VI contrasts the performance of our two DSM variants: flush to memory and SP. We observe that, for our set of benchmarks, flush achieves a smaller speedup compared to SP. These results also show that flush is able to achieve performance gains over the IQL-only case. However, flush either requires special instructions and recompilation or requires a predictor, a unit that it shares with the SP mechanism.

Table VI. SP and Flush Performance for DSM

	Cholesky	MP3D	Raytrace	Water-Nsq
IQL+Flush	1.00	1.13	1.28	1.04
IQL+SP	0.99	1.32	1.51	1.07

7. RELATED WORK

A software technique for eliminating latencies associated with critical data accesses involves collocating lock and data in the same cache line. However, since locks may be read even within critical sections, it is not generally recommended that locks be allocated in the same line as data they are protecting. If a mechanism is provided to eliminate or defer accesses to the lock until the end of a critical section, then data can be collocated with a lock profitably, allowing the data to be implicitly transferred along with the lock when it is acquired. Bitar and Despain first proposed collocation.[29] Goodman, *et al.*[6] made collocation more attractive by establishing the ability to defer access to the lock by an acquiring processor until the lock had been released. This method, Queue-On-Lock-Bit (QOLB) was a synchronizing prefetch operation in the sense that it provided for lock and data to be forwarded "as soon as possible, but no sooner." Using instruction set and programmer support, QOLB maintained a queue of lock requestors in hardware. Kägi, *et al.*[3] demonstrated that collocation captured consistent and substantial gains in performance for a set of benchmarks on a distributed shared-memory system. Collocation however requires substantial programmer involvement and at times, major restructuring of the application data structures. In addition, coupling the lock and data in the same cache line limits the size of the collocated data. QOLB led to other proposals for queued locks, notably MCS[8] and for the DASH multiprocessor.[21]

DASH provided a concept of queued locks in hardware for memory-based directories. However, the directory was always in the critical path on a lock release, the lock was sent to the directory which in turn picked a random waiter and serviced it. With IQLs, once a request has been forwarded, the directory is no longer in the critical path.

We have previously proposed Implicit-QOLB,[14] an early version of the IQL mechanism, which works by speculating about a program's access patterns—specifically of synchronization operations—and uses the notion of delayed responses to improve the throughput of synchronization. This work focused only on bus-based systems and did not address communication

latencies within critical sections. In recent work,[15] of which a preliminary version appeared elsewhere,[30] we demonstrate that the method is even more effective on a directory-based protocol, and that the SP mechanism can leverage the notion of IQL to achieve still larger performance gains.

Stenström *et al.*[31] and Cox and Fowler[32] independently proposed cache coherence protocol optimizations for migratory sharing patterns. Such behavior is exhibited primarily by data protected by locks or monitors. Both approaches succeed by merging an invalidation request for the migratory cache line with the preceding read-miss request. These mechanisms do not reduce the critical miss latency experienced on the first read miss, though reduced contention may have the indirect effect of reducing read miss latencies.

Mowry and Gupta[33] proposed a compiler prefetch heuristic for tolerating latency in shared-memory multiprocessors. The compiler interpreted explicit synchronization operations as a hint that data communication may be taking place. The approach was quite successful for programs with regular access patterns and structures. They mention that it is potentially easy for a programmer to use semantic information about an application and identify critical data structures in small applications but state, "[s]uch focusing in on critical data structures will be much harder for compilers." An additional issue with software prefetching for critical section data is the lack of knowledge regarding the migratory patterns of data: determining which processor should be prefetching data is nearly impossible statically because it depends on the (dynamic) selection of a winner among competitors to acquire the lock. By the time this decision has been made, it is already too late to avoid delay by prefetching needed data. Trancoso and Torrellas [34] attempted to reduce latencies within critical sections through the use of prefetching and data forwarding. They inserted prefetch and forwarding instructions by hand. Their techniques suffer from many of the same limitations of software approaches, specifically, the need for hardware and compiler support for new instructions and the inability to evaluate and exploit run-time behavior. Their results were pessimistic, concluding that complex, forwarding-based optimizations could not be justified.

Abdel-Shafi *et al.*[35] evaluated producer-initiated communication and proposed remote writes for data accesses associated with synchronization operations. The combination of software prefetching and remote writes provided good performance gains for a set of benchmarks. The mechanisms however, required software and programmer support to identify candidates for remote writes.

Similar data forwarding mechanisms have been proposed in the literature: the *forwarding write*,[36] and the DASH *deliver*.[21] DASH also had a producer-prefetch mechanism for pushing data to a set of consumers in

shared state. Kaxiras and Goodman[37] proposed speculative pre-send as an approach for data forwarding.

Ranganathan *et al.*[12] proposed the use of flush primitives to write back dirty data modified in critical sections to memory. They also added prefetches at the beginning of critical sections. Their mechanisms relied on compiler and programmer support to identify critical data to be flushed. However they state that late prefetches and contention effects limited additional performance benefits. Similar flush primitives have also been proposed by Hill *et al.*[38] and Skeppstedt and Stenström.[39] Mechanisms to reduce invalidation latencies by employing prediction to flush cache lines have also been proposed.[17,40] These techniques reduce a three-hop transaction to a two-hop transaction while SP converts a three-hop transaction into a local cache access.

8. CONCLUDING REMARKS

In this paper, we have studied two mechanisms for reducing communication latencies inside critical sections. First, we discussed IQLs as a mechanism to build an orderly queue of lock requestors to reduce synchronization delay. Second, we described SP, a mechanism to overlap lock transfer with data believed to be associated with that lock, thus attempting to convert all global data accesses performed in a critical section into local cache accesses. We showed that SP offers additional benefits on top of those provided by IQLs.

`Mp3d` was chosen as a benchmark specifically because it exhibits the kind of behavior we were targeting, and both mechanisms succeeded in reducing communication delays. The net result was that the application saw a speedup of 21% for the bus and 32% for the directory system over an aggressive base case of IQLs. Indeed, all the benchmarks saw reductions in shared-memory stalls within critical sections, though for some this delay was so small that the reduction did little to improve overall performance. Benchmarks with highly contended locks (such as `raytrace`), show large speedups in some cases. The SP mechanism can provide further speedups in overall performance by substantially reducing the network traffic and three-hop transactions.

We conclude that the two mechanisms can combine to reduce the communication delays within critical sections by more than 50%. In addition, speculative push can quite often collapse the read-modify-write sequences within a critical section into a local cache access. While the total reduction in stalls varies depending on the percentage of time the processor is stalled for communication latencies, the reduction was consistent across all benchmarks.

ACKNOWLEDGMENTS

This work was performed at the University of Wisconsin—Madison and was supported by NSF Grant CCR-9810114.

REFERENCES

1. T. E. Anderson, The performance implications of spin-waiting alternatives for shared-memory multiprocessors. *Proceedings of the 1989 International Conference on Parallel Processing*, volume II (software), pp. 170–174 (August 1989).
2. T. E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, **1**(1):6–16 (January 1990).
3. A. Kägi, D. Burger, and J. R. Goodman, Efficient synchronization: Let them eat QOLB, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 170–180 (June 1997).
4. A. Kägi, *Mechanisms for Efficient Shared-Memory, Lock-Based Synchronization*, PhD thesis, University of Wisconsin, Madison, WI (May 1999).
5. L. Rudolph and Z. Segall, Dynamic decentralized cache schemes for MIMD parallel processors. *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 340–347 (June 1984).
6. J. R. Goodman, M. K. Vernon, and P. J. Woest, Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors, *Proceedings of the Third Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 64–75 (April 1989).
7. G. Graunke and S. Thakkar, Synchronization algorithms for shared-memory multiprocessors, *IEEE Computer*, **23**(6):60–69 (June 1990).
8. J. M. Mellor-Crummey and M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Transactions on Computer Systems*, **9**(1):21–65 (February 1991).
9. B.-H. Lim and A. Agarwal, Reactive synchronization algorithms for multiprocessors, *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 25–35 (October 1994).
10. L. A. Barroso, K. Gharachorloo, and E. Bugnion, Memory system characterization of commercial workloads, *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 3–14 (June 1998).
11. K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker, Performance characterization of a Quad Pentium Pro SMP using OLTP workloads, *Proceedings of the 25th Annual International Symposium on Computer Architecture* pp. 15–26, (June 1998).
12. P. Ranganathan, K. Gharachorloo, S. Adve, and L. A. Barroso, Performance of database workloads on shared-memory systems with out-of-order processors, *Proceedings of the Eighth Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 307–318 (October 1998).
13. K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren, Architecture and design of AlphaServer GS320, *Proceedings of the Ninth Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 13–24 (November 2000).
14. R. Rajwar, A. Kägi, and J. R. Goodman, Improving the throughput of synchronization by insertion of delays, *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pp. 168–179 (January 2000).

15. R. Rajwar, A. Kägi, and J. R. Goodman, Inferential queueing and speculative push for reducing critical communication latencies, *Proceedings of the 2003 International Conference on Supercomputing*, pp. 273–284 (June 2003).
16. D. Kroft. Lockup-free instruction fetch/prefetch cache organization, *Proceedings of the Eighth Annual International Symposium on Computer Architecture* pp. 81–87 (May 1981).
17. A. R. Lebeck and D. A. Wood, Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 48–59 (June 1995).
18. A. Singhal, D. Broniarczyk, F. M. Cerauskis, J. Price, L. Yuan, G. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, and E. Hagersten, Gigaplane: A high performance bus for large SMPs, *Proceedings of the Symposium on High Performance Interconnects IV*, pp. 41–52 (August 1996).
19. A. Charlesworth, A. Phelps, R. Williams, and G. Gilbert, Gigaplane-XB: Extending the ultra enterprise family, *Proceedings of the Symposium on High Performance Interconnects V*, pp. 97–112 (August 1997).
20. J. Laudon and D. E. Lenoski, The SGI Origin: A ccNUMA highly scalable server, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 241–251 (June 1997).
21. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. Lam, The Stanford DASH multiprocessor, *IEEE Computer*, 25(3):63–79 (March 1992).
22. Institute of Electrical and Electronics Engineers, New York, NY. *IEEE Standard for the Scalable Coherent Interface (SCI)*, ANSI/IEEE Std. 1596–1992, (August 1993)
23. T. Lovett and R. Clapp, STiNG: A CC-NUMA computer system for the commercial marketplace, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 308–317, (May 1996).
24. T. Brewer and G. Astfalk, The evolution of the HP/Convex Exemplar, *Proceedings of the 42nd IEEE Computer Society International Conference (COMPCON)*, pp. 81–86 (February 1997).
25. A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir, The NYU ultracomputer—designing an MIMD shared memory parallel computer, *IEEE Transactions on Computers*, **C-32**(2):175–189 (February 1983).
26. E. H. Jensen, G. W. Hagensen, and J. M. Broughton, A new approach to exclusive data access in shared memory multiprocessors, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, (November 1987).
27. K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26 (May 1990).
28. Compaq Computer Corporation, *Alpha 21264 Hardware Reference Manual* (July 1999).
29. P. Bitar and A. M. Despain, Multiprocessor cache synchronization: Issues, innovations, evolution. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 424–433 (June 1986).
30. R. Rajwar, A. Kägi, and J. R. Goodman, Using speculative push to reduce communication latencies in critical sections, Technical Report CS-TR-1472, Computer Sciences Department, University of Wisconsin, Madison, WI (April 2000).
31. P. Stenström, M. Brorsson, and L. Sandberg, An adaptive cache coherence protocol optimized for migratory sharing, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 109–118 (May 1993).

32. A. L. Cox and R. J. Fowler, Adaptive cache coherency for detecting migratory shared data, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 98–108 (May 1993).
33. T. Mowry and A. Gupta, Tolerating latency through software-controlled prefetching in shared-memory multiprocessors, *Journal of Parallel and Distributed Computing* **12**(2):87–106 (June 1992).
34. P. Trancoso and J. Torrellas, The impact of speeding up critical sections with data prefetching and forwarding, *Proceedings of the 1996 International Conference on Parallel Processing*, Vol. III (software), pp. 79–86, (August 1996).
35. H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve, An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors, *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pp. 204–215 (February 1997).
36. D. K. Poulsen and P. -C. Yew, Data prefetching and data forwarding in shared memory multiprocessors. *Proceedings of the 1994 International Conference on Parallel Processing*, Vol II (software), pp. 276–280 (August 1994).
37. S. Kaxiras and J. R. Goodman, Improving CC-NUMA performance using instruction-based prediction. *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pp. 161–170 (January 1999).
38. M. Hill, J. Larus, S. Reinhardt, and D. Wood, Cooperative shared memory: Software and hardware for scalable multiprocessors, *ACM Transactions on Computer Systems*, **11**(4):300–318 (November 1993).
39. J. Skeppstedt and P. Stenström, A compiler algorithm that reduces read latency in ownership-based cache coherence protocols, *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques* (1995).
40. A.-C. Lai and B. Falsafi, Selective, accurate, and timely self-invalidation using last-touch prediction, *Proceedings of the 27th Annual International Symposium on Computer Architecture* (June 2000).