Computer Sciences Department

Lock-Based Programs and Transactional Lock-Free Execution

Ravi Rajwar
James Goodman

Technical Report #1440

April 2002

UNIVERSITY OF WISCONSIN MADISON

University of Wisconsin-Madison Technical Report #1440 (April 2002)

Lock-Based Programs and Transactional Lock-Free Execution

Ravi Rajwar and James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison
{rajwar,goodman}@cs.wisc.edu

Abstract

Writing shared-memory multithreaded applications requires a careful trade-off between programming ease and performance, largely due to subtleties in coordinating correct access to shared data. To ensure correctness, programmers rely on conservative locking, often at the expense of performance. The resulting serialization of threads is a constraining bottleneck to achieving high performance in multithreaded programs. The growing popularity of multithreaded architectures and multiprocessors is expected to result in more multithreaded applications. Programmability of such applications thus becomes an important problem. We investigate architectural support for improving the programmability, performance, and fault-tolerance of multithreaded programs.

This paper revisits the notion of an optimistic lock-free execution of multithreaded applications. In a lock-free execution, shared objects are not locked when accessed by various threads. We propose Transactional Lock Removal (TLR) *and show how a program that uses lock-based synchronization can be correctly executed by the hardware in a lock-free manner without programmer support or software changes even with conflicts. This is done using a key notion of logical time for conflict resolution, modest hardware, and features already present in many modern computer systems. In addition to providing non-blocking behavior, the mechanism provides a guarantee of starvation freedom without lock-acquisitions and guarantees conditional wait-free execution.*

The many benefits of the proposed technique include improved programmability, fault-tolerance, and performance. The system achieves benefits of lock-free data structures while allowing programmers to use the familiar lock-protected critical section for writing programs.

1 Introduction

Programming complexity is the single most significant problem in writing shared-memory multithreaded applications. Programmability is a key challenge and its importance will only grow as hardware multithreaded architectures and shared-memory multiprocessors become more common. Although threads simplify the conceptual design of programs, care and expertise are required to ensure correct interaction among threads. Programming complexity is generally higher than for most single threaded programs because sharing of data objects among various threads requires complex reasoning. Errors in reasoning about appropriate synchronization among threads results in incorrect program execution, and may be extremely subtle. The concept of a transaction serves as an intuitive model for writing such programs.

Though the precise meaning of the term *transaction* varies depending upon the context, a transaction is the embodiment of the concept of *atomicity*. Atomicity has two properties: serializability and failure atomicity. *Serializability* is analogous to sequential consistency [22] with regard to memory operations—the result of executions of transactions is as if there were some global order in which they had executed serially. *Failure-atomicity* is the all-or-nothing property—the transaction must either be executed to completion or it must not appear to have executed at all.

While a transaction is a fundamental concept in computer science, its implementation depends on a combination of hardware and software. Processors today provide a restricted form of such transactions in the instruction set. Examples are atomic read-modify-write operations on a single word such as TEST&SET and LOAD-LINKED/STORE-CONDITIONAL synchronization primitives. While these instructions cannot provide general transactional functionality because of their limited size, they can be used to implement *critical sections*. Critical sections enforce mutually-exclusive access among threads to shared objects—only one thread is allowed to operate on the object at any given time—and thus trivially satisfy serializability.

Failure atomicity is achieved by logging modifications performed by the transaction, and then making these changes visible instantaneously using an atomic operation to commit the transaction. For all but the simplest transactions, this process is complex. Since achieving failure atomicity is difficult, critical sections do not provide this capability. The desirability of failure atomicity is however not in question, especially in highly concurrent systems.

Critical sections are the most popular abstraction for reasoning about correctness and coordinating data sharing among threads. Locks are most commonly used to implement critical sections. A lock is a software construct associated with a shared object and determines whether the shared object is currently available. Nearly all architectures support instructions for implementing lock operations and thus locks have become the synchronization primitive of choice for programmers and are extensively used in operating systems, database servers, web servers, and other software.

Unfortunately, critical sections implemented using locks have two fundamental problems: 1) complex trade-off between programmability and performance, and 2) inherent limitation of the locking construct.

The complex trade-off between programmability and performance exists because programmers have to reason about data sharing during code development using static analysis. Often the programmer is unaware of the critical section's dynamic behavior. This limitation and the complexity in reasoning about data sharing results in the programmer's using conservative synchronization to easily guarantee correct execution under unknown runtime conditions. Such use leads to dynamically unnecessary serialization because such locks are frequently unnecessary for a correct execution. While conservative use guarantees correctness, provides stable software, and leads to faster code development, it also masks parallelism. Fine-grain locks may sometimes help performance but make code fragile and error prone. Coarse grain locks help write correct code and reduce errors but hurt performance. Additionally, locks can contribute to significant overhead and degrade overall system performance.

The inherent limitation of the locking construct stems from the notion of the programmer-specified wait while some thread is in the critical section. A lock marked as *held* forces other threads to wait for the lock value to be *free*. This limitation manifests itself in 2 catastrophic ways:

a) *Poor system wide interactions with thread scheduling.* If a thread holding a lock is de-scheduled by the operating system, other threads waiting for the lock cannot proceed because the lock is not free. In a high concurrency environment, all threads may wait until the de-scheduled thread runs again. This results in *convoying* (a convoy of waiting threads is formed) and may result in a severe problem of *priority inversion* (no thread may ever proceed). A *non-blocking* implementation guarantees *some* process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes [10]. The non-blocking condition guarantees the system as a whole makes progress despite individual halting failures or delays.

b) *Fault-tolerance limitations.* If a thread holding a lock terminates due to a fault, other threads waiting for the lock never complete as the lock is never free again. This problem is catastrophic in a transaction oriented environment where threads are largely independent except while accessing some critical shared structures. Data modified within the critical section are left in an inconsistent state resulting in application failure (critical sections lack failure atomicity). A wait-free implementation guarantees *any* process can complete any operation in bounded number of steps, regardless of the execution speeds of other processes [10]. A wait-free implementation adds starvation freedom to the non-blocking condition.

In spite of these fundamental problems of locks, a lack of competitive alternatives has led to a nearly universal use of lock-based critical sections for synchronizing thread accesses. These problems are becoming increasingly important and solutions must be found to enable programmers to exploit hardware thread parallelism efficiently and robustly while avoiding such limitations.

This paper proposes the use of modest hardware to convert lock-based critical sections transparently and dynamically into *lock-free optimistic transactions* and the use of a *timestamp-based fair conflict resolution* scheme to provide transactional semantics and starvation-freedom. We believe this approach, called *Transactional Lock Removal (TLR)*, can cleanly and effectively address the problems outlined earlier. By converting critical sections to optimistic transactions, concurrency is exposed independent of lock granularity. Since the execution is optimistic, data conflicts (of all threads accessing a given memory location simultaneously, at least one thread is writing to the location) must be detected. We use existing cache coherence protocols to detect data conflicts. If the speculative data can be buffered using local caches, all non-conflicting transactions proceed and complete concurrently without any serialization or dependence on the lock. Transactions experiencing data conflicts are automatically ordered appropriately without interfering with non-conflicting transactions and without lock acquisitions. TLR, like many speculative execution schemes, relies on the ability to buffer speculative memory state.

Speculative Lock Elision (SLE) [30] is a recent hardware proposal for eliding lock acquires from a dynamic execution stream, thus breaking a critical performance barrier by allowing non-conflicting critical sections to execute and commit concurrently. SLE showed how lock-based critical sections can be executed speculatively and committed atomically without acquiring locks if no data conflicts were observed among the critical sections. While SLE provided concurrent completion for critical sections accessing disjoint data sets, data conflicts result in threads restarting and acquiring the lock serially. In the presence of conflicts, SLE experiences locking overhead and cannot provide failure atomicity because it acquires locks. Thus, SLE still suffers from the key problems of locks.

TLR uses SLE to elide locks and construct an optimistic transaction, and uses a timestamp-based conflict resolution scheme to guarantee lock-free execution. A single, globally unique, timestamp is assigned to all memory requests generated for data within the optimistic lock-free critical section. On a conflict, some threads may restart (employing hardware speculative execution) but the same timestamp determined at the beginning of the optimistic lock-free critical section is used for subsequent re-executions until the critical section is successfully executed. A timestamp update occurs after a successful execution. This mechanism guarantees each thread will eventually win any conflict by virtue of having the lowest timestamp in the system and thus will succeed in executing its optimistic lock-free critical section.

Unlike SLE, TLR does not suffer from the limitations of locks because it uses timestamps to provides a guaranteed lock-free execution even in the presence of conflicts. Programmers can use coarse-grain and conservative locks while obtaining the behavior of fine-grain synchronization without locking overhead. Since locks are not acquired, the inherent software wait for a lock variable is eliminated and a non-blocking execution is achieved along with failure-atomicity. Further, the timestamp-based conflict resolution scheme guarantees all threads eventually succeed in a bounded number of steps, thus providing a conditionally *wait-free* behavior. The proposed hardware is modest and the scheme does not require changes to the underlying cache coherence protocol except requiring additional payload in some messages.

Software lock-free schemes using lock-free data structures have been proposed to address the inherent limitations of locking [11, 5, 39, 3]. Lock-free schemes optimistically provide concurrent data structure implementations without requiring a critical section or software wait on a lock. These schemes often require more complex operations than critical sections and rely on programmers to write appropriate code. Programmers have to reason about correctness in the presence of complex data structures. These alternatives commonly suffer from difficulty of use, complex programming methodologies, and often high software overheads, thus aggravating the complexity/performance trade-off. With TLR, programmers continue using the familiar lock-based critical section while automatically obtaining the benefits of lock-free data structures.

In Section 2 we discuss the concepts behind Transactional Lock Removal and show how lock-free transactional semantics of lock-based programs are transparently obtained. Section 3 provides details of the mechanism. Implementation strategies are discussed in Section 4 and experimental results in Section 6.

2 Transactional Lock Removal: Concepts

We first describe the basic idea of extracting a lock-free optimistic execution from lock-based codes. In Section 2.2, we discuss the role of cache coherence protocols in TLR. Then we give the concepts for providing liveness in an optimistic execution in a protocol independent manner. We list implementation-independent invariants in Section 2.5 to help implement TLR on systems.

2.1 Transparently executing critical sections as lock-free optimistic transactions

Proposals for providing generalized transactional semantics for multithreaded programs exist in the form of architectural support [12, 37] or complex software mechanisms [34]. Architecture proposals have required extensions to instruction sets and new programming methodologies while software proposals suffer from high overhead of coordinating transactions in software. Both approaches suffer from complexity in using the provided transaction mechanisms. Programmers must learn new programming styles and correct programming becomes difficult because of complex usage. Programmers can no longer use the convenient critical section abstraction for reasoning. Further, existing applications employing lock-based critical sections for synchronization cannot benefit. These techniques do succeed in addressing the inherent limitations of locks by avoiding use of locks.

SLE demonstrated how a lock-based critical section can be predicted, and then committed atomically without lock acquires (without even requesting exclusive permissions for the lock) if no data conflicts were observed and the data is cached. Temporal silence of locks enables one to avoid writing the lock itself. The idea behind SLE involves using the cache coherence protocol to obtain appropriate permissions on necessary cache lines, modifying data speculatively if needed, and then providing the appearance of instantly executing the critical section by making updates visible to other processors at a single commit point. Critical sections are predicted by observing the dynamic instruction stream and identifying patterns for possible lock operations, and the lock operations are elided by exploiting properties of temporal silence [30]. The elision can be applied even to nested locks as long as their temporal silence holds. Properly nested locks are trivially handled this way and improper nesting can be handled subject to temporal silence conditions.

Our approach is different from the earlier approaches in two significant ways.

1. Rather than change the programming model to obtain transactional semantics, we change the hardware implementation to transparently provide such semantics. A critical section is treated as a transaction and optimistically executed without lock operations. The intuition lies in treating locks as defining the scope of a transaction, using a conflict resolution scheme to correctly order conflicting transactions, and using a technique such as Speculative Lock Elision to commit the transaction atomically.
2. Our conflict resolution scheme provides starvation-freedom and thus can provide conditionally *wait-free* behavior of lock-based codes. The behavior is conditional only because of potential resource limitations.

By maintaining the programming interface of a familiar lock-protected critical section, programmers do not have to learn new ways to write programs. Additionally, existing legacy code using critical sections can directly benefit from this. By treating critical sections as lock-free optimistic transactions, inherent concurrency in the transactions is exposed, independent of lock granularity. By using a fair conflict resolution scheme, we guarantee high-performance lock-free and conditional wait-free executions.

Figure 1 shows the idea of inserting all memory operations within a critical section (without locks) atomically into a global memory ordering (appropriately interleaved with other memory operations) thus

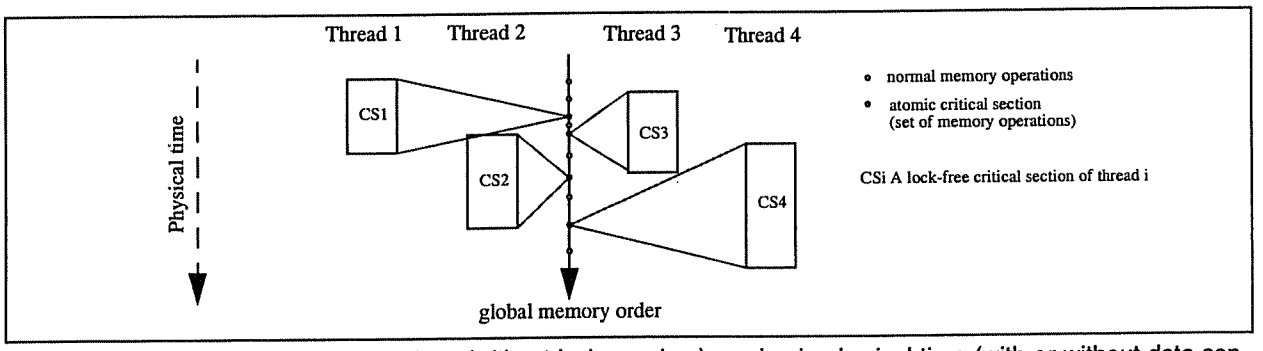

Figure 1. While critical section executions (without lock acquires) overlap in physical time (with or without data conflicts), each logically appears to be inserted atomically and instantly in a global memory ordering.

automatically satisfying the transactional requirements of serializability. Failure atomicity is guaranteed because failing transactions do not expose their updates.

2.2 Exploiting hardware cache coherence protocols

When a data block not present in the cache is read, the cache coherence protocol is triggered and the block brought into the cache. Writing the data block requires exclusive permissions block. This is done by invalidating all shared copies of the block in other caches. Once other caches have been invalidated, the local copy can be modified if necessary. Exclusive permissions force other processors to request the latest architecturally correct data copy from the exclusive owner of the block. This is how all invalidation-based cache coherence protocols work.

The above protocol functionality provides us with two capabilities. First, accessed data is easily tracked by local caching. Second, data conflicts are detected trivially—writes to shared data trigger invalidation messages to sharers, and accesses to exclusively owned blocks automatically go to the exclusive owner.

During speculative execution of an optimistic transaction, processors track all speculatively accessed blocks using their local caches. A data conflict occurs if either a speculatively accessed block receives an invalidation message from another processor (the other thread is writing data the current thread has accessed), or an external request from another thread is received for data exclusively owned and speculatively modified by the current thread (the other thread is accessing data the current thread has speculatively written). On a data conflict, appropriate recovery action is performed. Speculatively written data is buffered and not exposed to other processors until the speculating processor successfully completes. If another thread requests data that has been speculatively written by the current processor, the non-speculative data must be provided in response and the speculation is terminated/restarted.

If sufficient cache resources do not exist, processors cannot use cache coherence protocols for tracking data accesses and for detecting data conflicts. Insufficient cache resources trigger a misspeculation and the lock must be acquired for correct execution. In this case, we cannot apply our optimistic lock-free transaction transformation. Similarly, uncached memory operations also trigger a misspeculation. Any operation

that cannot be undone (in the event of a misspeculation), cannot be speculatively executed. However, a correct execution can *always* be guaranteed by turning off speculation and acquiring the lock.

The issue of sufficient buffering resources is an engineering decision involving a trade-off. While more resources can be provided, caches today are sufficiently large to buffer most critical sections. Another constraint is the duration of the critical section execution in terms of the scheduling quantum. While this again is an engineering decision, most critical sections we are aware of typically access a handful of cache lines and easily execute within a scheduling quantum (threads may still be de-scheduled within the critical section). A correct execution is always guaranteed independent of the resource availability because the lock can always be acquired.

2.3 Analogy with database concurrency control

We draw an analogy between our use of hardware coherence protocols and concurrency control concepts in database systems. Two conventional lock modes in database systems for lockable objects are *shared* mode and *exclusive* mode. An object locked in shared mode can be read by multiple transactions while an object in exclusive mode can be read and/or written by a single transaction that exclusively owns the object. If a transaction needs to write a shared-lock object, the lock must first be upgraded to an exclusive lock. Similarly, cache lines are generally of two types: *shared* lines (*Owned* and *Shared*) and *exclusive* lines (*Modified* and *Exclusive).* A shared state line to be written must first be upgraded to an exclusive line.

A shared object, protected by a lock, has multiple data fields, each field corresponding to the cache line the field resides in. Each cache line can be viewed as another finer-grain lock and the cache coherence protocol as a hardware lock manager. TLR takes such an object, splits it automatically into the data fields (cache lines), and then performs concurrency control on the individual cache lines (the fields). The mechanism acquires appropriate permissions (exclusive or shared) on the lines and executes the transaction. The object lock is automatically converted into multiple fine-grain locks with each lock per cache line itself. However, no lock overhead exists now because each cache line behaves as a lock itself.

2.4 Guaranteeing liveness of a lock-free optimistic transaction

For guaranteeing liveness, forward progress must occur. In the absence of data conflict, TLR behaves same as SLE and liveness of a lock-free transaction is trivially guaranteed because of no misspeculation. A data conflict triggers a misspeculation and the processor restarts executing the speculative lock-free critical section. SLE trivially guarantees liveness in the presence of conflicts by acquiring locks if necessary but we want to avoid lock acquires. This paper concerns transactions accessing conflicting data sets at the same time and with forward progress techniques that do not rely on lock acquires..

Example of livelock of a lock-free optimistic transaction

Figure 2 shows livelock. Two processors, P1 and P2, write shared memory locations, A and B, in their critical sections and each does so in reverse order. Initially, P1 has block A in exclusive state (M) and P2

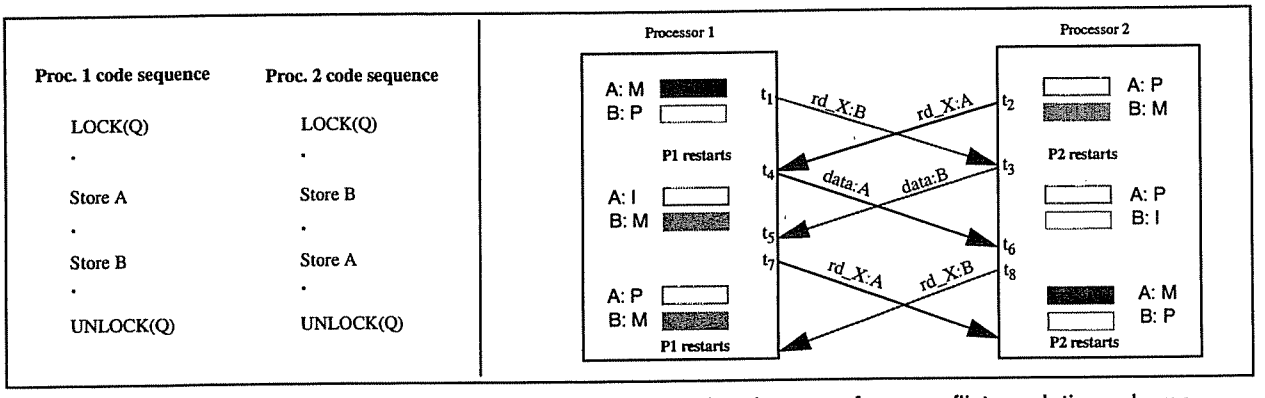

Figure 2. Livelock in a lock-free execution of a critical section in the absence of any conflict resolution scheme.

has block *B* in exclusive state (M) and both blocks have been speculatively accessed within the critical section. At t_1, P1 issues a request for exclusive permissions (`rd_X`) for *B* and at t_2 P2 issues a `rd_X` for *A*. The blocks transition to pending (P) states because data is unavailable. P1 receives P2's `rd_X` for *A* at time t_4. This is detected as a data conflict because another processor is writing the block P1 speculatively accessed. P1 processes P2's request, invalidates the local copy, and responds with the architecturally correct non-speculative data for *A*. P1 restarts speculative execution because the data conflict prevents a successful atomic execution. A similar sequence of events occurs at P2 for block *B*. This sequence can happen indefinitely with no processor making forward progress because each processor repeatedly restarts the other processor. Forward progress cannot be guaranteed and the lock must be acquired.

Ensuring successful atomic execution

A successful atomic execution of a transaction is ensured by using the cache coherence protocol to acquire exclusive ownership of memory locations accessed by the transaction. If the thread executing the transaction does so for all required memory locations, the thread can execute, modify the locations and make all updates visible atomically to other threads. In Figure 2 no thread can acquire exclusive ownership of all necessary memory locations (*A* and *B*) simultaneously and threads repeatedly restart. Once exclusive permissions are obtained, they must be retained until the transaction executes successfully.

The following conditions must be met if exclusive permissions are retained:

1. To ensure liveness, deadlock must be avoided. Deadlock can occur because threads acquire and retain exclusive ownership of different cache lines. Deadlock freedom is achieved by using a uniform conflict resolution scheme and by ensuring a thread does not retain exclusive ownership of any memory location if the thread cannot acquire exclusive ownership of some other required location.
2. To ensure starvation freedom, all threads must eventually succeed. This is achieved by using an appropriate conflict resolution scheme guaranteeing all conflict losers eventually become winners.

We discuss these conditions in three steps. First, we discuss distributed conflict resolution. Second, we show how to provide starvation-freedom by using an appropriate and simple conflict resolution scheme. Third, we present *one* way to *retain* exclusive ownership of cache blocks and provide deadlock freedom

independent of the cache coherence protocol. We will then show how all of this fits together to provide a starvation-free and lock-free execution of the example of Figure 2.

2.4.1 Distributed arbitration for conflict resolution

A distributed arbitration mechanism determines priorities among conflicting requests that enable a winner to continue executing and losing threads may misspeculate and restart the transaction. We use the cache coherence protocol for performing distributed arbitration because the protocol already provides mechanisms for communicating data conflicts automatically to all concerned processors. All requests sent to the coherence protocol within the optimistic transaction carry a globally unique identifier associated with the requesting node. The identifier can be as simple as the static processor identifier itself. The identifier helps resolve conflicts by determining priorities among conflicting transactions. Conflict resolution is performed locally by processors involved in the conflict and involves a comparison of the local node identifier with the incoming conflicting request identifier.

2.4.2 Starvation-free conflict resolution using timestamps

For starvation freedom, the conflict resolution mechanism must guarantee all contenders eventually succeed, for example by ensuring a first-come first-serve arbitration with losers retaining their positions in priority. A static identifier, such as a processor identifier, is inherently unfair and cannot provide starvation freedom. We use *timestamps* for fair arbitration—the contender with a lower timestamp wins the conflict. Timestamps inherently capture the notion of progressing time and easily provide dynamically changing priorities. Our timestamps are taken from the *local* real-time system clock. The timestamps are made globally unique by appending processor identifier bits to the bits extracted from the system clock. Our timestamps are loosely synchronous: exact synchronicity is not required. This is one way of locally generating unique timestamps and other schemes exist [21].

Importantly, a timestamp is assigned to the entire critical section execution instance; all memory operations within the critical section are assigned the *same* timestamp. If speculative execution fails due to losing a conflict arbitration, the same timestamp is *retained* and *reused* for re-execution and a new timestamp is assigned only after a successful execution. The forward running clock guarantees the new timestamp is larger than the earlier one. In any execution, some processor always wins all its conflicts and a failing processor will eventually have the lowest timestamp by virtue of it being re-used. This processor will then win all its conflicts in its transaction and thereby successfully complete the transaction. Lamport [21] used timestamps derived from *logical clocks* to implement distributed mutual exclusion with a starvation-freedom guarantee. Our use is similar to Lamport's use of logical clocks but we only require timestamps for conflict resolution while Lamport used timestamps for explicitly ordering the execution of mutual exclusion regions among different processors. Note this implies that with TLR, transactions that conflict in their

data sets but do not actually observe any detected conflicts during their execution can execute in any order, independent of the timestamps of the transactions.

2.4.3 One way to mask conflicts and retain acquired exclusive ownership

As discussed earlier, a successful lock-free atomic execution requires a processor to acquire and retain exclusive ownerships of appropriate cache blocks if necessary. As per the underlying coherence protocol, a processor with an exclusively-owned block receives and must respond to subsequent requests for the block. The processor can retain exclusive ownership by sending a negative acknowledgement (NACK) to the requestor thus forcing the requestor to retry at a future time. However, many modern coherence protocols do not support NACKs.

For a general solution independent of coherence protocols, we describe one mechanism to retain exclusive ownership. To retain exclusive-ownership, the processor buffers the incoming request and does not apply it right away. Thus, while the system assumes the request has been applied, the processor masks any conflict by deferring the request for a bounded time.

In modern coherence protocols (both snooping and directory), the request phase is split from the response phase. The data response from the responder (memory or current block owner) may appear an arbitrary time later than when the request is ordered by the coherence protocol. The protocol transitions its state machine based on requests received at the coherence point. If two processors P1 and P2 both issue `rd_X` requests for the same block and P1's request is ordered first, P2's request is sent to P1. However, P1 may not have the data yet and its block state is pending. P1 buffers P2's request and services it when P1 receives the data response. This can be visualized as a chain of processors linked together.

We use the same concept of buffering requests but generalize it by also deferring lower priority requests for blocks exclusively-owned and stable (i.e., the data is available). Note, in the base protocol, only requests to pending blocks (i.e., the data is unavailable) are buffered.

If the speculating processor acquires all necessary blocks in exclusive owned state, wins all conflicts and retains permissions on the blocks by deferring incoming requests to these blocks, the processor can appear to execute its transaction instantly. Once successful, speculation is committed and all deferred requests are applied in a first-in first-out manner. If the speculating processor loses any conflict and cannot acquire ownership of a cache block, the processor must relinquish all retained ownership of other data blocks and service all deferred requests. This ensures deadlock freedom—a losing processor cannot forcibly *retain* any exclusive ownership. A requestor losing a conflict and being deferred must be informed about the decision as the requestor must relinquish any retained exclusive ownership. This process, while not requiring coherence protocol transition changes, does require additional messages (Section 3.3).

2.5 Implementation-independent invariants

Based on our discussion above, we outline three implementation independent invariants. If these invariants are provided by a system, TLR can be implemented.

Invariant 1: The same timestamp is used for all requests within a given transaction. The timestamp must be retained following an unsuccessful speculation and must be updated in a strictly monotonic order following a successful optimistic transaction execution.

Invariant 2: The lowest timestamp request must never be deferred

Invariant 3: On being deferred, a processor must release all its retained exclusive ownerships

Invariant 1 provides conditions for construction of fair conflict resolution guaranteeing all losing contenders eventually become winners. Invariant 2 guarantees the contender with the lowest timestamp will eventually complete. By not being deferred, the contender can retain exclusive ownership on the necessary blocks and will eventually acquire all necessary blocks. Invariant 3 guarantees deadlock freedom. On being deferred, the processor must release all retained ownerships (by simply servicing any deferred requests) before re-executing the transaction.

The precise implementation for providing these invariants is a function of the coherence protocol implementation. In Section 2.4.3 we gave one way to enforce invariants 2 and 3 without requiring coherence protocol changes nor any assumptions about the protocol. We revisit these invariants in Section 3 where we present details of one way to implement our scheme.

The three invariants collectively provide the following two guarantees:

1. A processor will eventually have the lowest timestamp in the system
2. A processor with the lowest timestamp in the system eventually has a successful transactional execution

In summary, *in a bounded number of steps, a node will eventually have the lowest timestamp for all blocks it accesses and operates upon within its optimistic transaction and is thus guaranteed to have a successful starvation-free lock-free execution.*

2.6 Example revisited

We now put together the three steps discussed above to ensure starvation-free conflict resolution and a lock-free execution by revisiting the earlier example of Figure 2. Figure 3 shows two processors, P1 and P2. P1 has block *A* in exclusive-owned (M) state and P2 has block *B* in exclusive-owned (M) state, similar to Figure 2. Additionally, each processor P1 and P2 now has a unique identifier ID1 and ID2 respectively where ID1 < ID2 (P1 has higher priority). All memory operations within the optimistic transaction are assigned the same unique identifier. Therefore, P1's `rd_X` for *B* has ID1 appended and P2's `rd_X` for *A* has ID2 appended. On receiving P1's request, P2 compares its identifier ID2 with the incoming request identifier ID1. Since the incoming request has higher priority, P2 services the request without delay and

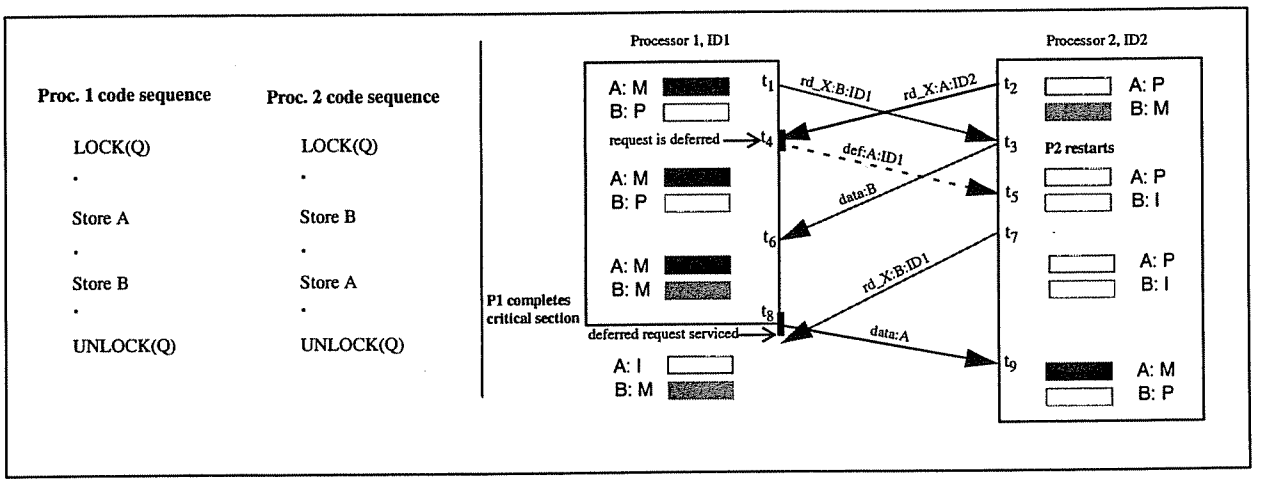

Figure 3. Live-lock free execution in the presence of conflicts using a fair conflict resolution scheme.

responds with block B (the non-speculative value). On applying the request, a data conflict is triggered and P2 restarts execution of its optimistic transaction. Now, at P1 a different sequence of events occurs. P1 receives P2's `rd_X` for *A* at t_4. Since ID1 < ID2, P1 wins the conflict and buffers the request. The cache block state for *A* stays M. For the protocol, P2 owns the block but P1 has locally masked the conflict and will guarantee an atomic execution thus maintaining any memory consistency model requirements. P1 also sends a marker message along with its identifier informing P2 of the deferral and P2's conflict loss. At t_6, P1 receives data for block *B* from P2. Note, P1 has acquired and retained permissions on both blocks *A* and *B* and can execute the optimistic transaction successfully. At t_8, when P1 finishes the transaction and architecturally commits its speculative state, P2's buffered request is serviced and P1 responds with the latest architecturally correct data. Meanwhile, P2 has restarted and is re-executing its transaction. P1 on successfully committing its transaction updates the local timestamp to a value greater than what it was (and > ID2). This update is similar to Lamport's logical clock construction and guarantees a fair conflict resolution scheme. The key difference between Figure 2 and Figure 3 is P1's ability to retain exclusive permissions in Figure 3.

3 Transactional Lock Removal: Details

We now discuss details of a way to implement our scheme. We discuss it in four parts: a) initiating speculation, b) generating outgoing requests, c) handling incoming messages, and d) committing speculation. We also revisit the invariants outlined earlier and show ways to enforce them.

3.1 Initiating speculation and speculative execution

The first step involves identifying a critical section as an optimistic transaction for execution. This allows us to identify the sequence of memory operations that may constitute a transaction. We use SLE for doing so. A critical section is predicted by matching instruction sequence patterns and looking for special

synchronization primitives. On a lock acquire prediction, the lock acquire is elided, a checkpoint of the architected register state is created, and speculative execution mode is entered. Cache blocks speculatively accessed within the optimistic transaction are tracked using an *access bit* associated with each line in the cache. On initiation of speculation, the local timestamp is generated (Section 2.4.2). This timestamp is appended to all misses generated within the predicted optimistic transaction and is re-used until a successful execution. Invariant 1 is partially met by doing so—the same timestamp is used for all misses generated within the given transaction instance. In speculative execution mode, instructions are speculatively retired and store operations are sent to a speculative write buffer. While exclusive ownership requests are sent to the memory system, speculatively written data is not exposed until a successful commit. On a misspeculation, speculative write buffer entries are discarded and the architected register state is restored. Within an optimistic transaction, writes to the same cache line and even same memory location can be combined.

3.2 Generating outgoing requests

In a cache supporting multiple outstanding misses, when an access misses in the local cache, a miss status handling register (MSHR) [18] is allocated and the request sent to the memory system. The block transitions to a *pending* state (i.e., a data response has not been received). Later, the data response is matched with the corresponding MSHR and data sent to the processor. The MSHR is freed and the block transitions to a *stable* state (i.e., the block is not pending and data is available). We use this existing buffer and functionality and add fields to the MSHR to track whether the request has been deferred. The local timestamp calculated above is appended to the outgoing request.

3.3 Handling incoming messages

Incoming messages from the coherence network detect all conflicts. When a processor receives a request from another processor for a cache block with its access bit set (that is part of the optimistic transaction), the block can be in a stable state or in a pending state. If the block is in a stable state, the request can be immediately serviced if necessary whereas a block in a pending state cannot be serviced as the stable copy is somewhere else. The pending state is guaranteed to transition into a stable state at some point in the future as guaranteed by any correct base coherence protocol.

Processing incoming requests and performing arbitration among conflicting nodes is specific to the implementation of the coherence protocol. Techniques for enforcing invariants 2 and 3 differ based on the underlying protocol. Implementations for NACK-based protocols (where the coherence protocol allows only one outstanding intervention for a given block in the entire system at any time and additional interventions for that block are sent a negative acknowledgement) is straightforward. The conflict winner can simply NACK the requestor and use the same mechanism to inform the loser of the conflict resolution. Implementations that support multiple outstanding interventions for a given block simultaneously require a

slightly different algorithm because the processor with the only exclusive valid copy of the block may not see the latest request for the block. This may happen because a chain of pending intervention requests for the block get constructed by the coherence protocol. The only requirement is that the invariants must be maintained by the algorithm.

In our implementation, our base protocol supports multiple outstanding interventions to the same block and does not support negative acknowledgements. We have implemented our algorithm on top of this protocol without changing the protocol but rather using the chains formed by the protocol itself to perform distributed arbitration. The algorithm makes no assumptions about the coherence protocol other than the protocol being an invalidation-based coherence protocol. Two messages are required, informing the requestor whether it has been *deferred* or *not deferred*. These messages have no coherence interaction and may use any network. For sake of brevity and because the algorithms depend on the coherence protocol itself, we do not discuss it further. However, the Appendix has an example of how such arbitration is performed for a coherence protocol using the above two messages.

An incoming message generates a misspeculation only if a conflict occurs for blocks that have been accessed in an optimistic critical section. Requests to other blocks do not experience any conflict.

3.4 Ending/Committing speculation

When a transaction end is successfully reached, SLE is invoked to commit speculative state appropriately. At this point, all appropriate permissions must be available in the local cache. Once all permissions are available, the speculative data buffered in the speculative write-buffer is exposed to the memory system. All deferred requests are serviced in a first-in first-out order and the local timestamp takes a new value (satisfying invariant 1). The register checkpoint taken earlier is discarded and execution continues. If necessary, the *access bits* in the cache are also cleared. In addition to a successful execution, speculation may end because of events such as resource constraints, certain non-speculative instructions, and operating system thread de-scheduling. If resource constraints force speculation end, the elided lock can be written and once the write is complete, all speculative cache state can be committed and execution proceeds normally without misspeculation. For other events, a misspeculation recovery is performed by restoring the register checkpoint, discarding speculatively written data, and servicing any buffered requests.

Handling the shared cache state

A processor strictly executing exclusive ownership requests for a transaction can defer all requests. A cache block in shared state is not owned exclusively by any node and a write operation to such a block requires the state to be upgraded to an exclusive ownership (Section 2.3). External exclusive-ownership requests to shared blocks cannot be deferred and trigger a misspeculation. However, a shared block can always be upgraded to an exclusive block and subsequent requests can be deferred. Thus, the shared cache

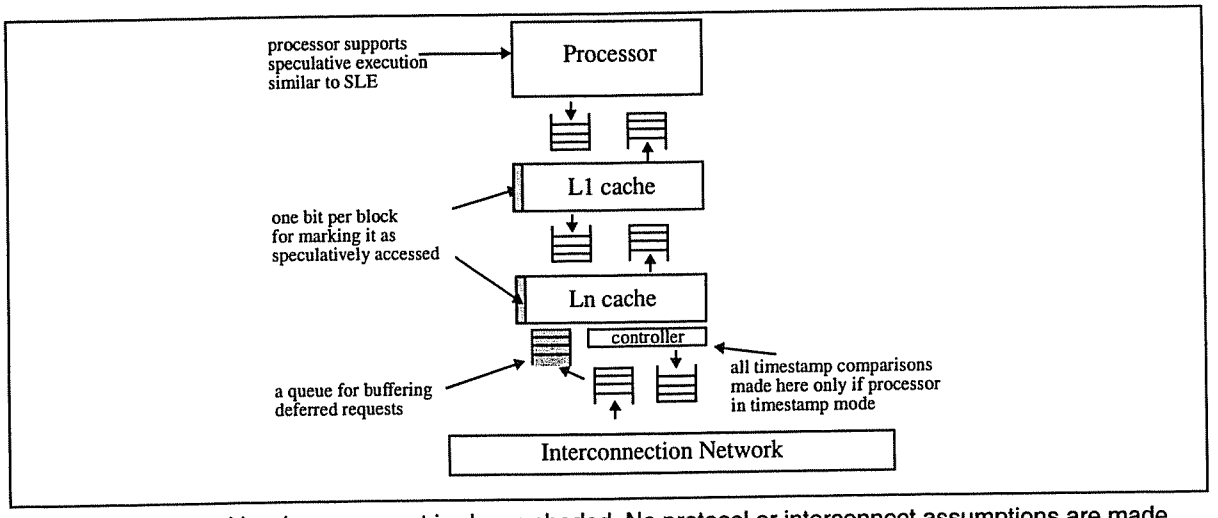

Figure 4. Additional hardware support is shown shaded. No protocol or interconnect assumptions are made.

state is trivially handled by upgrading it if necessary. A common optimization employed collapses a read followed by a write request into a single write request. This minimizes upgrade-induced violations.

4 An Implementation

Consider a shared-memory multiprocessor where every processor has a local cache hierarchy and the processors are connected together via an interconnection network. The interconnection network provides the functionality for maintaining invalidation-based cache coherence and transfer data. We make no assumptions regarding the memory consistency model of coherence protocol. The protocol may be snoop broadcast (on an ordered or un-ordered interconnect) and may be directory based (on a ordered or un-ordered interconnect).

We assume a processor with Speculative Lock Elision capability: support for predicting critical sections for lock-free optimistic transactions, local speculative buffering capability, and ability to locally detect data conflicts.

Figure 4 shows additional hardware structures as shaded regions in the local node. A local timestamp generator (involves extracting bits from a loosely synchronous system clock) is added. An *access bit* per block tracks data accessed during a predicted optimistic transaction. This set of blocks is conservative and may be larger than the actual set because the processor does not have precise information regarding whether the access was indeed from within the critical section because of out-of-order processor execution. Misses generated in this mode carry with them the local timestamp. An additional *deferred coherence input queue* is present. This queue buffers incoming requests that have been deferred by the local processor. Two messages sent only within the local cache hierarchy (*start_defer* and *end_defer*) from the processor to the cache controller are needed. The *start_defer* is sent when the processor transitions into speculative lock-free transaction mode and *end_defer* is sent on exiting such a mode. The *end_defer* mes-

sage may clear the *access bits* in the local cache hierarchy if necessary. These messages are ordered with respect to each other and multiple pairs of messages may be present in the local hierarchy. The MSHR is augmented with additional fields to track status of pending requests that may be deferred (and enforcing invariants of Section 2.5).

Since we do not change the coherence protocol (e.g., no need for NACK messages), additional *marker* messages are required for distributed concurrency control. These messages have no coherence interactions, can use any point-to-point network (e.g., data network), and do not require any ordering and are only used to perform distributed concurrency control. These messages carry sufficient information to enforce invariants outlined earlier. An example demonstrating their use is provided in the Appendix.

5 Evaluation methodology

We evaluate our scheme using three microbenchmarks and seven applications chosen from the SPLASH/SPLASH2 suites. We compare our proposal of Transactional Lock Removal to two schemes—the base case with the popular TEST&TEST&SET locks, and SLE, which provides lock-free execution only in the absence of conflicts and requires lock acquisitions in the presence of conflicts

5.1 Synthetic benchmarks

The *shared counter benchmark* consists of one counter protected by a lock where n processors increment the counter $2^{16}/n$ times. Updates are short and no inherent parallelism exists. The *multiple unique counter benchmark* consists of n unique counters protected by a single lock. Each processor updates only one of n counters. The increment is performed $2^{16}/n$ times. This benchmark is the converse case of the first benchmark. While a single lock protects the counters, there is no dependence across the various critical sections for the data itself. The *doubly-linked list benchmark* consists of a doubly-linked list with *head* and *tail* pointers and protected by a single lock. Each processor dequeues an item by removing the item pointed to by the *head*, and then enqueues it by adding it to the *tail*. A process that removes the last item sets both *head* and *tail* to NULL, and a process that inserts an item into an empty list sets both *head* and *tail* to point to the new item. The benchmark finishes when $2^{16}/n$ enqueue/dequeue operations have completed. Concurrency exists in the benchmark. A non-empty queue can support concurrent enqueue and dequeue operations. This potential concurrency is difficult to exploit by conventional means. When the queue is non-empty, each transaction modifies *head* or *tail*, but not both, so enqueuers can potentially execute without interference from dequeuers, and vice versa. When the queue is empty, transactions must modify both pointers. This behavior is not realizable in any simple way using locks, since an enqueuer does not know if it must lock the tail pointer until after it has locked the head pointer, and vice-versa for dequeuers [12, 34]. The critical section in this benchmark is non-trivial and involves extensive pointer manipulations and multiple cache line accesses. Additionally, the same lock is accessed by two different critical sections.

Table 1: Benchmarks

Application	Suite	Type of Simulation	Inputs	Type of Critical Sections
Barnes	SPLASH	N-Body	2K bodies	cell locks during tree building (nested)
Cholesky	SPLASH	Matrix factorization	tk14.O	task queues, column locks
Mp3D	SPLASH	Rarefied field flow	24000 mols, 25 iter.	cell locks
Radiosity	SPLASH2	3-D rendering	-room in batch mode	task queues, object locks (nested)
Water-nsq	SPLASH2	Water molecules	512 mols, 3 iter.	global structure
Ocean-cont	SPLASH2	Hydrodynamics	x130	conditional updates
Raytrace	SPLASH2	Image rendering	teapot	counters, work queue

5.2 Applications

The applications listed in Table 1 have been selected because they display varying lock behavior, memory access patterns, and critical section behavior including nested critical sections. `Mp3d` does frequent synchronization to largely uncontended locks and lock access latencies cannot be hidden by a large reorder buffer. `Cholesky` and `radiosity` have contended work queues that are accessed frequently. `Ocean-cont` has a conditional update code sequence. `Raytrace` has contended memory buffer queues and counter updates. `Barnes` has varied locking behavior and high contention including nested locks, while `water-nsq` has little contention. Importantly, these benchmarks have been optimized for sharing and employ fine-grain locks. They thus have little communication in most cases for low thread counts. We are interested in determining the robustness and potential of TLR even for these well-tuned benchmarks.

5.3 System configuration

Our target system configuration is shown in Table 2. Our multiprocessor system is a MOESI broadcast snooping system modeled after the Sun Gigaplane [35]. The broadcast is performed over an ordered network supporting high bandwidth snooping. We use SimpleMP, an execution-driven simulator for running multithreaded binaries. The simulator accurately models out-of-order processors and a detailed memory hierarchy in a multiprocessor configuration. To model coherency and memory consistency events accu-

Table 2: Simulation parameters

Processor	
Processor speed	1 GHz (1 ns clock)
Reorder buffer	128 entry with a 64 entry load/store queue (64-entry PC-indexed silent store-pair predictor [30])
Fetch mechanism	16 entry instruction fetch queue, 3-cycle branch mispredict fetch redirection penalty
Issue mechanism	out-of-order issue/commit of 8 instructions per cycle, loads issue as soon as ready
Branch predictor	8-K entry combining predictor, 8-K entry, 4-way BTB, 64 entry return address stack
Read-modify-write predictor	128-entry PC indexed predictor for collapsing read-modify-write sequences into a single request
Functional units	Pipelined 8 integer alus, 2 multipliers, 4 floating point units, 3 memory ports
Instruction cache	Single level, 64-KByte, 2-way associative, 1-cycle access, 16 pending misses
Data cache	Single level, 128-KByte, 4-way assoc., write-back, dual-ported, 1-cycle access, 16 pending misses, 64-byte lines
Write buffer	64-entry (each entry 64-byte wide)
Memory ordering	Total Store Ordering (similar to the Pentium 4, and Ultrasparc III)
Coherence protocol	Sun Gigaplane XB-type MOESI protocol, 120 outstanding requests. broadcast snoop latency: 20 cycles, Perfect L2: 12 cycle access, data transfer latency:16 cycles

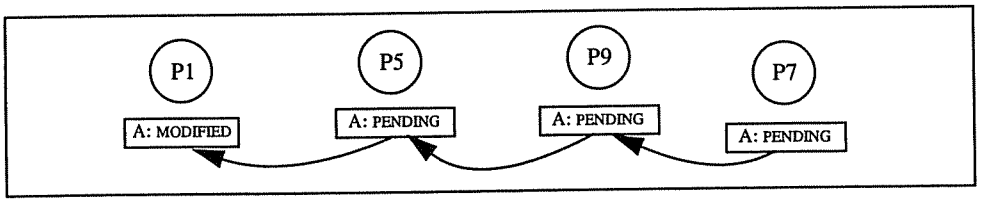

Figure 5. How a queue is maintained and data transfer occurs

rately, the processors operate (read and write) on data in caches and write-buffers. Contention is modeled in the memory system. To ensure correct simulation, a functional checker simulator executes behind the detailed timing simulator *only* for checking correctness. The functional simulator works in its own memory and register space and can validate total store ordering (TSO) implementations.

6 Results

Before discussing results, we provide an intuition behind the expected behavior of TLR. We then use this intuition to discuss the performance of microbenchmarks and then discuss application results.

6.1 Performance intuition

With TLR, processors request data without acquiring the lock and the data request is appropriately queued by the coherence protocol based on the conflict resolution policy using timestamps and some arbitration rules. Figure 5 shows four processors P1, P5, P9 and P7 requesting the same cache line A thus exhibiting true data conflict. For simplicity, let us assume the conflict resolution scheme has ordered the priorities in the following order: P1, P5, P9, and P7 as shown in the figure itself. Any other ordering would not change the basic performance intuition being demonstrated here. P1 is currently executing its optimistic lock-free transaction and has accessed cache line A. P1 defers (and buffers) P5's request for A. P9's request is buffered by P5 and P7's request is buffered by P9. P1 operates on A, complete its critical section and then respond to P5's request with the latest data for A. Subsequently, P5 operates upon the data, execute its own transaction, and on completion, respond to P9's request with the latest data for A, and so on. Thus, while processors attempt to execute the same transaction, they are automatically ordered on the data request itself and no explicit lock requests are generated. This direct transfer of data, coupled with the absence of lock requests and overhead, provides the intuition for high-performance in the presence of data conflicts. Further, while P1 is operating on A, other processors wait for the latest copy rather than introduce contention in the system by requesting locks and data. The behavior is similar to hardware queue locks [8] but now the queueing is occurring on the data itself and no lock requests are generated. Removing explicit lock requests and locking overhead under contention reduces network contention and latency.

6.2 Microbenchmarks

We first discuss microbenchmark behavior. The y-axis shows wall-clock time for completing the execution of the program. The x-axis shows varying processor count. Each data point in the graphs represents the

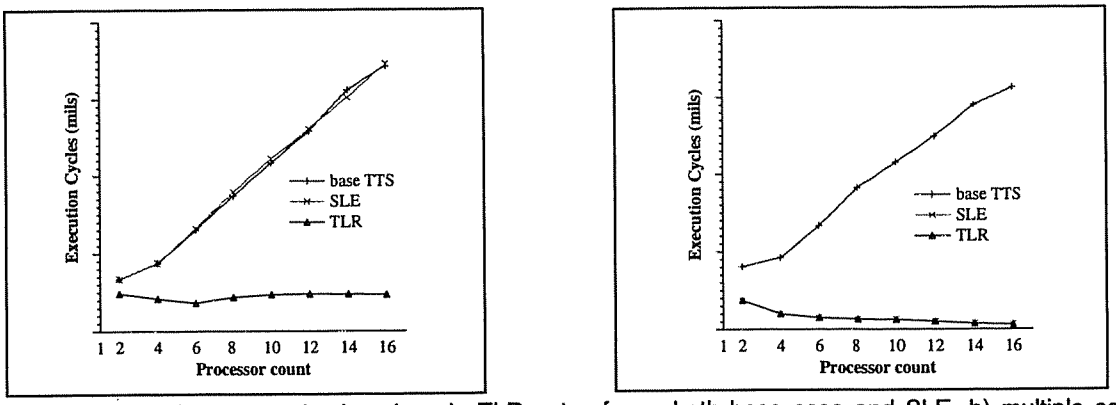

Figure 6. a) Shared counter microbenchmark. TLR outperforms both base case and SLE. b) multiple counter microbenchmark. TLR and SLE perform the same and both outperform the base case.

same amount of work done by the system. Thus, in a 16 processor system, each processor does lesser work than in a 8-processor system but the total work done in the system is the same. Figure 6 shows two counting microbenchmarks. Three lines are plotted on each graph: base scheme with TEST&TEST&SET locks, SLE, and our proposal of TLR.

Figure 6a shows *shared counter* results. The counter is successively incremented by all processors. This example has no exploitable concurrency because of true data conflicts in each critical section execution. The base TTS scheme degrades performance with increasing threads because of severe contention for the lock. The SLE scheme behaves similar to the base TTS scheme because SLE detects frequent data conflicts and falls back on the lock-acquisition sequence. Following our performance intuition discussion in Section 6.1, we get good queued behavior for TLR and increasing concurrent threads does not degrade performance the way the base schemes do.

Figure 6b shows results for N counters protected by a single lock where each counter is accessed uniquely by a given processor. Although lock contention exists, this example has exploitable concurrency because no data conflicts exist among different critical sections—the data protected is disjoint among multiple processors. The TTS scheme degrades performance as more threads run concurrently because of severe contention for the lock. TLR and SLE behave identically because of the absence of any data conflicts (Section 2.4). They experience no lock overhead and true concurrency is exploited.

Figure 7 shows results for the doubly-linked list benchmark. Processors repeatedly enqueue and dequeue elements from a queue. An enqueue and dequeue operation can occur in parallel. However, concurrent enqueues need serialization and so do multiple concurrent dequeues. This benchmark has difficult to exploit concurrency. The base TTS scheme degrades performance similar to the other microbenchmarks because of severe lock contention. SLE does not perform well either (and performs similar to the base TTS scheme) because determining when to apply speculation is difficult due to the dynamic concurrency of the benchmark. More often than not, SLE falls back to the base case of lock acquisitions because of detected

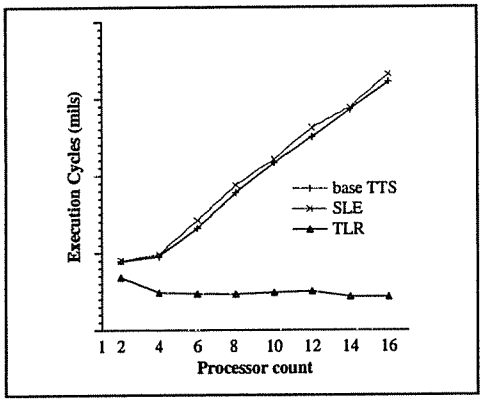

Figure 7. Doubly-linked list microbenchmark. TLR outperforms both SLE and base case..

data conflicts. Any concurrency SLE exploits is offset by locking overhead when SLE needs to acquire the lock. TLR performs quite well and at times can exploit enqueue/dequeue concurrency.

6.3 Applications

Figure 8 shows application performance. The y-axis is normalized execution time. Each benchmark has three bars: the first bar corresponds to the performance of the base case employing TEST&TEST&SET locks. The height of this bar is always 1. The second bar corresponds to SLE [30] with a restart threshold of 1. This scheme can remove lock overhead if no true data conflicts occur else the lock acquire needs to be performed. The third bar corresponds to our proposal of TLR using timestamps for conflict resolution in a lock-free execution. Both, the second and third bars are normalized to the first bar. Further, each bar is divided into two parts: contributions due to lock variable accesses and the remaining contributions. The breakup is approximate since accounting for stall cycles due to individual operations is difficult but they give an idea of the amount of performance one could potentially gain.

In all runs for the given benchmarks and configuration, TLR achieves a lock-free execution of all critical sections, in other words: the data accessed *within* a critical section fit in our local cache hierarchy. Additionally, with no programmer involvement and without any knowledge of the actual data structures used, we provide a lock-free execution transparently, a significant advance.

The benchmarks with infrequent lock contention for the base case are `Water-nsq` and `MP3D`. Our scheme behaves very similar to SLE because SLE works quite well in the absence of any data conflicts. `Ocean-cont` has lock-contention but has infrequent data conflicts because of conditional code sequences within the critical section. Because of infrequent data conflicts, SLE and our scheme behave similarly. `MP3D` has a large number of fine-grain locks and thus suffer cache misses. The lock-portion for `MP3D` thus is due to long latency read misses to the lock. When the cache is made larger, the lock accesses do not undergo cache misses (by not writing the lock, the lock variable is kept cached in shared state). This is an example where very fine-grain locks can add substantial overhead in terms of poor cache behavior.

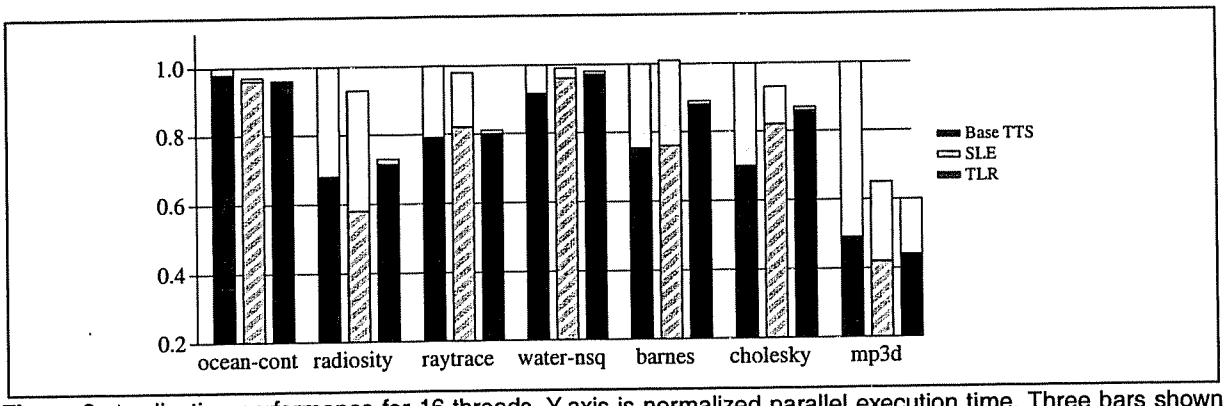

Figure 8. Application performance for 16 threads. Y-axis is normalized parallel execution time. Three bars shown for each benchmark. The first bar is the base case using TEST&TEST&SET locks. The second bar is using SLE. The third bad is our proposal for TLR. For each bar, the upper portion is contribution due to lock-variable accesses (lock-portion), and the lower portion corresponds to the rest (non-lock-portion). The lock-portion also contains time spent waiting for a lock. The two rights bars are normalized with respect to the base case.

Benchmarks with frequent lock contention and frequent data conflicts are `radiosity`, `raytrace`, `barnes`, and `cholesky`. Only `radiosity` and `cholesky` benefit a little from SLE because SLE can sometimes remove locking overhead for some critical sections but requires lock acquisitions for the remaining critical sections. Thus a substantial portion of lock memory operation contributions remains. `Barnes` and `raytrace` do not benefit from SLE because of high contention for data as well as the lock.

TLR removes all locking overhead even in the presence of data conflicts, provides a lock-free execution, and performs better than both SLE and the base case for these benchmarks. By removing lock requests under contention, network contention and traffic is dramatically reduced. Further, since the data is directly being requested and the request is automatically queued for an appropriate time, the control transfer of a critical section among two different processors is on the order of a data transfer latency and the requesting processor gets the latest correct data. Thus the request/response sequence of a lock acquire is eliminated and the request for the data itself forms as an ordering mechanism. As we can see, for these benchmarks, nearly all lock overhead is removed (the small portion left is the contribution due to the execution of lock instructions by the processor itself because they nevertheless consume some processor execution bandwidth and only the actual writes to lock variables are removed). The non-lock-portion gets larger at times because there is inherent communication for the shared data itself even though the locking overhead has been removed. This communication accounts for some of the stalls in the processor core.

7 Related work

Lock-free and non-blocking methods. Lamport introduced lock-free synchronization to allow multiple threads to work on a data structure concurrently without a lock [20]. Herlihy gave a theoretical framework for constructing wait-free objects [11, 10]. Software only lock-free schemes have been shown to perform poorly as compared to lock-based schemes because of excessive data copying to allow roll-back [1, 5]. Hybrid hardware/software schemes have been proposed. The LOAD-LINKED/STORE-CONDITIONAL

(LL/SC) instructions allow for an optimistic atomic read-modify-write on a single word [14]. Transactional memory [12] and the Oklahoma update [37] were generalization of the LL/SC primitives outlined above. Both schemes required special instructions, hardware support, and coherence protocol extensions to provide mechanisms to write transactional code. Transactional memory is strictly not non-blocking and relied on software back off to guarantee forward progress. Oklahoma update did not provide starvation freedom although it did provide liveness by relying on a two-phase commit process and sorting memory addresses in hardware to order their request. Software transactional memory [34] uses software primitives to implement transactions but performs poorly with respect to its lock-based counterparts. Software-only proposals suffer from difficulty of use and a lack of generality and often poor performance. Speculative Lock Elision [30] dynamically elides lock acquire and release operations from an execution stream but requires lock acquisitions in the presence of conflicts. Extensive work has been done in improving performance of software non-blocking schemes [25, 3, 39, 28]. Software proposals have been made to make lock-based critical sections non-blocking [38] and thread scheduling that is aware of blocking locks [17, 26].

Database concurrency control and deadlocks. Transactions are well understood and well studied in database literature [9]. The use of timestamps for resolving conflicts and ordering transactions in database systems has been well studied [4, 33]. Holt [13] provides a good framework for reasoning about deadlocks in computer systems. Extensive work has been done in optimistic concurrency control (OCC) for database systems [19]. OCC was proposed as an alternative to locking in database management systems. In spite of extensive research, OCC techniques have not been popular because of key limitations [27]. Our proposal is not intended to replace database transactions because of different characteristics and requirements. The requirements placed on critical sections are far less strict than those on database transactions.

Speculative execution and parallelization. Speculative execution for aggressive implementation of memory consistency models was proposed by Gharachorloo et al. [6] and later extended [32, 7]. Similarly, work has been done in speculative parallelization of programs [16, 36]. While the buffering and speculative execution mechanisms they use are similar to ours, these proposals do not provide lock-free execution of lock-based code and do not address critical section serialization and thus are orthogonal to our scheme.

Efficient synchronization and lock acquire latency tolerance. Efficient synchronization has been extensively studied in literature. These techniques attempt to optimize the lock and data transfer operations [8, 2, 24, 15, 31]. Other work attempts to overlap latency of lock acquisitions with other computation if possible [29, 23]. The techniques are *not* lock-free and are orthogonal to our proposal but form part of the literature in dealing with the inefficiencies of lock-based synchronization and lock acquisition latencies. All these techniques suffer from the inherent locking overhead and serialization due to lock acquisitions. Our proposal is aimed at completely removing any serialization on the lock by not requiring any lock acquisitions, removing any locking overhead, and addressing the key limitations of locks.

8 Concluding remarks and future work

We have proposed Transactional Lock Removal (TLR), a hardware mechanism to convert lock-based critical sections transparently and optimistically into lock-free optimistic transactions and a timestamp-based fair conflict resolution scheme to provide transactional semantics and starvation-freedom, if the data accessed by the transaction can be locally cached and subject to some implementation specific constraints (Section 2.2). TLR provides both serializability and failure atomicity.

We summarize the contributions of our mechanism under 3 categories:

- **Programmability**. Reasoning about granularity of locks is not required because serialization decisions are made at run time based on actual data conflicts and independent of lock granularity. Thus, a critical problem in reasoning about writing multithreaded programs is solved. Cache lines are the coherence unit and thus represent the finest granularity possible and we provide this granularity without programmer involvement.
- **Fault tolerance**. Since the software wait on locks is done away with among conflicting threads, properties of lock-free and wait-free execution are achieved transparently. This translates to improved system wide interactions, no convoying or priority-inversion dangers, and robust execution in the presence of failing threads.
- **Performance**. Independent of lock granularity, because serialization decisions are made only in data conflict conditions, the performance of the finest granularity locking is automatically obtained. Further, since a queue of requestors is constructed in the hardware by using the coherence protocol, the data transfers are efficient and low overhead. Programmers can focus on writing correct code while hardware automatically extracts performance.

TLR is the first to combine these properties and provide a robust solution to the synchronization problem. While TLR does trade-off hardware for these properties, we believe the hardware cost is modest and it not complex. Additionally, we address the inherent limitations of the locking construct automatically while maintaining the well understood critical section abstraction for the programmer. Subject to resource constraints, our scheme is the first to provide a *wait-free* execution of a lock-based program transparently.

Although our proposal is a hardware-only scheme, we believe software developers can use such functionality in several ways. The size of transactions can be architecturally specified thus guaranteeing programmers a lock-free atomic execution of a sequence of memory operations. Such functionality helps programmers write simpler high-performance wait-free algorithms. Some of the hardware support, required for example in identifying critical sections, can be reduced by using appropriate compiler support. Future work involves exploring how software wait-free algorithms can benefit from our proposal and achieve high performance. Further, operating systems can exploit the notion of transactional execution to provide strong guarantees and appropriate operating systems involvement can prevent software failures (that affect one thread) to interact negatively with other concurrent threads and allow other threads to continue execution.

We view our proposal as a step in the direction towards high-performance and highly reliable concurrent multithreaded execution. Enabling reliable, high performance, and easy programming is an important problem and much work needs to be done in that direction.

References

[1] J. Allemany and E.W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 125–134, August 1992.

[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] G. Barnes. Method for Implementing Lock-Free Shared Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993.

[4] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[5] B.N. Bershad. Practical Considerations for Lock-Free Concurrent Objects. Technical Report CMU-CS-91-183, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1991.

[6] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, August 1991.

[7] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[8] J.R. Goodman, M.K. Vernon, and P.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.

[9] J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science, 60, 1978*, 1978.

[10] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[11] M. Herlihy. A methodology for implementing highly concurrent data objects. In *ACM Transactions on Programming Languages and Systems 15, 5, 745–770.*, 1993.

[12] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[13] R.C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–195, Dec 1972.

[14] 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.

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

[16] T. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[17] L.I. Kontothanassis, R.W. Wisniewski, and M.L. Scott. Scheduler-Conscious Synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, 1997.

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

[19] H.T. Kung and J.T. Robinson. On Optimistic methods of Concurrency Control. In *ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, pages 213-226.*, 1981.

[20] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11), 1977.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, 1978.

[22] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[23] J. Martinez and J. Torrelas. Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors. In *Workshop on Memory Performance Issues*, June 2001.

[24] 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.

[25] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[26] M.M. Michael and M.L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[27] C. Mohan. Less Optimism About Optimistic Concurrency Control. In *Proceedings of the Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 199–204, February 1992.

[28] M. Moir. Laziness Pays! Using Lazy Synchronization Mechanisms to Improve Non-Blocking Construction. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, July 2000.

[29] V.S. Pai, P. Ranganathan, S.V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.

[30] R. Rajwar and J.R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[31] R. Rajwar, A. Kägi, and J.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.

[32] P. Ranganathan, V.S. Pai, and S.V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[33] D.J. Rosenkratz, R.E. Stearns, and P.M. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[34] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[35] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, and E. Hagersten. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of the Symposium on High Performance Interconnects IV*, pages 41–52, August 1996.

[36] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[37] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel & Distributed Technology*, 1(4):58–71, November 1993.

[38] J. Turek, D. Shasha, and S. Prakash. Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, August 1992.

[39] J.D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.

The appendix demonstrates the role of marker messages in avoiding deadlocks when request deferral is allowed in a cache coherence protocol.

Appendix: Role of marker messages

We give an example to show the role of marker messages in a general protocol and how deadlock is avoided. The discussion here simply demonstrates how deadlock can be avoided appropriately using marker messages. In the example, we show how deadlock can happen in a base protocol if requests can be deferred. We then show how marker messages can be used to prevent a deadlock from occurring.

Consider the left half of Figure 9. It shows 4 processors, P0, P1, P2, and P3. For simplicity, assume the timestamp values among these processors is as follows: P0 < P1 < P2 < P3. In other words, P0 has the highest priority. For brevity, assume all requests are requests for exclusive ownerships. The arcs are labeled as "Event number/Message/Block". For example, "1/`rd_x`/A" from P3 to P0 means, at time t_1 a request for A is sent from P3 to P0. Further assume processor P0 has block A in stable exclusive owned state (shown as a solid box) and P1 has block B in exclusive owned state. The sequence on the left is as follows. At t_1, P3 sends a request for A to P0. P0 buffers the request and does not respond because P0 has higher priority. At t_2, P2 sends a request for another block B to P1. P1 buffers the request and does not respond because P1 has higher priority. Note, P1 is unaware of P0's request and vice-versa. Now, at t_3, P1 sends a request to P3 for block A because P3 is the current owner as per the cache coherence protocol. P3 does not have data because the block is pending and its own request has been buffered by P0. At t_4, P0 sends a request to P2 for block B because P2 is the latest owner of B. Again, P2 does not have the block because

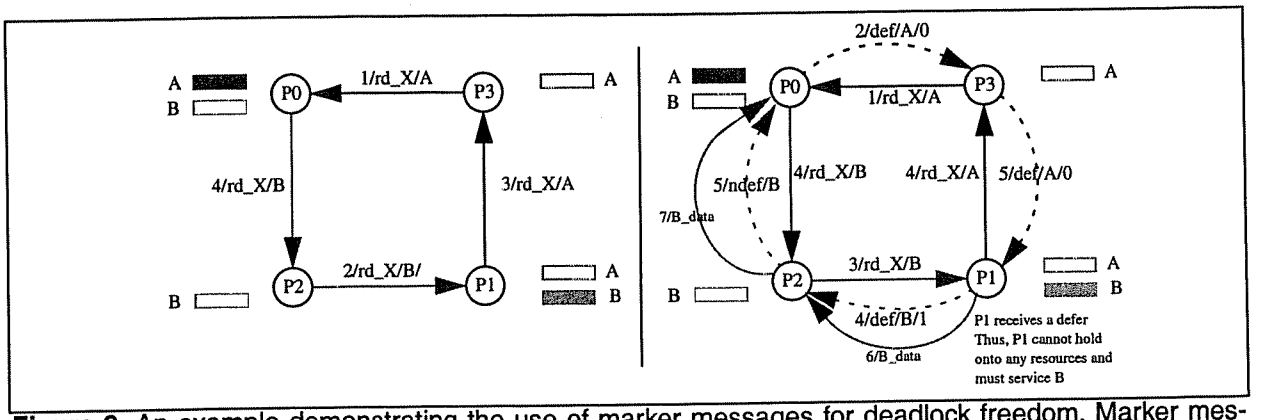

Figure 9. An example demonstrating the use of marker messages for deadlock freedom. Marker messages are shown as dotted lines and the base protocol messages are shown as solid lines.

P1 buffered P2's request. As can be seen, we have a cycle. P0 is waiting for block B from P2, P2 is waiting for block B from P1, P1 is waiting for block A from P3, and P3 is waiting for block A from P0 and this results in deadlock. This happened because although a pending chain is formed for each block (for A, P0→P3→P2, and for B P1→P2→P0), P1 is unaware of P0's request and P0 is unaware of P1's request.

Marker messages are used to resolve this. Consider the right half of Figure 9. The marker message is shown as a dotted arc and is labeled as follows: "Event number/Message/Block/Timestamp". The sequence is the same as above except now when P0 defers P3's request of t_1, P0 sends a marker message to P3 along with its own timestamp 0. Thus P3 knows the timestamp of the processor that deferred it. Similarly, P1 sends a marker message to P2 with its own timestamp 1. When P2 receives P0's request for block B, P2 cannot defer the request because P0's timestamp < P1's timestamp. Recall that P1 sent its timestamp to P2 in the marker message. P2 sends a message to P0 informing it that P0 is not deferred for this block. Therefore, P0 knows it has the lowest timestamp in the chain P1→P2→P0. Now when P1 sends a request for block A to P3 at t_4, P3 compares P1's timestamp with the timestamp of the processor that deferred P3, i.e., P0. Since P1 < P0, P3 sends a message to P1 informing it of the *transitive* deferral (intended to force a node to give up *retained* exclusive ownership to other blocks) along with P0's timestamp. When P1 receives a defer message, it knows it is not the lowest timestamp in the chain for A: P0→P3→P1. Thus, P1 must give up its exclusive ownership on block B. P1 applies P2's request by sending a data message back to P2. P2 then services P0's request and P0 receives data for block B. Thus, a cycle was prevented from being constructed. We believe the above is a general case and other cases can be reduced to this case. While a formal proof of deadlock-freedom is out of the scope of the paper, we hope the sketch above should provide intuition as to why the scheme is deadlock free.