Computer Sciences Department

Using Speculative Push to Reduce Communication Latencies in Critical Sections

Ravi Rajwar
Alain Kagi
James Goodman

Technical Report #1472

April 2000 (Submitted March 2003)

UNIVERSITY OF
WISCONSIN
MADISON

Using Speculative Push to Reduce Communication Latencies in Critical Sections

Ravi Rajwar, Alain Kagi and James Goodman

Computer Sciences Department
University of Wisconsin-Madison, Madison, WI USA
{rajwar, alain, goodman}@cs.wisc.edu

April 17, 2000

Abstract

Communication latencies within critical sections constitute a major bottleneck in some classes of emerging parallel workloads. In this paper, we propose a mechanism, Speculative Push, *aimed at reducing this communication latency. Speculative push allows the cache controller, responding to a request for a cache line inferred to have a lock variable, to predict the data sets the requestor will access within the critical section. The controller then pushes these addresses from its own cache to the target cache in an exclusive state. It also writes back the data to memory. By overlapping the transfer of the protected data along with the transfer of the lock, the communication latencies within critical sections can be substantially reduced. By pushing data in exclusive state, the mechanism can collapse the read-modify-write sequences within a critical section into a local cache access. The write-back to memory gives the receiving cache the option to ignore the push. We make a case for the use of* Inferentially Queued Locks *(IQLs), not just for efficient synchronization but also for reducing communication latencies. 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. The speculative push mechanism extracts information about program structure by observing IQLs. Neither of the mechanisms require any programmer or compiler support nor any instruction set changes.*

Our results demonstrate that for a set of benchmarks with high communication characteristics, IQLs are able to provide speedups when there is frequent synchronization. In each of the benchmarks we studied, the combination of IQLs and speculative push removed more than half of the processor's observed latency during critical sections.

1 Introduction

The shared-memory programming model is now widely established as a leading paradigm for parallel computing. The abstraction of a shared memory is particularly attractive for irregular applications [39], where reasoning about program behavior and predicting performance may be difficult. In this model, memory takes on a secondary function in addition to holding values: it also provides the means for synchronization and coordination of activities among the processors.

Under the shared-memory programming model, when multiple processors attempt to update a set of variables simultaneously, a *data race* may occur wherein the execution outcome depends on the relative speed of the operations and the order 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 wide range of synchronization mechanisms have emerged over the years [38, 21, 17, 4, 18, 32, 29, 19, 5], and no single mechanism is available in all architectures, though virtually all architectures provide a hardware means for acquiring a lock atomically. Beyond this basic capability, numerous mechanisms have been proposed for enhancing the efficiency of locking in hardware, but to date very few implemented multiprocessor systems have incorporated these ideas. It has been argued that parallel applications that spend too much time in critical sections could be restructured to minimize synchronization and in fact this is true of many structured scientific applications [26]. These applications are computationally intensive, highly regular, and generally display easily exploitable parallelism.

However, among the new classes of programs emerging as important applications for parallel systems, online transaction processing (OLTP) workloads display radically different behavior from the traditional scientific applications: they are characterized by high communication miss rates [6, 15, 24, 36, 7]. A recent study by Ranganathan et al. [36] showed that a large fraction of the misses are generated within critical sections. For a given configuration of a 4-way system running the Oracle database engine, 20% of the execution time was spent servicing communication misses to dirty data in remote caches. OLTP workloads are characterized by fine-grain updates of meta-data[1] and frequent synchronization that protects such data. Thus, the protected data sets migrate among processors with the passing of the lock. This behavior contributes to a large portion of the access latencies to dirty data in remote caches. With a larger number of processors, faster processor speeds, and relatively increasing remote access latencies, the processor stalls induced due to communication misses within critical sections will only increase.

These results show that, at least for OLTP systems, efficient handling of data accessed in critical sections must be provided along with efficient locking mechanisms in order to reduce the substantial communication latencies.

This paper has two objectives. The first is to make a case for the use of *inferentially queued locks* (IQLs), that is, devoting hardware to build an orderly queue of lock contenders. The queue is speculative in that the processor infers the existence, and limits, of a critical section from the use of synchronization instructions. The second objective is to propose a new technique, *speculative push*, for reducing the miss latency associated with data accesses within critical sections. The speculative push allows the cache controller of a processor currently holding a lock not only to defer momentarily its response for a request for the cache line holding the lock, but to provide additional cache lines at the same time, anticipating misses likely to occur immediately after the requestor has acquired the lock. By overlapping the transfer of the data with the lock, the communication latency experienced within a critical section can be reduced. Our simulation results show that for a set of benchmarks, the proposed mechanism accurately captures the

1. The meta-data primarily has structures for communication and synchronization among various database processes.

behavior of migratory data associated with frequently executed critical sections. The mechanism is able to eliminate a large fraction of stalls incurred within critical sections while accessing shared data which is dirty in remote caches. Speculative push combined with IQLs provides speedups over our aggressive base case.

For critical section accesses, speculative push has some inherent advantages over more traditional approaches for latency reduction such as prefetching and compiler assisted data forwarding. A speculative push transfers data as soon as the data is ready to be forwarded: it does not interfere with the execution of the processor performing the push. By pushing data along with the lock, the target node can be prevented from issuing an access for the pushed data. In addition to potentially reducing network traffic, this mechanism can substantially reduce the observed miss latencies once the target node has acquired the lock. Finally, the speculative mechanism can adapt to run-time behavior.

This paper makes a case for the use of IQLs not just for efficient synchronization, but also for reducing the communication miss latencies experienced once the lock has been successfully acquired. 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. An advantage of hardware queued locks is the early, accurate knowledge of the new acquirer of a lock.

In the presence of frequent synchronization to migratory locks there is great benefit in optimizing the lock access: network contention is reduced, thus having a positive effect on memory system performance. As part of the contribution, we show how IQLs can be efficiently supported by supplementing existing cache coherence protocols with some additional functionality.

A key to the success of the techniques of IQLs and speculative push is in the way they interact to achieve high prediction accuracy (i.e., knowledge of program structure) and to reduce latencies associated with both lock and critical data access. In addition, they do not require any programmer or compiler support and do not require re-compilation of code.

We discuss an implementation of IQLs for snooping and directory protocols in Section 3. Then we extend the base protocols to incorporate mechanisms for speculative push in Section 4.

We select a set of benchmarks that display frequent synchronization and high communication characteristics. The benchmarks are scientific: however, they display behavior of migratory locks and communication misses within critical sections.

2 Background and related work

For the duration of a critical section, a processor's progress can be critically affected by access to remote data. In a critical section, the processor typically must access variables that are actively shared. There are

two important reasons why access to these variables may be slower than other accesses: (1) misses due to sharing, and (2) interference from other nodes.

First, the initial access to a shared variable is likely to experience a cache miss because the value was recently modified by another processor and must be fetched from that cache. Such accesses often occur immediately after the successful acquisition of the lock. While latency tolerating techniques are well established for prefetching cache lines about to be accessed, this procedure is generally not performed *before* the critical section is entered. Prefetching data before acquiring the lock can have the highly detrimental second effect of interfering with data being actively accessed in another processor. For this reason even aggressively speculative processors generally do not prefetch operands past a synchronization point.

Regarding cache misses, the lock is especially vulnerable to these phenomena. Since the lock controls entrance to the critical section, it must be available — at least for read access — at any time. Since it must be written to, both as a part of the acquire operation and as a part of the preceding release, a delay is likely during acquisition in order to obtain write permission. In addition, while the processor is in the critical section, the lock may be read by other processors seeking access, requiring it to reacquire write permission in order to release the lock. The mechanisms we propose have the ability to eliminate virtually all of these communication delays, leaving only a single transfer delay from release of a lock until the acquisition by the queued processor.

Because of possible read accesses to locks 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, 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 [8]. Goodman, Vernon, and Woest [17] 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, Burger, and Goodman [22] 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 [32] and for the DASH multiprocessor [28].

Recently Rajwar, Goodman, and Kägi [35] proposed Implicit-QOLB. Their mechanism 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. They only provide an implementation in the context of a bus-based system and do not address communication latencies within critical sections. However, their work provides the basis for our IQL mechanism. We demonstrate that the method is even more effective on a directory-based protocol, and show that the speculative push mechanism can leverage the notion of IQL to achieve still larger performance gains.

Stenström, Brorsson, and Sandberg [43] and Cox and Fowler [13] 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 block 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.[1] 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.

Abdel-Shafi et al. [2] 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 [34], and the DASH deliver [28]. DASH also had a producer-prefetch mechanism for pushing data to a set of consumers in shared state. Kaxiras and Goodman [23] proposed speculative pre-send as an approach for data forwarding.

Ranganathan et al. [36] 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

1. Assuming that a program is "properly labeled" [16] or "data-race-free" [3], synchronization statements should exist between the time when one processor modifies data and a second processor reads that data.

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. [20] and Skeppstedt and Stenström [42].

Trancoso and Torrellas [44] 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.

A study by Ranganathan et al. [36] characterized the behavior of an online transaction processing (OLTP) benchmark using the Oracle commercial database engine and out-of-order processors on a 4-way shared-memory multiprocessor. An important observation was the frequency of data communication misses among the processors. Servicing read misses to remote dirty cache lines accounted for 20% of the *total execution time* of the system. These misses — a large fraction of which occurred within critical sections — accounted for 50% of all misses from the second level cache. They also noted that most of these cache misses targeted only a small fraction of the total number of cache lines experiencing misses. This study implies high predictability of the protected data sets for locks.

An often overlooked but important issue is the actual implementation of the synchronization operations in real systems. Many architectural restrictions are imposed for code sequences around lock acquire and release operations. These restrictions generally interfere with the ability to prefetch data beyond an unacquired lock. For example, memory operations have to be handled carefully when issued past a Store-Conditional instruction in case they result in an eviction of the Load-Locked line and artificially force the Store-Conditional to fail [12]. In addition, if the processor is not snooped on invalidates to the cache (permitted in some implementations of relaxed memory ordering [12]), for correctness the processor may not speculatively issue loads into critical sections.

Processors such as the MIPS R10000 [46, 1] with aggressive implementations of sequential consistency have support for issuing load operations speculatively and detecting violations in hardware. However, the data dependant nature of the misses may prevent loads from being issued early enough to avoid delay. Some recent papers have demonstrated that, because of the data dependant nature of the computation, the number of outstanding second level cache misses is very small for database servers [6, 15, 24, 36, 7].

With an increasing trend in the server market towards simple fast processors with aggressive memory systems, it may be important for a mechanism to not rely on a processor's ability to generate misses early enough so as to hide memory access latencies to actively shared data.

3 Inferentially queued locks

We begin this section by noting that 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 cache has a shared or exclusive copy of a cache line containing the lock. In the first case we will refer to a lock as being *acquired, held,* or *released.* In the second case we will 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 and for which it may be inferred that this processor is attempting to acquire a lock. A *responder* is a processor holding a writable copy of a lock-line sought by a requestor.

Two observations about performance loss due to lock interference provide motivation for inferentially queued locks:

1. When a processor requests a lock-line for purposes of acquiring the lock, it is very likely to spin-wait upon discovering that a lock is already held. If the response to this request is delayed briefly, any increase in the probability that the lock has been released will increase the success rate for the initial attempt, and thereby reduce total communication. The latency to acquire a held lock is minimal if such a request is serviced immediately after the lock is released.
2. If such a request is serviced immediately, that is, while the lock is still held, the release of the lock will probably be delayed, because releasing it will require the lock-line to be re-obtained in a writable state. Again, any delay resulting in an increase in the probability that the lock has been released is likely to improve performance, not only by reducing total communication, but also by avoiding a delay in releasing the lock.

Processors with non-blocking caches [25] can issue multiple outstanding requests to the memory system. Such processors use special buffers such as miss status holding registers (MSHRs) [25] to resolve the pending misses when they are serviced from lower levels of the memory hierarchy. In multiprocessor configurations, these structures are also used to buffer[1] requests from other processors to caches lines which are in a pending state. The cache controller services a buffered request, if any, when it has finished processing the current request that caused the cache line to move to the pending state in the first place.[2]

Inferentially queued locks 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 it 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 locks occurs directly between the two nodes involved without the coherence network being in the critical path.[3]

---

1. This depends on the underlying cache coherence protocol. Commonly, only one intervention request per processor is allowed.
2. This again depends on the underlying coherence protocol and memory consistency model.

The method is speculative because the hardware does not have sufficient information to know for sure even that a lock is being employed, but infers this from program behavior, and specifically infers acquire and release points for a critical section. IQLs can be supported naturally in many modern system implementations.

3.1 Basic protocol

The basic protocol must recognize when a lock is being accessed and take certain actions. Specifically, when a lock is acquired, the local processor has a copy in its cache, and it must record the fact that the lock is held. When a lock is released, the local processor must record the fact that the lock is no longer held, and it must also initiate actions triggered by release of the lock. A request from a remote node to access a lock-line should be annotated to indicate that the line is being sought as part of an acquire operation, invoking the protocol that allows deferring that request for the lock line. Since this line is being requested for writing, we refer to this request as a low-priority read-for-exclusive (`rd_x_lp`). The request may be deferred for a short time, but not ignored.

When a processor observes a synchronization event likely to indicate the acquisition of a lock (such as a Load-Locked instruction), it must first distinguish between a simple, atomic read-modify-write operation and the acquisition of a lock. Details of how this distinction might be inferred for the specific case of Load-Locked and Store/Conditional primitives can be found elsewhere [35]. We assume here that the processor has concluded from previous executions of this code sequence that the synchronization event is the acquiring of a lock. If this assumption is wrong, performance may be degraded, but the program will be correctly executed. If the lock-line is already present in the local cache, it is marked HELD in a hardware Lock State Table. Otherwise, space is allocated in the cache, and a low-priority read-for-exclusive is initiated. The entry in the Lock State Table is marked REQUESTED. When the line eventually arrives, the Lock State Table is changed to HELD. On a subsequent write to the byte address thought to be the lock, the entry in the Lock State Table is changed to PRESENT. When a lock-line is evicted from the cache for whatever reason, including invalidation, the Lock State Table entry is marked INVALID.

In summary, all locks known to the controller through the Lock State Table will be in one of four states:

a) HELD: the lock is held by the local processor.

b) REQUESTED: the local processor has requested the lock but the lock has not arrived yet.

c) PRESENT: the local processor has the lock line present in its cache, but does not hold the lock.

d) INVALID: the local processor does not have the lock line present in its cache, and (probably) does not hold the lock.

---

3. For a directory, on a lock release, upgrade permissions may be required from the directory. In addition, the waiting processors will in turn access the directory to acquire the lock. In the case of snoop systems, the processors send repeated requests on the snooping bus.

When a low-priority read-for-exclusive request is received, the Lock State Table is consulted. If the lock-line requested is known to the local processor and is in state PRESENT, the request is handled as any other read-for-exclusive request, and a response is sent to the requestor. If the lock-line is in state HELD or REQUESTED, the lock-line is marked for special action, which will be effected upon the release of the inferred lock, and the request is buffered in an MSHR. A subsequent write to the byte address of the lock in the lock-line triggers a response sent to the recorded requestor (tracked in the MSHR), and the corresponding entry in the Lock State Table is marked INVALID. The abstracted state transition diagram is shown in Figure 1.

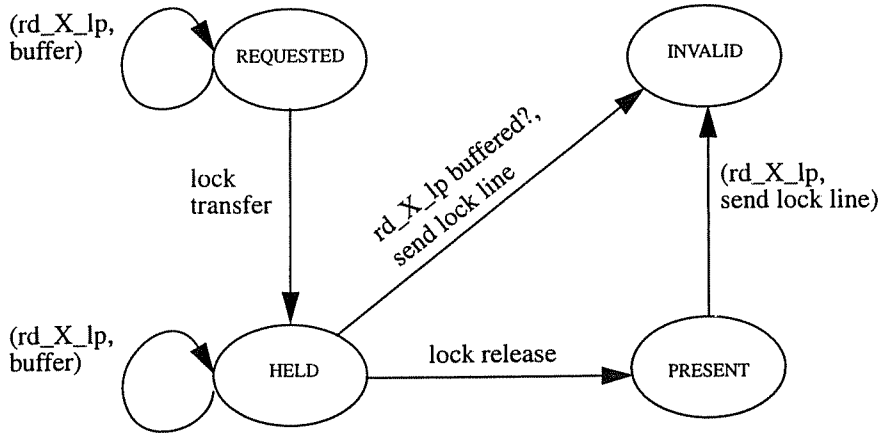

Figure 1. Building a queue of lock requestors.

A processor will not receive multiple low-priority read-for-exclusive requests (this is generally guaranteed by the base protocol), because the first such request transfers the apparent owner of the lock line to the requestor, who then becomes the recipient of a subsequent request. By this means, a queue of requesting processor can be assembled, with each processor receiving the lock-line sequentially in the order of original request. Special arrangements must be made to handle the occurrence of a regular-priority read, whether for-exclusive or shared. The requirement to distinguish such a request and handle it correctly is the reason for the low-priority read-for-exclusive designation, though interactions should only occur in rare cases where the lock protocol is being violated or the hardware has otherwise mis-speculated.

We next discuss how IQLs can be supported on two popular design approaches: snooping-systems and directory-systems. We take two base protocols from commercial systems and supplement them to support IQLs.

3.2 Inferentially queued locks and snooping systems

We use a snoop design similar to the Sun Gigaplane. The Sun Gigaplane 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 3-state (*Owned, Shared, Invalid*) snooping cache coherence protocol with no transient states. The cache with the requested block in *Owned* state (as seen by the bus) will respond to the next request for that block. Details for the above protocol can be found elsewhere [41, 11]. IQLs can be supported naturally with the above protocol as the protocol already has a 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 block 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 has joined the queue of lock requestors at the tail and will be serviced when the earlier requestor releases the lock.

3.3 Inferentially queued locks and directory systems

Directory protocols are primarily used to allow cache coherence to scale to a large number of processors. Directory protocols store a *directory state* for each memory block currently cached in any node. Common directory protocols implemented have been invalidation-based. With directory protocols, state information for a block is obtained from a *directory* through network transactions, and communication with various cached copies is performed by explicit messages using an arbitrary network. Commonly, information for a block can be found in a fixed directory 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 block at the home node of the block. Examples of memory-based directory protocol systems include the SGI Origin-2000 [27] and the Stanford DASH Multiprocessor [28]. In cache-based schemes, the sharing information is distributed among the various copies (rather than at the home node). Each cache block contains a pointer to the node that has the next cached copy of the block in a distributed linked-list organization. The IEEE 1596-1992 Scalable Coherent Interface (SCI) standard indicates a full specification and C code for a standardized cache-based directory organization and protocol. Commercial implementations of the SCI include the Sequent NUMAQ [31] and the Convex Exemplar X [9]. We discuss mechanisms for supporting IQLs in a memory-based directory. For cache-based directories, it is relatively easier to support IQLs, since there already exists a notion of queues for cache blocks.

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.

Memory-based directories

We use the SGI Origin-2000 coherence protocol as the base protocol. The protocol supports the standard MESI (*Modified, Exclusive, Shared, Invalid*) states and is non-blocking: memory does not buffer requests while waiting for other messages to arrive. The protocol supports request forwarding for three party transactions. In addition, silent evictions of clean-exclusive lines are supported. The protocol does not rely on an ordered network. Only two virtual channels are provided and deadlock in the request network (due to request forwarding) is broken by the use of *backoff* messages. Details of the SGI Origin-2000 protocol can be found elsewhere [27, 14].

In the base protocol, memory is the owner for all clean lines in the system: thus any request for clean data is immediately serviced by memory. In addition, for read-for-exclusive requests, ownership is transferred to the requestor and *invalidates* are sent to other cached copies. The cached copies subsequently send *invalidate acknowledgements* to the requestor. Requests to blocks not owned by memory are *forwarded* to the owner (and in the case of a read-for-exclusive request, the requestor becomes the owner). The directory goes to a transitional *Busy* state until a *revision* message is received from the previous owner. All requests received by the directory while in *Busy* state are NACKed.

Building queues

To support the notion of queued locks, the protocol must be supplemented with some additional functionality. Under the base protocol, the directory enters a transitional *busy* state while tracking down an *exclusive* copy in the system. For queued locks, the directory, instead of NACKing another request while in *busy* state, must forward the request to the previous requestor. If the previous requestor's request is still pending, the intervention will be buffered. Forwarding a request to the last requestor guarantees that a processor receives at most one intervention for a given memory block.

In our implementation, we use an additional bit, the `synch_bit`, per directory entry.[1] This bit determines whether the supplementary protocol needs to be invoked. In addition, the `owner` pointer is overloaded to store the `last_requestor` when the `synch_bit` is set.

To understand how queues are created we step through a simple example. We will then discuss how the directory detects queue breakdowns (in the case of write-backs). Consider three processors, P1, P2, and P3 attempting to enter a critical section.

1. Alternative ways of capturing such information includes using unused state bits in the state encoding etc. This information could also be cached at the directory controller.

To start with, processor P1 sends a `rd_X_lp` request to the directory. The directory owns the block and the block is not cached anywhere in the system. The directory responds to the request with exclusive permissions and enters an *Exclusive* state. P1 becomes the owner of the block.

Subsequently, Processor P2 sends an `rd_X_lp` 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: for P1 and for P2. The `synch_bit` is also set.

Now, P3 sends a `rd_X_lp` request to the directory. The directory state is *Busy*. However, the `synch_bit` is set. Rather than NACK the request, the directory forwards it to the last requestor, P2. In addition, 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, if necessary, until the lock is released. At the time when the intervention is serviced, a *revision* message (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. There is a guarantee that for every `rd_X_lp` request serviced, there will always be a *revision* message. This ensures that the directory will always be able to transition into an *Exclusive* or any other stable state.

A simplified state transition diagram for the supplementary protocol is shown in Figure 2.

Handling queue breakdown

Due to the distributed nature of the queue (the directory does not track the order in which requests are received: it only marks who has requested the block), it is possible that the queue may break-down due to write-backs. For the above example, suppose P1 received P2's intervention but had silently evicted the block. In this case, there is no problem since P1 acknowledges P2 and responds to the directory with a *revision* message (same sequence as in the base protocol for handling silent evictions). However, suppose P1 is writing back the block to the directory. Under the base protocol, P1 can ignore the intervention since the directory can recognize this race condition and detect that P2 will not be serviced by P1. The directory can do this as it keeps track of the owner P2. However, under the supplementary protocol, if there are multiple bits set in the bit-vector (the `synch_bit` is set and a queue exists), the directory can not make the determination as to the identity of the processor which will not receive a response from P1 — the directory does not remember that it forwarded P2's request to P1. It is important to note that this race condition is indeed rare. Of course, it must be handled correctly, if not efficiently.

We adopt a simple approach to solving this problem: on receiving a write-back to a block with a `synch_bit` set and more than 2 bits set in the bit-vector (if only 2-bits are set, the directory can uniquely

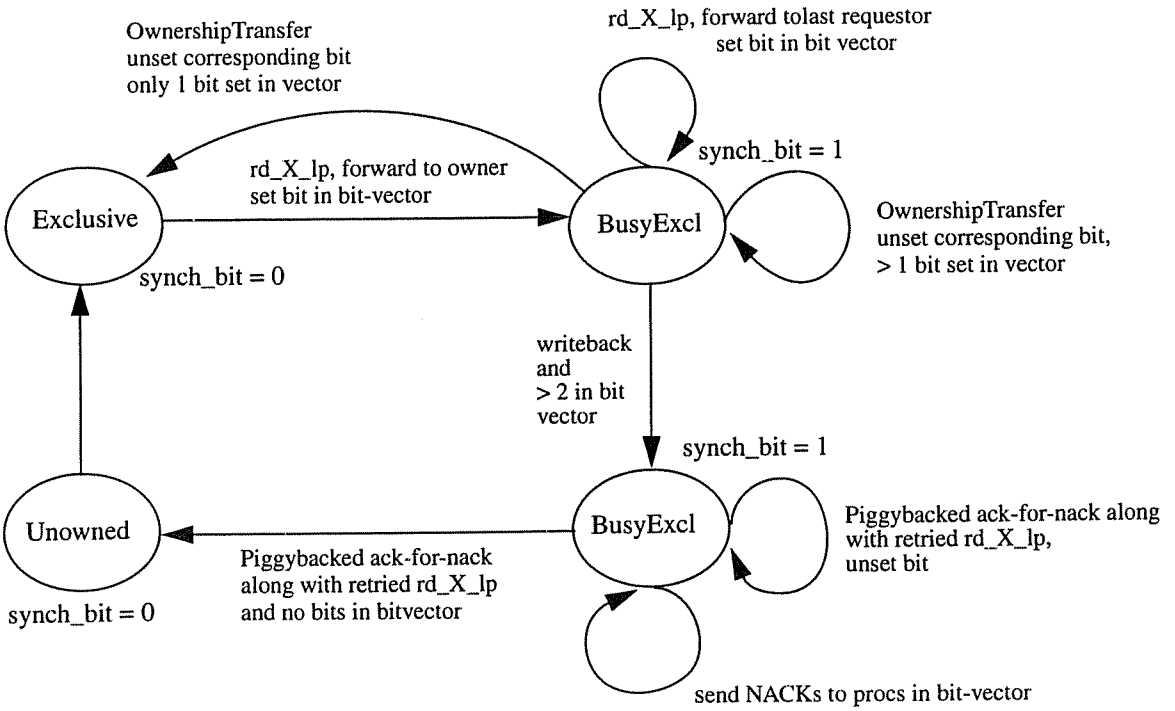

Figure 2. Protocol transitions for speculatively queued locks in a directory. A subset of the transitions is shown. Primarily, the process of forming a queue (using a synch_bit and a last_requestor). The queue-breakdown sequence is shown when a writeback occurs.

determine the processor which will not receive a response), the directory unsets the `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 processor's 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. It is easy to see that such a mechanism can guarantee that a directory will, in a bounded period of time, reach an *Unowned* state in the case of a rare race-condition.

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.

We have discussed ways to implement IQLs in both snooping and directory protocol systems. The implementation for snooping systems is quite simple and leverages many existing mechanisms. The implementation is slightly more involved for memory-based directory protocols. This is primarily because of the way directory protocols have been designed and the sharing patterns they have been optimized for. However, the changes are not substantial and if the targeted workload for these systems have frequent synchronization, it may be necessary to provide such support in the protocols themselves.

4 Speculative push

For critical sections that are actively shared, access to the lock is important to optimize because it may be accessed by other processors even while guaranteeing exclusive access to the data it protects. But some of that data is frequently modified, with the result that efficiently handling the lock only exposes a subsequent delay in accessing the protected data. Given that the IQL mechanism has already inferred the behavior of the program with regard to critical sections, it is only a small additional step to track the shared information in the critical section and offer it to the requesting processor, along with the lock, so that when it acquires the lock it finds already present in its cache the data it was unable to prefetch. An additional benefit is that the data is already in exclusive state, allowing the processor to modify the cache line without experiencing a further delay it would experience if it had originally retrieved the line for reading. In this section, we discuss a hardware mechanism, speculative push, for reducing the latency of access to data associated with migratory locks.

The concept of speculative push can be applied to either a snoop-based or directory-based system, though the details specific to these two schemes are quite different. We begin by describing the basic protocol independent of the underlying coherence protocol. Later, we will show how the approach may be implemented in the two coherence protocols discussed earlier.

4.1 Basic Protocol

The protocol builds on Figure 1 of Section 3. The Lock State Table provides the information for inferring when the processor is within a critical section. It is a simple matter to record the cache lines that are modified during the critical section, and these cache lines (other than the lock-line) become candidates for forwarding to the processor acquiring the lock. We restrict the mechanism to modified lines because, while some shared lines may also result in misses, we found it highly likely that such lines would already be present in the requestor's cache.

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. We found that some simple mechanisms could be employed to reduce the number of wasted push operations, and the choice of lines pushed is discussed in Section 4.4. Note that, for purposes of speculative push, it is not important to distinguish between a HELD lock being released with a request outstanding and an incoming request finding a lock present but not held (PRESENT). In both cases, the triggering event causes predicted caches lines to be pushed.

If the predicted lines are in *modified* state in the responder's cache, the information (which may include the data or simply the address hints, depending on the trade-off with the underlying protocol) is sent (or can be piggybacked with the lock transfer) to the requestor. Doing so essentially informs the requestor to expect data to be pushed. Thus, a transfer of data modified within a critical section can be initiated before

the requestor has even acquired the lock. The line is pushed in exclusive state. Thus, if the pushed data is not referenced, little harm is done, since the receiver can silently evict the block.

A write-back is also sent to memory, thus providing the target node of the push, the important option to reject the push if no local buffer space is available. Interestingly, even if the push is rejected, some benefits may accrue. 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. In our simulations, we were very conservative in accepting the push. In particular, a pushed cache line was never allowed to evict a valid cache line, that is, it was accepted only if there was a line available with invalid data. This approach is surprisingly effective, in part because migratory lines that are frequently written are frequently invalidated, leaving available holes in just the right places.

4.2 Serializing the speculative push

For a simple 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. The data would normally be pushed immediately after the response providing the lock. Since the data is also being written to memory to allow the receiving node to ignore it, this three-way communication presents ordering requirements in a directory-based system. In our example bus system as well, since data is transmitted point-to-point, a write-back operation requires some care to assure that the data is received, and serialized correctly, at the target node.

Since the speculative push is being initiated on a network separate from the one used for enforcing serialization (at the snoop bus or directory) the recipient can not commit use of the data until the push has been serialized (though early access may still be beneficial, for example, if value speculation [30] is being performed in the critical section — data is present and most probably will be a valid copy). The overlapped transfer of the lock and data provides performance gains and the requestor will receive data it is likely to modify, in *exclusive* state, without having issued a request. The fact that the data is writable has the potential to reduce upgrade traffic substantially.

To serialize the push, an annotated write-back (the annotation identifies the target of the speculative push) is sent to memory and is serialized in the global memory order. For a bus-based implementation, 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. For a directory implementation, the directory node must be involved because it serves as the coherence point. Responding to the annotated write-back, the directory node communicates with the target node, granting coherence permission (exclusive) or sending a NACK to the target node if necessary.

The speculative push sequence involves two separate actions: one involves the push of the data (or the address hints). The other grants coherence permissions to the target once the push has been serialized in the global memory order. The two actions are necessary to exploit the opportunities for reducing communication latency. Getting information to the target node quickly is necessary to avoid unnecessary requests: some potential benefit is lost if the target node requests a cache line just before it arrives as a result of a push. However, the data cannot be used until it has been serialized in the global memory order.[1]

Note that the data itself can be transmitted with either action. The underlying protocol can affect the trade-off involved in the actual implementation of the speculative push. For example, in the case of a snoop-based system, it may make sense to push data independent of coherence permissions. Aggressive systems are designed so that the data network is not the bottleneck and performance is limited by snoop bandwidth. Thus, the data network can be utilized to overlap the data transfer with the granting of coherence permissions. On the other hand, for some directory systems, address hints may be enough since the data is in any case going to first go to the directory for serialization. For the two systems we have simulated, 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 with the lock transfer).

The two actions described above can occur in any order and different networks may be used for them. In addition, there may be multiple pushes of a cache line to a given processor — since there are no guarantees of ordering from the network. To assure correct handling of such a situation, we treat the two messages as symmetric. If a push is rejected, the corresponding coherence permission must also be rejected and vice-versa. A simplified transition diagram is shown in Figure 3.

Some bookkeeping is necessary to track multiple cache lines to assure consistent responses to both actions on each. For generality, without making assumptions about the network, we required a requestor to include in a `rd_X_lp` an indication of the number of lines it can track (but not necessarily sink) at any given time. For the applications we studied, we found that this number could be quite small. Generally, nearly all the benefit available could be achieved by pushing up to four lines.

The push/coherence permission information is stored in a small table at the cache controller. It can be guaranteed that both messages in the pair will always 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. It is important to note that any push reject can be matched with any coherence reject and so on. As the mechanism is speculative, it is possible that the push and coherence permissions 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.

1. At the expense of additional complexity, these serializations can be relaxed if support for detecting this is added to the protocol.

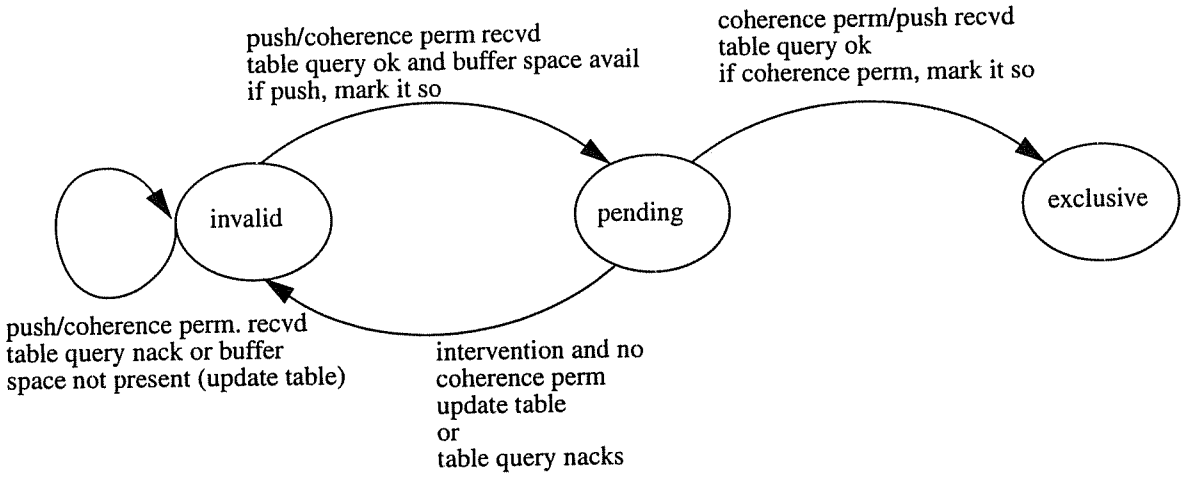

Figure 3. Transition diagram for push. The table query essentially matches up push and coherence permission pairs. If everything matches up, the transition occurs to *Exclusive*. If there is a mismatch during the table-query, the push is rejected.

A third processor may attempt to read data in a cache line while the line is being pushed. This may be due to data races or 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). Details are omitted for brevity.

4.3 Timing of the push

As with any forwarding or prefetch scheme, the timing is critical. By piggybacking the speculative push address hints on the lock transfer, one can inform the target of pending pushes. Thus, the target's cache controller pre-allocates data buffers for the pushed data, thereby preventing the target from requesting the lines. 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. By preventing the target from requesting data being pushed, contention effects at the directory and network are also reduced. In case of a mis-speculated push, the block can be silently evicted. For snooping systems, since the bus provides an implicit ordering, in many cases, the latency can be overlapped to a larger extent than in a directory system.

4.4 Predicting communication misses

Typically, a small set of cache lines is modified within a critical section. A predictor is used to track cache lines that are modified within a critical section, for purposes of selecting candidates for speculative push. Two actions identify candidates for future pushes: lines that caused write misses during a critical section and lines that have 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 that the data is migratory. Local data that is unshared may also cause a write miss. However, it is more likely to remain in the cache, since it will not be invalidated when another processor writes it.

We implemented a fairly simple predictor, identifying a critical section by a lock address and a lock PC In addition, each entry has a set of four cache line addresses associated with the lock identifier and a 2-bit confidence predictor. The scheme is not highly optimized, but generally succeeds in avoiding pushing data that is only locally referenced while consistently identifying true migratory lines. If a line is received as a result of a push, it is entered in the predictor for the corresponding critical section with a maximum confidence prediction. Each time a cache line write miss occurs, the address is entered into the predictor, if necessary, replacing a cache line with minimum prediction confidence. If the cache line address is already present in the predictor, its confidence level is increased by one until the maximum is reached.

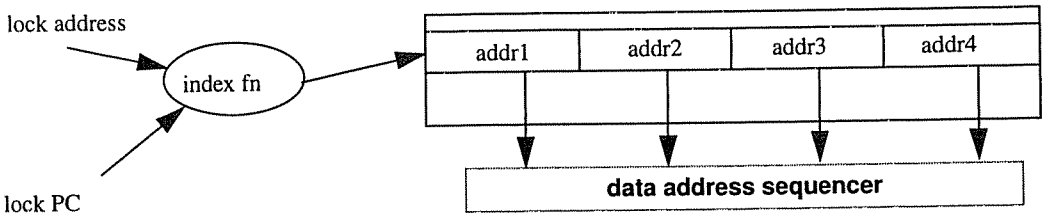

Figure 4. Communication-miss address predictor. Shown here for a 4-address predictor.

When the predictor is consulted for selecting lines to push, all lines with confidence greater than the minimum (that is, those that have caused more than a single write miss) are candidates, with the highest confidence lines selected first. If a cache line is not present in the cache, or if it is present, but not dirty, the confidence level is reduced by one. Note that this last feature avoids the possibility of a cache line being repeatedly pushed but never written

We believe that more sophisticated predictors — for example, one using information about remote requests for modified data in the local cache, or combining collective information from multiple nodes about migratory patterns — may be able to improve the effectiveness of the speculative push. This simple predictor proved adequate for the benchmarks we tested.

5 Experimental methodology

We use an execution-driven simulator developed for performing studies of shared-memory multiprocessor systems. The simulator is derived from the SimpleScalar Toolset [10] and allows detailed simulations of shared memory multiprocessor systems. The simulator performs a cycle-by-cycle simulation of an aggressive out-of-order processor core and a detailed event driven simulation of the memory hierarchy. Data movement is closely modeled in the pipeline. Instructions which were not present in the SimpleScalar ISA were added as necessary using the annotation mechanism as available in the toolset. Our simulator models data movement in the memory hierarchy accurately and models bandwidth and port contention at all levels. We assume sequential consistency for our memory model. However, we have an aggressive implementation of sequential consistency similar to the MIPS R10000.

5.1 Benchmarks

Not all benchmarks display the behavior of high communication miss rates we are targeting. OLTP servers are the class of applications which do have communication miss rates within critical sections. However, due to the limitations of time and the complexity of obtaining detailed performance results with commercial workloads, we use benchmarks which demonstrate similar behavior at a smaller scale. We select four benchmarks which display varied critical section behavior. Raytrace, Ocean-cont, and Water-Nsquared are

Table 1: Benchmarks

Applications	Type of Simulation	Inputs	Processor Count
Raytrace	3-D Rendering	teapot	16
Mp3d	Rarefied fluid flow	24000 mols, 25 iter.	4
Water-Nsquared	Water molecules	512 mols, 3 iter.	16
Ocean-cont	Hydrodynamics	x130	16

from the SPLASH-2 [45] benchmark suite while MP3D[1] is from the SPLASH [40] benchmark suite. For a given benchmark, the same benchmark binary is used for the different mechanisms

5.2 Target systems

The simulated snooping bus system is shown in Figure 5. It is modeled after the Sun Gigaplane [41]. The point-to-point data network is not shown. The simulated directory based system is shown in Figure 6. The parameters for the various components and configurations is given in Table 2 and Table 3. The proces-

1. MP3D was compiled with the locking option (doing so removes data races in the benchmark).

sor is an aggressive one with a large number of functional units. Thus, functional unit stalls for scientific computation are low.

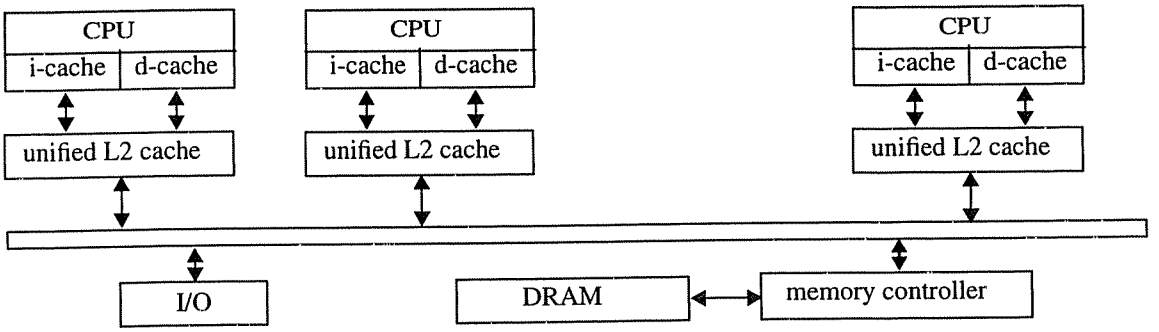

Figure 5. A snoop based configuration. The point-to-point data network is not shown.

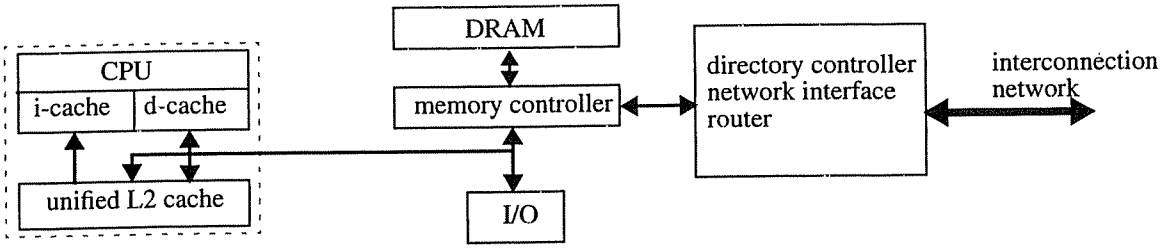

Figure 6. A directory based configuration. The directory and memory controller are integrated. The processors is integrated with the L2 cache.

Table 2: Integrated processor and cache subsystem

Processor	
Processor speed	1 GHz (1 ns clock)
Reorder buffer	64 entry with a 32 entry load/store queue
Issue mechanism	out-of-order issue/commit of 4 operations per cycle, 64 entry return address stack, aggressively issue loads (similar to the MIPS R10000)
Branch predictor	8-K entry combining predictor, 8K entry, 4-way set-associative BTB
L1 instruction cache	64-KByte, 2-way associative, 1-cycle access, 8 outstanding misses
L1 data cache	128-KByte, 2-way associative, write-back, dual-ported, 1-cycle access, 8 outstanding misses
L2 unified cache	1-MByte, 4-way associative, write-back, 10-cycle access, 16 outstanding misses
L1/L2 bus	Runs at processor clock
Line size	64 bytes

Table 3: External network and memory configuration

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

5.3 Explanation of metrics

Speedup is calculated as the ratio of the parallel cycle time of the base case (for that table) to the parallel cycle time for the optimized case. Thus, a speedup > 1 implies performance gains. We also measure the impact on the network for the various mechanisms. This metric is normalized with respect to the base for that table. It is calculated as the value for the optimized case over the value for the base case. If a number is greater than 1.0, the optimized case is performing worse for that metric than the base case. Since we are targeting L2 misses within critical sections, we compare absolute miss counts. These are calculated the same way as the network measurements.

In order to get a deeper understanding of the behavior of the mechanisms, we break down the execution time of the benchmark into various components. It is hard to do this for out-of-order processor systems configured for multiprocessors. However, we adopt an approximation approach similar to the one used in literature [36, 37]. At 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. Since our proposals target shared memory access latencies within critical sections, we further divide the shared memory access induced stalls by the type of event they triggered: more specifically, we classify the miss based on where the miss was serviced from and what the state of the cache line servicing the miss was. The shared memory accesses do not include lock variable accesses. This is done specifically to avoid any bias-

ing of miss counts and latency accounting by including different synchronization primitives. The stall times provided are percentages of the execution time (or the shared memory stall time if applicable) for the specified optimization and are not normalized.

6 Results

6.1 Performance of Inferentially Queued Locks

It is no surprise that the two benchmarks performing little synchronization do not benefit from IQLs. However, ocean does show a 24% speedup for the directory configuration primarily because of the reduction in network traffic .

Table 4: IQL performance. Normalized rows are compared to the base case of a Test&Test&Set lock implemented with the load-locked/store-conditional primitives. The *stall contribution* row measures the contribution of lock accesses (tests, acquires, and releases) to the execution time of the system. See Section 5.3 for details.

		Ocean		Water-Nsq		Raytrace		Mp3D	
		snoop	dir	snoop	dir	snoop	dir	snoop	dir
Speedup (normalized)		1.03	1.24	1.02	1.04	1.44	5.51	1.13	1.28
Snoop Bus traffic (normalized)		0.95		0.90		0.36		0.79	
Network Traffic (normalized)	total msgs		0.93		0.89		0.12		0.80
	requests		0.90		0.89		0.09		0.79
	responses		0.93		0.90		0.14		0.83
	interventions		0.95		0.87		0.19		0.77
Stall contribution	TTS locks	2.8%	9.31%	3.54%	5.82%	48.8%	66%	31.6%	32.1%
	IQLs	1.80%	1.67%	2.08%	2.78%	30.5%	64%	23.7%	20.3%

Raytrace on the other hand shows substantial improvements for both, the directory and bus configuration. Raytrace has a highly contended lock which makes its performance very sensitive to the synchronization primitive (as can be seen from the significant amount of time spent in synchronization operations). The gains primarily come from the significant reduction in network traffic due to the queuing effect of IQLs. High contention effects at the directory and network degrade performance to a much larger extent than in the case with the snooping bus. MP3D has migratory locks and frequent synchronization. The gains for IQL primarily come from the reduction in network traffic (a lock acquire under IQL can occur with a single network transaction by the requestor. For the test&test&set primitive, the network traffic is substantially larger).

Table 5: Speculative push performance. Normalized rows are compared to the case of IQLs. CS stands for a critical section. The *stall contribution* rows measures the contribution of various memory accesses to the execution time (or, if applicable, the shared memory stall time) of the system. See Section 5.3 for details. All shared memory statistics do not include accesses to lock variables.

		Ocean		Water-Nsq		Raytrace		MP3D	
		snoop	dir	snoop	dir	snoop	dir	snoop	dir
Speedup (normalized)		1.01	1.01	1.03	1.07	1.02	1.46	1.20	1.34
Push Behavior	pushes recvd	602	637	33372	32325	49458	50358	590517	540754
	% accepted	100%	100%	99.98%	100%	65.02%	71.22%	98.51%	97.48%
	% used	100%	100%	93.05%	93.97%	94.95%	67.34%	97.91%	93.61%
	% evicted	0%	0%	0.01%	0%	03.04%	28.33%	1.51%	1.51%
	% invalidated	0%	0%	6.94%	6.03%	02.01%	4.33%	0.58%	4.88%
Snoop Bus traffic (normalized)		1.00		0.86		1.11		0.87	
Directory System Network Traffic (normalized)	total msgs		1.00		0.82		0.99		0.79
	requests		1.00		0.86		1.12		0.85
	responses		1.00		0.88		0.99		0.82
	interventions		1.00		0.57		0.78		0.66
L2 miss counts in CS (normalized)	remote dirty load	0.19	0.65	0.11	0.12	0.13	0.06	0.14	0.18
	overall	0.78	0.77	0.26	0.13	0.53	0.52	0.15	0.13
Stall contributions. These are not normalized and are a percentage of the parallel cycle time for the specific optimization *CS-shared-mem-stall*: contribution of shared memory accesses within critical sections to overall shared memory access stall time *CS-DirtyMiss-Stores*: contribution of misses to dirty data in remote caches and of store latencies within critical section to the shared memory access time.									
Execution time contribution of shared memory accesses for IQLs		60.43%	56.49%	15.07%	16.92%	10.95%	9.24%	53.13%	58.32%
CS-shared-mem-stall	IQLs	0.34%	0.61%	31.67%	48.47%	43.16%	65.16%	55.69%	63.85%
	w/ Spec Push	0.27%	0.48%	8.9%	19.76%	24.86%	43.20%	26.96%	30.49%
	gain over IQL	1.20	1.24	4.66	3.65	1.12	1.57	3.58	3.76
CS-DirtyMiss-Stores	IQLs	0.13%	0.26%	31.34%	47.40%	33.06%	57.45%	53.28%	63.03%
	w/ Spec Push	0.04%	0.15%	5.21%	12.62%	3.52%	15.82%	14.79%	23.81%

6.2 Speculative push performance

We compare the speculative push performance with IQLs as the base case. The interesting benchmarks are Raytrace and MP3D. Even though raytrace is a candidate for substantial speedups with the speculative push mechanism, we see a small 2% gain for the bus configuration. The directory configuration sees a speedup of 46%. This can be explained by analyzing the push behavior. With raytrace, due to the inability

of the cache to sink a push, the push rejection rate is quite high (35 to 30%). Thus, a large number of pushes are being performed without any gain. In addition, the network traffic goes up due to the extra failed pushes. In the snoop configuration, a cache-to-cache transfer is faster than accessing memory. Thus, even though the misses to remote caches goes down substantially, there is a performance loss due to a longer access latency to memory (an increase in misses serviced from memory). The increased bus contention and longer access latencies to memory more or less negate the performance gains due to the successful pushes. With the directory, it is faster to access memory than a remote node (2-hop vs. 3-hop). Thus, even though we see rejected pushes, the subsequent miss is serviced by the directory and we still see a performance improvement of 46%.

MP3D shows speedups of 20% and 34% for the bus and directory respectively. This benchmark has frequent synchronization with migratory data. Speculative push consistently captures this behavior and succeeds in eliminating a large portion of 3-hop transactions and access latencies within critical sections. This is precisely the behavior we were expecting to capture. Since critical section accesses contribute to a large factor of execution time, we see a corresponding speedup. The *CS-shared-mem-stall* row in the table shows the contribution of critical section accesses to shared memory stall time. As we can see, there is a consistent reduction in the time spent in critical sections across all benchmarks, at times by more than 50%.

It is important to note that the speculative push mechanism subsumes the write-fault which generally occurs with read-modify-write sequences since the cache line is pushed in an exclusive state. To a large extent, this in turn reduces the number of invalidate requests which would have been generated due to the write-faults.

7 Concluding remarks

In this paper we have studied two mechanisms for reducing communication latency inside critical sections. The IQL mechanism is not new, and our results corroborate earlier published results showing large gains for benchmarks that either have high lock contention or heavy network contention in general and substantial locking. However, the novel speculative push mechanism showed substantial additional benefits.

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 20% for the bus and 34% 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 high contention locks (such as Raytrace), show large speedups in some cases. The speculative push

mechanism can provide further speedups in overall performance by substantially reducing the network traffic and 3-hop transactions.

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

References

[1] *MIPS R10000 Microprocessor User's Manual Version 2.0.* 1996.

[2] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 204–215, February 1997.

[3] 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.

[4] 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.

[5] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *DFVLR Conf. 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Springer-Verlag LNCS 295*, June 1987.

[6] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 15–26, May 1999.

[7] Luiz Andre Barroso, Kourosh Gharachorloo, Andreas Nowatzyk, and Ben Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 3–14, January 2000.

[8] 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.

[9] Tony Brewer and Greg Astfalk. The evolution of the HP/Convex Exemplar. In *Proceedings of the 42nd IEEE Computer Society International Conference (COMPCON)*, pages 81–86, February 1997.

[10] 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.

[11] 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.

[12] Compaq Computer Corporation. *Alpha Architecture Handbook Version 4.* 1998.

[13] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[14] David E. Culler and Jaswinder Pal Singh with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufman, San Francisco, CA, 1999.

[15] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 15–26, May 1999.

[16] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[17] 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.

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

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

[20] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1992.

[21] 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.

[22] 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.

[23] 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.

[24] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 15–26, May 1999.

[25] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the Eighth Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[26] 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*, May 1999.

[27] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[28] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John L. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[29] 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.

[30] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit Via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[31] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[32] 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.

[33] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[34] David K. Poulsen and Pen-Chung Yew. Data Prefetching and Data Forwarding in Shared Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume II (software), pages 276–280, August 1994.

[35] Ravi Rajwar, Alain Kagi, and James R. Goodman. Improving the Throughput of Synchronization by Insertion of Delays. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 168–179, January 2000.

[36] Parthasarthy Ranganathan, Kourosh Gharachorloo, Sarita Adve, and Luiz Andre Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-Of-Order Processors. In *Proceedings of the Eighth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 307–318, October 1998.

[37] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 285–298, December 1995.

[38] 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.

[39] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[40] 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.

[41] 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.

[42] J. Skeppstedt and P. Stenström. A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[43] Per Stenström, Mats Brorsson, and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[44] Pedro Trancoso and Josep Torrellas. The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume III (software), pages 79–86, August 1996.

[45] 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.

[46] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, Apr. 1996.