Appears in the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct. 6-Oct. 9, 2002, San Jose, California.

Transactional Lock-Free Execution of Lock-Based Programs

Ravi Rajwar and James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706 USA
{rajwar, goodman}@cs.wisc.edu

Abstract

This paper is motivated by the difficulty in writing correct high-performance programs. Writing shared-memory multithreaded programs imposes a complex trade-off between programming ease and performance, largely due to subtleties in coordinating access to shared data. To ensure correctness programmers often rely on conservative locking at the expense of performance. The resulting serialization of threads is a performance bottleneck. Locks also interact poorly with thread scheduling and faults, resulting in poor system performance.

We seek to improve multithreaded programming trade-offs by providing architectural support for optimistic lock-free execution. In a lock-free execution, shared objects are never locked when accessed by various threads. We propose *Transactional Lock Removal* (TLR) and show how a program that uses lock-based synchronization can be executed by the hardware in a lock-free manner, even in the presence of conflicts, without programmer support or software changes. TLR uses timestamps for conflict resolution, modest hardware, and features already present in many modern computer systems.

TLR's benefits include improved programmability, stability, and performance. Programmers can obtain benefits of lock-free data structures, such as non-blocking behavior and wait-freedom, while using lock-protected critical sections for writing programs.

1 Introduction

Programming complexity is a significant problem in writing shared-memory multithreaded applications. Although threads simplify the conceptual design of programs, care and expertise are required to ensure correct interaction among threads. Errors in reasoning about appropriate synchronization among threads while accessing shared data objects result in incorrect program execution, and may be extremely subtle.

Transactions serve as an intuitive model for coordinating access to shared data. A *transaction* [7] comprises a series of read and write operations that provide the following properties: failure-atomicity, consistency, and durability. *Failure-atomicity* states a transaction must either execute to completion, or in the presence of failures, must appear not to have executed at all. *Consistency* requires the transaction to follow a protocol that provides threads with a consistent view of the data object. Serializability is an intuitive and popular consistency criterion for transactions. *Serializability* requires the result of executions of concurrent transactions to be *as if* there were some global order in which these transactions had executed serially [7]. *Durability* states that once a transaction is committed, it cannot be undone.

ASPLOS X 10/02 San Jose, CA, USA

While the concept of transactions is simple and convenient for programmers to reason with [10], processors today provide only restricted support for such transactions in their instruction sets. Examples are the atomic read-modify-write operations on a single word. The restricted size for these operations and limitations placed on their use render them ineffective in providing functionality of general transactions.

A lack of general transaction support in processors has led to programmers often relying on critical sections to achieve some of the functionality of transactions. *Critical sections* are software constructs that enforce mutually exclusive access among threads to shared objects and thus trivially satisfy serializability. Failure-atomicity is difficult to achieve with critical sections because it requires support for logging modifications performed in the critical section and then making these modifications visible instantaneously using an atomic operation. Critical sections therefore do not provide failure-atomicity. Critical sections are most commonly implemented using a software construct known as a *lock*. A lock is associated with a shared object and determines whether the shared object is currently available. Nearly all architectures support instructions for implementing lock operations. Locks have become the synchronization mechanism of choice for programmers and are extensively used in various software such as operating systems, database servers, and web servers.

Motivation. Two key limitations of lock-based critical sections motivate the work in this paper: 1) Complex trade-off between programmability and performance, and 2) Problems of stability (i.e., behavior in the presence of unexpected conditions) of the application.

Performance/programmability limitation of locks. The complex trade-off between programmability and performance exists because programmers have to reason about data sharing during code development using static information rather than dynamic run-time information. Programmers often use conservative synchronization to easily write correct code. While such use may guarantee correctness, provides stable software, and leads to faster code development, it also inhibits parallelism because threads are unnecessarily serialized. Fine-grain locks (e.g., one lock per node in a data structure) may help performance but make code difficult to write and error prone. Coarse-grain locks (e.g., one lock per data structure) help in writing correct code and reducing errors but hurt performance. Additionally, locks can contribute to significant overhead, serialize execution, and degrade overall system performance [16]. Exploiting dynamic concurrency is also often a non-trivial task [13].

Stability limitation of locks. If some thread owns a lock, and has marked it *held*, other threads requiring that lock have to wait for the lock to become *free*. Such wait can negatively impact system behavior. If the lock owner is de-scheduled by the operating system, other threads waiting for the lock cannot proceed because the lock is not free. In high concurrency environments, all threads may wait until the de-scheduled thread runs again. *Non-blocking* primitives guarantee some thread will execute an operation in a finite number of steps despite individual halting failures or delays [11]. Further, if the lock owner aborts, other threads waiting for the lock never complete as the lock is never free again.The shared structures updated by the aborted thread are left in an inconsistent state as critical sections lack failure-atomicity. A *wait-free* primitive guarantees any process completes any operation in a finite number of steps [11]. Wait-freedom adds starvation freedom to the non-blocking property. Conventional locks are neither non-blocking nor wait-free.

In spite of the limitations of locks, a lack of competitive alternatives and the intuitive appeal of critical sections has led to a nearly universal use of lock-based critical sections for synchronizing thread accesses. Even in environments where transactions are supported in software, they are implemented using lock-based critical sections. The limitations outlined above are becoming important and solutions must be found to enable programmers to exploit hardware thread parallelism efficiently and easily. A desirable approach therefore is to provide transparent *transactional execution* behavior for critical sections while maintaining the familiar paradigm of critical sections. Doing so enables the powerful concept of a transaction to be transparently reflected in common multithreaded programs and allows programmers the continued use of the lock-based critical section paradigm.

To address the problems outlined above, this paper proposes *Transactional Lock Removal (TLR)*. TLR uses modest hardware to convert lock-based critical sections transparently and dynamically into *lock-free optimistic transactions* and uses *timestamp-based fair conflict resolution* to provide transactional semantics and starvation freedom.

TLR uses Speculative Lock Elision (SLE) [30] as an enabling mechanism. SLE 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 critical sections. A *data conflict* occurs if, of all threads accessing a given memory location simultaneously, at least one thread is writing to the location. While SLE provided concurrent completion for critical sections accessing disjoint data sets, data conflicts result in threads restarting and acquiring the lock serially. Thus, when data conflicts occur, SLE suffers from the key problems of locks due to lock acquisitions.

TLR elides locks using SLE to construct an optimistic transaction but in addition also uses a timestamp-based conflict resolution scheme to provide lock-free execution even in the presence of data conflicts. A single, globally unique, timestamp is assigned to all memory requests generated for data within the optimistic lock-free critical section. Existing cache coherence protocols are used to detect data conflicts. On a conflict, some threads may restart (employing hardware misspeculation recovery mechanisms) 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 only after a successful execution. Doing so guarantees each thread will eventually win any conflict by virtue of having the earliest timestamp in the system and thus will succeed in executing its optimistic lock-free critical section. If the speculative data can be locally buffered, all non-conflicting transactions proceed and complete concurrently without serialization or dependence on the lock. Transactions experiencing data conflicts are ordered without interfering with non-conflicting transactions and without lock acquisitions.

Paper contribution. TLR is the first hardware technique for transparent lock-free execution of lock-based programs while providing transactional behavior (serializability and failure-atomicity) and starvation freedom. TLR has three primary benefits.

1. **TLR improves programmability**. TLR, rather than change the programming model to obtain transactional semantics, changes the hardware implementation to transparently provide such semantics. By allowing programmers to continue using the familiar lock-protected critical section interface, programmers do not have to learn new ways to write programs. TLR does not require software support and existing legacy code using critical sections can directly benefit from TLR.
2. **TLR improves performance**. TLR extracts and exploits fine-grain parallelism inherent in the program independent of the locking granularity employed by the programmer. Serialization of data accesses occurs based on data conflicts and only when such serialization is necessary for correctness.
3. **TLR improves stability**. TLR does not suffer from limitations of locks because it uses timestamps to obtain a lock-free execution in the presence of conflicts. As locks are not acquired, the software wait for a lock variable is eliminated and a non-blocking execution is achieved along with failure-atomicity. Further, the conflict resolution scheme guarantees all threads eventually succeed in a finite number of steps, thus providing a *wait-free* behavior subject only to resource constraints.

Section 2 presents the TLR algorithm and Section 3 discusses its implementation. Stability aspects of TLR are discussed in Section 4 and we evaluate TLR's performance in Section 5 and Section 6. We show that, for test&test&set locks, hardware with TLR outperforms hardware without TLR and, on the average performs better than MCS locks for fine-grain locks for our applications. We also show that using coarse-grain locks with TLR can outperform fine-grain locks because of improved memory system behavior. Finally we discuss related work (Section 7) and conclude (Section 8).

2 Transactional Lock Removal

TLR aims to achieve a *serializable* schedule of critical sections where all memory operations within a critical section are atomically inserted into some global order. This is illustrated in Figure 1. In this paper, the term transaction is used to mean a lock-free critical section satisfying the serializability condition.

Serializability requires the result of executions of concurrent transactions to be as if these transactions executed in *some* serial order. In the absence of data conflicts, serializability can be ensured using a technique such as SLE but the presence of data conflicts among concurrently executing threads requires additional mechanisms provided by TLR.

The basic idea behind TLR is as follows: a) Treat locks as defining scope of a transaction, b) Speculatively execute the transaction without requesting or acquiring the lock, c) Use a conflict resolution scheme to order conflicting transactions, and d) Use a technique to give the appearance of an atomic commit of the transaction, such as is provided by SLE [30].

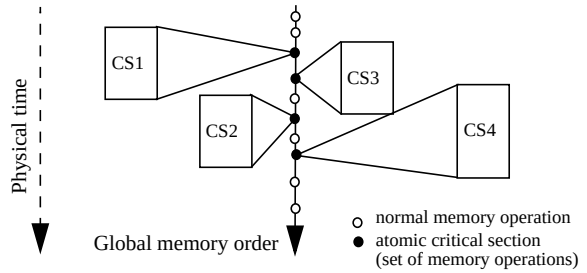

Figure 1. *While critical section executions (without lock acquires) overlap in physical time (with or without data conflicts), each critical section logically appears to be inserted atomically and instantly in a global ordering.*

TLR performs *active* concurrency control to ensure correct coordinated access to the data experiencing conflicting access by using the data itself rather than locks. Unlike TLR, SLE only identifies situations where lock-based concurrency control is not necessary—namely the absence of data conflicts among threads—and relies on the default lock-based concurrency control mechanisms if data conflicts occur.

We discuss achieving serializability in the presence of data conflicts in Section 2.1. In that section, we also discuss the use of timestamps as a conflict resolution mechanism. We then present the TLR algorithm for ensuring a serializable execution in Section 2.2. Section 2.3 gives an example of the TLR algorithm.

2.1 Problem: Achieving serializability

An execution of an optimistic lock-free transaction can be made serializable if the data speculatively modified by any transaction are not exposed until after the transaction commits and no other transaction writes to speculatively read data. A serializable execution can be achieved trivially by acquiring exclusive ownership of all required resources. If the thread executing the transaction does so for all required resources, the thread can operate upon the resources and then commit the updates atomically and instantly, thus achieving serializability.

In cache-coherent shared-memory multiprocessors the above requires: 1) Acquiring all required cache blocks (that are accessed within the transaction) in an appropriate ownership state, 2) Retaining such ownership until the end of the transaction, 3) Executing the sequence of instructions forming the transaction, 4) Speculatively operating upon the cache blocks if necessary, and 5) Making all updates visible atomically to other threads at the end of the transaction. However, as we shall see next, the presence of conflicts may prevent resources from being retained thus preventing successful execution of the lock-free transaction.

2.1.1. Necessity for conflict resolution

Livelock can occur if processors executing critical sections speculatively and in a lock-free manner repeatedly experience conflicts (as with SLE, the lock can always be acquired and forward progress is guaranteed but we require a solution that does not rely on lock acquisitions). Consider Figure 2 with two processors, P1 and P2. Assume both P1 and P2 have elided the lock (using SLE) and are in optimistic lock-free transaction execution mode. Both processors are accessing (and writing) shared memory locations *A* and *B* in the critical sections. The two processors write the two locations in reverse order of each other—P1 writes *A* first and then *B* while P2 writes *B* first and then *A*. Messages and state transitions for the corresponding blocks are shown. Time instances are labeled t_i where *i* denotes progressing

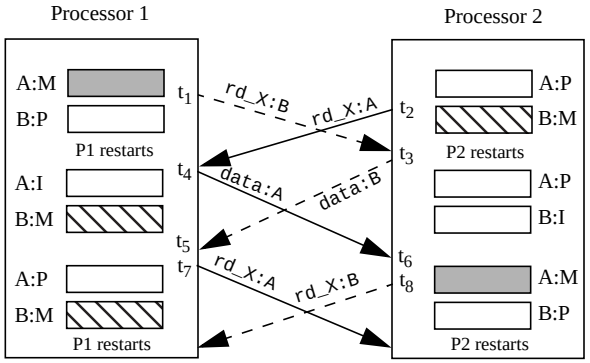

Figure 2. *In this example, both processors repeatedly restart. A and B are memory locations. Coherence states: modified (M), pending (P), invalid (I). Time progresses down.*

instances. Time progresses down. P1 has speculatively accessed block *A* and cached it in exclusive state (M). P2 also has speculatively accessed block *B* and cached it in the M state.

At t_1, P1 issues a request for exclusive ownership (`rd_X`) for block *B* (P1's write to *B*) and at t_2, P2 issues an `rd_X` block A (P2's write to *A*). The respective cache blocks transition into a transient (pending) state. At t_4, P1 receives P2's `rd_X` request for block *A*. P1 detects the request as a data conflict (block *A*, speculatively written to by P1, is accessed by another thread before P1 has completed its lock-free critical section). P1 triggers a misspeculation and restarts its lock-free critical section. Similarly, P2 receives P1's `rd_X` for *B* at t_3 and P2 restarts execution. Both P1 and P2 respond with the valid non-speculative data. The above sequence may occur indefinitely with no processor making forward progress because each processor repeatedly restarts the other processor.

Livelock occurs because neither processor obtains ownership of *both* cache blocks *simultaneously*. To ensure livelock freedom, among competing processors one processor must win the conflict and retain ownership. TLR assigns priorities to the lock-free transactions and employs the following key idea:

> Transactions with higher priority never wait for transactions with lower priority. In the event of a conflict, the lower priority transaction is restarted or forced to wait.

Consider two transactions T_1 and T_2 executing speculatively. Suppose T_2 issues a request that causes a data conflict with a request previously made by T_1, and T_1 receives T_2's conflicting request. The conflict is resolved as follows: if T_2's priority is lesser than T_1's priority, then T_2 waits for T_1 to complete (T_1 wins the conflict), else T_1 is restarted (T_2 wins the conflict). The "wait" mechanism may either involve an explicit negative acknowledgement or a delayed processing of the request. There are conceptual similarities to the *wound-wait* proposal by Rosenkrantz et al. [32] for distributed concurrency control [29].

For starvation freedom the resolution mechanism must guarantee all contenders eventually succeed and become winners. We use timestamps for conflict resolution and we discuss them next.

2.1.2. Timestamps for fair conflict resolution

We use timestamps for resolving conflicts to decide a conflict winner—earlier timestamp implies higher priority. Thus, the contender with the earlier timestamp wins the conflict.

The timestamps we use have two components: a local logical clock and processor ID. The logical clock is a way of assigning a

number to an event and the number is thought of as the time at which the event occurred. An event in our case is a successful execution of a TLR instance. The local logical clock value is increased by 1 or higher on a successful TLR execution and captures time in units of successful TLR executions on a given processor. Since these logical clocks are local, the logical clocks on different processors may have the same value. Such ties are broken by using the processor ID. Thus the timestamp comprising of the local logical clock and the processor ID are globally unique.

All requests generated from within a given transaction on a processor are assigned the same timestamp—namely the value of the timestamp at the start of the transaction. On a successful TLR execution, the processor increments its local logical clock to a value higher than the previous value (typically by 1) or to a value higher than the highest of all incoming conflicting requests received from other processors, whichever is larger. Doing so keeps the local logical clocks on the various processors loosely synchronized whenever a conflict is detected.

Our use of timestamps is similar to that proposed by Lamport [22]. Lamport used timestamps derived from logical clocks to implement distributed mutual exclusion with a starvation freedom guarantee. However, we only require timestamps for conflict resolution while Lamport used timestamps for *explicitly* ordering the execution of mutual exclusion regions among different processors. Thus 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. Since TLR does not require synchronized clocks, real-time systems clock could also be used.

Starvation freedom is achieved by retaining and reusing timestamps in the event of a misspeculation and restart under TLR. By reusing timestamps, processors retain their position. By updating timestamps as above, a processor will eventually have the earliest timestamp in the system and thus will eventually win all conflicts. TLR uses timestamps solely for the purpose of comparing priorities of two conflicting threads to determine which has a higher priority. Thus timestamp roll-over due to fixed size timestamps is easily handled without loss of TLR properties [29].

2.2 Solution: TLR algorithm

We assume a processor with support for SLE. All operations executed by a processor while performing TLR (i.e., the processor is considered to be in TLR mode) are part of the optimistic transaction and are speculative. Conventional cache coherence protocols are used to allow processors to retain ownership of cache blocks. In an invalidation-based cache coherence protocol, a processor with an exclusively-owned cache block receives and must respond to subsequent requests for the block. The processor controls the block and can appropriately respond. Figure 3 shows the TLR algorithm. In the discussion below, we use the term deferred to imply the processor retains ownership.

The first step is calculating the globally unique local timestamp as discussed in Section 2.1.2.

The second step is identifying start of a transaction. We use SLE to identify the start and end of transactions. SLE does so by exploiting *silent store-pairs*: a pair of store operations where the second store undoes the effects of the first store and the intervening operations appear to execute atomically [30][1]. SLE thus avoids writing (and even requesting exclusive permissions for) the lock variable. The first store of the pair corresponds to the start of the transaction and the second store of the pair corresponds to the transaction end. Once the start is identified, the lock is elided thus leaving the lock *free*. The processor register state is saved for recovery in the event of a misspeculation.

1. Calculate local timestamp
2. Identify transaction start:
   a) Initiate TLR mode (use SLE to elide locks)
   b) Execute transaction speculatively.
3. During transactional speculative execution
   - Locally buffer speculative updates
   - Append timestamp to all outgoing requests
   - If incoming request conflicts with retainable block and has later timestamp, retain ownership and force requestor to wait
   - If incoming request conflicts with retainable block and has earlier timestamp, service request and restart from step 2b if necessary. Give up any retained ownerships.
   - If insufficient resources, acquire lock.
     - No buffer space
     - Operation cannot be undone (e.g., I/O)
4. Identify transaction end:
   a) If all blocks available in local cache in appropriate coherence state, atomically commit memory updates from local buffer into cache (write to cache using SLE).
   b) Commit transaction register (processor) state.
   c) Service waiters if any
   d) Update local timestamp

Figure 3. *TLR algorithm. A mechanism for retaining ownership of cache blocks is assumed to be present. A retainable cache block is defined as a block in an exclusively owned coherence state. Requests are forwarded to the cache with the writable copy of the block. Data conflict is defined in Section 1.*

The third step comprises actions that may occur concurrently during speculative execution. A cache miss generated for data within the speculative execution carries with it the processors timestamp. Requests from other processors that result in a data conflict (for data accessed within the transaction) are checked for priority. If the incoming request has a later timestamp than the local processor, the incoming request's response is deferred. If the incoming request has an earlier timestamp, the local processor loses the conflict. It must service earlier deferred requests in-order and then service the conflicting incoming request. Doing so maintains coherence ordering in the protocol for that block. The execution may restart but the local clock is not updated.

During speculative execution, if any resource constraints, or operations that cannot be undone, are encountered, TLR cannot be applied. The processor requests the lock by exposing the elided writes and exits speculative mode. Since the lock is kept in shared state under TLR, any write to the lock triggers invalidations thus automatically informing other participating processors of the violation of the silent store-pair elision under TLR. During speculative execution, data modified is buffered in the write

1. SLE identifies regions for atomic execution without any semantic information from the software and is a purely hardware technique that only observes the dynamic instruction stream. The notion of silent store-pairs employed by SLE for doing so is an example of the notion of Temporal Silence investigated by Lepak and Lipasti [23].

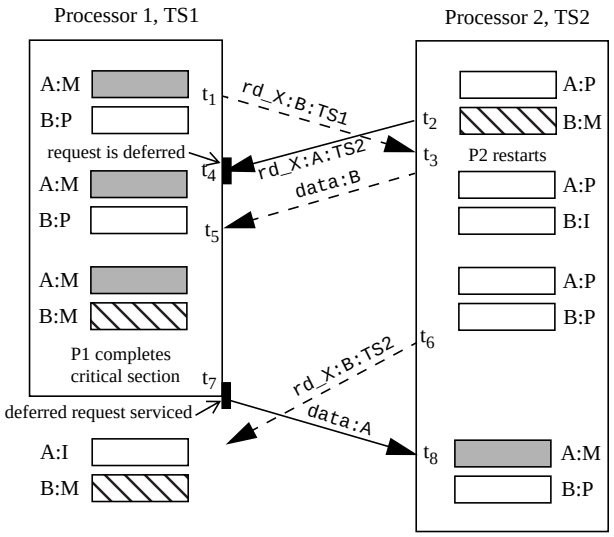

Figure 4. *A conflict resolution scheme is employed allowing processor 1 to retain exclusive ownership of both cache blocks A and B. TS1 and TS2 are timestamps and TS1 < TS2. By deferring a response, conflicts are masked and a serializable execution is achieved. Coherence states: modified (M), pending (P), invalid (I). Time progresses down.*

buffer and exclusive requests for the cache block are issued to the memory system.

Finally, when a transaction end is identified, the transaction is committed. If all appropriate blocks have been brought into the cache in appropriate state (exclusive or shared), then the buffered data in the write-buffer is *atomically* committed into the cache—all required lines are already in writable state in the cache. If not, then speculative execution can proceed until the blocks corresponding to the write-buffer are available in appropriate state. After the speculative data has been committed into the cache, deferred requests from before are then serviced in order. The local logical clock update is performed as per the rules of Section 2.1.2.

Up to now, we have focused on interaction among timestamped requests—requests that are part of critical sections. However, in some programs, the data protected by locks may be accessed from outside a critical section and hence without locks, and may conflict with timestamped requests. While this is a data race, it may be acceptable for the program. Such situations may be correctly handled in various ways. One approach is to trigger a misspeculation when an un-timestamped request is received. Thus, if any thread performs a conflicting access from outside a critical section, then TLR cannot be applied because a data race exists. Another approach is to treat un-timestamped requests as deferrable. Such a request is assumed to have the latest timestamp in the system (and thus the lowest priority) and the un-timestamped request is atomically ordered after the current critical section. Since a data response is not sent until after the critical section, the requestor cannot consume the data and hence is ordered with the correct value.

2.3 TLR algorithm example

We revisit the example of Figure 2 using the algorithm outlined in Figure 3. Consider Figure 4. Two processors, P1 and P2, execute a lock-free critical section and both write shared memory locations *A* and *B* in the critical section. Both the processors have a unique timestamp—TS1 for P1 and TS2 for P2 where TS1 < TS2 (processor P1 has higher priority than processor P2 and wins all conflicts). Assume that now both processors have the additional ability to buffer and delay responding to incoming requests. As in the earlier example, the two processors write the two locations, *A* and *B*, in reverse order of each other. Both P1 and P2 have elided the lock and are executing in TLR mode.

At t_1, P1 issues a `rd_X` for block *B* (P1's write to *B*) and at t_2, P2 issues a `rd_X` for block *A* (P2's write to *A*). The respective cache blocks transition into a transient (pending P) state. All memory operations within the transaction are assigned the same timestamp. Therefore P1's `rd_x` for *B* has TS1 appended and P2's `rd_X` for *A* has TS2 appended. At t_3, P2 receives P1's request and compares the incoming request timestamp TS1 with its local timestamp TS2. Since the incoming request has an earlier timestamp than P2, P2 services the request and responds with the data for block *B* (non-speculative value). On applying the incoming request, a data conflict is triggered at P2 and P2 restarts execution of its transaction. At t_4, P1 receives P2's `rd_X` request for block *A*. Since TS1 < TS2, P1 wins the conflict and defers the request by buffering it. The cache block state for *A* remains M. At t_5 P1 receives data for block *B* from P2. P1 has acquired and retained permissions on *both* cache blocks *A* and *B* and can successfully execute and atomically commit the transaction. At t_7, P1 completes its transaction, architecturally commits its speculative state and services P2's deferred request. P1 responds with the latest architecturally correct data. Meanwhile, P2 has restarted and is re-executing its transaction. The key difference between Figure 2 and Figure 4 is P1's ability to retain exclusive permissions in the latter example.

3 A TLR implementation

We discuss how TLR is implemented. The algorithm outlined earlier in Figure 3 relies on the ability of a processor to retain ownership of a cache block. Two policies to retain exclusive ownership of cache blocks are NACK-based and deferral-based. With NACK-based techniques, a processor refuses to process an incoming request (and thus retains ownership) by sending a negative acknowledgement (NACK) to the requestor. Doing so forces the requestor to retry at a future time. With deferral-based techniques, a processor defers processing an incoming request by buffering the request and masking any conflict. NACK-based and deferral-based techniques are contrasted elsewhere [29].

In this paper, we use a deferral-based scheme because it does not require coherence protocol support (such as NACKs). With deferrals, the conflict-winning processor with an exclusively owned cache block delays processing the incoming request for a bounded time (preferably until the processor has completed its transaction) and thus defers the request. The coherence transitions (and state transitions as seen by the "outside world") are assumed to have occurred but the processor does not locally apply the incoming request. Request deferral and delayed responses works in split-coherence-transaction systems where the address request processing is split into two sub-coherence-transactions—request and response. The response (often the data) may appear an arbitrary time later and any number of other requests and responses may occur between the two sub-coherence-transactions.

We now discuss a deferral-based implementation of the algorithm. Figure 5 shows a shared-memory multiprocessor where every processor has a local cache hierarchy and they are connected together via an interconnection network. We make no assumptions regarding the memory consistency model or coherence protocol. The protocol may be broadcast snooping or directory-based and interconnect may be ordered or un-ordered. The

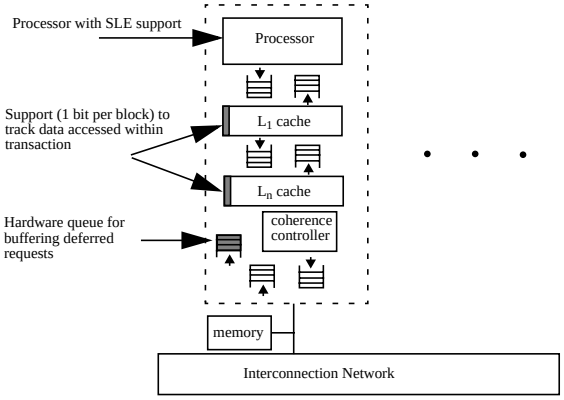

Figure 5. *A TLR implementation. Additional hardware support is shown shaded. Processor supports SLE.*

processor is assumed to have SLE capability: support for predicting regions as transaction, local speculative buffering, mechanism to track data accessed within transactions (an access bit per cache block tracks data accessed during the transaction), and ability to detect data conflicts [30, 29].

TLR support is required at the coherence controller where decisions for deferrals are made. We do not require changes to the coherence protocol state transitions. The TLR concurrency control algorithm runs in parallel and along with the coherence protocol and only performs deadlock-free concurrency control.

Misses generated within a transaction carry a timestamp. An additional deferred coherence input queue is present to buffer 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* message 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.

In Section 3.1, we discuss coherence protocol interactions with TLR. We base our discussion around a modern broadcast snooping protocol, the Sun Gigaplane [35]. This choice does not take away from the generality of our discussion. Timestamp-based conflict resolution is necessary only if deadlock dangers exist. Section 3.2 discusses a situation where timestamp-based conflict resolution can be relaxed if deadlock is guaranteed to not occur. Section 3.3 discusses the resource constraints for TLR.

3.1 An implementation of the deferral algorithm

Up to now, we have assumed the requests get forwarded to the appropriate cache that exclusively owns the cache block and has the data. We now briefly outline the problem of deadlock that may occur due to interactions between the TLR concurrency control algorithm and a general coherence protocol. This deadlock is not in the TLR algorithm but may result because of how the underlying coherence protocol may be implemented. In Section 3.1.1. we discuss how such a deadlock can be prevented without coherence protocol changes. We then discuss TLR interaction with cache blocks in the shared state in Section 3.1.2.

3.1.1. Propagating priority information

On a cache miss, the cache block performs a transition from invalid to a pending state and it stays in a pending state between

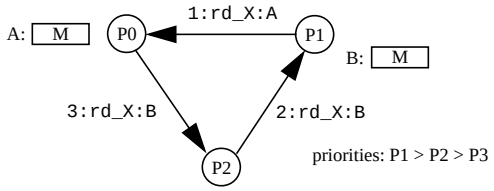

Figure 6. *Unlike the earlier example with 2 processors, the presence of an additional processor complicates issues because now all requests are distributed in the system and all processors are not guaranteed to observe all other requests*

the request initiation and completion. At some time between the two phases, the request gets ordered by the coherence protocol and the cache may become the owner of the cache block according to the coherence protocol, even though data is unavailable. This request-response decoupling introduces a complication because even though a processor may lose a conflict under TLR, it does not have data to provide. Consider Figure 6 where three processors P0, P1, and P2 are shown executing transactions. The arc labelling "`1:rd_X:A`" means a read for exclusive ownership (`rd_X`) request for block *A* was issued at time t_1. Assume priority ordering is: P0 > P1 > P2 where P0 has the highest priority. P0 has cache block *A* in exclusive owned (M) state and P1 has cache block *B* in M state.

At time t_1, P1 issues a `rd_X` request for *A*. Since P0 owns the block *A*, P1's request is forwarded to P0. P0 receives P1's request, wins the conflict, buffers P1's `rd_X` request for *A*, and defers a response. Now P1 exclusively owns block *A* because P1's request has been ordered by the protocol but the data (and hence the write permissions to the block) are with P0. *P1 is waiting for P0 for cache block A.* At time t_2, P2 issues a `rd_X` request for *B*. P1 owns the cache block and thus P2's request is forwarded to P1. P1 receives P2's request, wins the conflict, buffers P2's `rd_X` for *B* request and delays a response. Now P2 exclusively owns the block *B* because P2's request was ordered by the protocol but the write permissions to the block are still with P1. *P2 is waiting for P1 for cache block B.* At time t_3, P0 issues a `rd_X` request for *B*. P2 owns the cache block (even though the data is still with P1) and thus P0's request is forwarded to P2. P2 compares its local priority with P0's incoming message and loses the conflict. P2 must service P0's request by responding with data. However, P2 cannot do so because P2 is waiting for P1 to release cache block *B*. P1 will not release the cache block *B* because P1 won the conflict but P1 is itself waiting for P0 for cache block *A*.

P2 is waiting for P1 (for cache block *A*) which is waiting for P0 (for cache block *B*) which is waiting for P2 (for cache block B). If such a wait is uncontrolled, deadlock occurs. The waiting processors are unaware of other waiting processors and inadvertently form a cyclic wait.

The key idea for implementing a deferral-based concurrency control mechanism is to propagate information about processor priorities along the coherence protocol chains to prevent cyclic waits. On a miss, a processor allocates a pending buffer, a miss status handling register (MSHR) and tracks the request. If the processor receives a request (an intervention) from another processor for the outstanding block, an intervention buffer or the MSHR tracks the incoming request. When the processor receives data for the block, the processor operates upon the data and sends it to the requestor based on the information stored in the local MSHR. In Figure 6, for the chain for block *A*, P0 is aware of P1 but P1 is not aware of P0. Similarly, for block *B*, P1 is aware of P2 but not vice versa and P2 is aware of P0 but not vice versa. P0 can send information to P1 (regarding deadlock-free concurrency

control) but P1 cannot send information to P0 because P1 is unaware of P0.

P0 must inform P1 that P0 has higher priority and must not be forced to wait for block *B*. The presence of P2 in the chain prevents P1 from observing P0's request. Mechanisms can be added to propagate such information along the chain. The conflicting requests must propagate along the coherence chain towards the root (i.e., the stable block) to "restart" lower priority requests. We use special messages, we call *marker messages*, for doing so.

Marker messages are directed messages sent in response to a request for a block under conflict for which data is not provided immediately. The delay may be because either the processor is forcing the request to wait or the processor does not have the data for the block in question but is considered to be the owner of the block. The idea behind marker messages is to make processors aware of their immediate neighbors in a chain. These messages have no coherence interactions. The marker messages are *only* required when the processor is doing TLR and receives a conflicting request for a exclusively owned block. We have a mechanism to propagate timestamps requests upstream (*probes*) to the cache that has the block with valid data. Probes are only used to propagate a conflict request upstream in a cache coherence protocol chain. Thus, when P2 receives P0's request for *B*, P2 forwards the probe (with P0's timestamp) to P1 since P2 received a marker message from P1. P1 receives P0's forwarded probe (via P2) and loses the conflict because P0 has higher priority than P1. P1 releases ownership of block *B* and the cyclic wait is broken.

3.1.2. Handling the protocol shared state

Often, within a critical section, a processor may read a shared location, operate upon the value and write a new value to the same location. The read operation brings the corresponding cache block locally into shared state and the subsequent write results in an upgrade operation where the processor requests exclusive ownership of the cache block so that the processor can update the block. External invalidation requests to shared blocks typically cannot be deferred because no processor exclusively owns the block (upgrades in some protocols may not expect an acknowledgement). These requests must be serviced without delay and may trigger a misspeculation (violation in atomicity of the transaction). To reduce the probability of such upgrade-induced misspeculation, we employ instruction-based prediction to reduce the necessity of requiring upgrades following misspeculation.

The basic idea behind the predictor is as follows. Load operations within a critical section are recorded and any store operations within the critical section to the same address results in the predictor update occurring corresponding to the appropriate load operation. For out-of-order processors, the predictor update must occur at instruction commit because only then does the processor know for certain if the memory operation occurred within the transaction (out-of-order processors issue memory operations without regard to program order but instruction retirement is in program order). The predictor is indexed by instruction address. Predictors for optimizing patterns as above have been proposed earlier [17]. We show in our results section that the use of the simple read-modify-write predictor substantially improves performance of the base system without TLR as well as with TLR.

Cache blocks that are only read within critical sections are brought into the cache in a shared state. If repeated upgrade-induced violations occur, the processor can issue exclusive requests for the blocks, obtain the blocks in owned state and defer external requests to such blocks. Doing so guarantees a successful TLR execution even without the above optimization.

3.2 Selectively relaxing timestamp order

Deadlock is not possible if only one cache block is under conflict within the transaction because a cyclic wait is impossible (the head node of the coherence chain is always a stable state and does not wait for anyone else). Timestamps serve two functions: providing starvation freedom and deadlock-freedom. In protocols such as the Sun Gigaplane (which are non-nacking protocols), a queue of requests is automatically formed for a given block if multiple processors issue ownership requests while the block states are pending and the deferred queue is serviced in a serial order. In such situations, strict timestamp order can be relaxed. Thus, a timestamp-induced restart can be temporarily avoided if only a single cache block is contended for. However, if an additional cache block is accessed that may deadlock (i.e., generates a cache miss), then the timestamp order must be enforced.

3.3 Resource constraints

TLR has resource limitations similar to SLE. If the cache is used to track the lock and data accesses for a critical section, the finite size of the cache restricts the data set size that can be tracked speculatively. The associativity of the cache also places a limit because conflict misses force evictions of cache blocks. Well known and well understood techniques, such as victim caches [15], for handling such situations exist. Victim caches are small, fast, fully associative structures that buffer cache lines evicted from the main cache due to conflict and capacity misses. The victim cache can be extended with a speculative access bit per entry to achieve the same functionality as a regular cache.

Since the write buffer buffers speculative memory updates, its size restricts the number of static block addresses that can be written to within a critical section. Since writes are merged in the write buffer and memory locations can be re-written within the write buffer (because atomicity is guaranteed), the number of unique cache lines written to within the critical section determines the size of the write buffer.

In addition, for the implementation we provide, TLR also requires sufficient buffering for deferred requests. The size of buffering can be calculated *a priori* and is a function of the system size and victim cache size. In any case, TLR like SLE can guarantee correctness under all circumstances and in the presence of unexpected conditions can always acquire the lock. Another resource constraint is the operating systems scheduling quantum—it must be possible to execute the critical section within a single quantum.

4 TLR Stability Properties

We have discussed TLR's performance and programmability aspects. We now discuss the implications of TLR on stability of multithreaded programs. In the TLR algorithm described in Section 2.2, three key invariants must hold: a) the timestamp is retained and re-used following a conflict-induced misspeculation, b) timestamps are updated in a strictly monotonic order following a successful TLR execution, and c) the earlier timestamp request never loses a resource conflict and thus succeeds in obtaining ownership of the resource. If TLR is applied, these invariants collectively provide two guarantees:

1. A processor eventually has the earliest timestamp in the system, and
2. A processor with the earliest timestamp eventually has a successful lock-free transactional execution.

The two properties above result in the following observation: "In a finite number of steps, a node will eventually have the earli-

est 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."

However, the guarantees are true only if TLR can be applied. In the presence of constraints, such as resource limitations and un-cacheable requests, these guarantees cannot be provided. These limitations make the guarantee of stability properties conditional. Such a guarantee can be constructed using the size of the victim cache and the scheduling quanta. Some of these parameters can be architecturally specified. For example, if the system has a 16 entry victim cache and a 4-way data cache, the programmer can be sure any transaction accessing 20 cache lines or less is ensured a lock-free execution. A programmer expecting guaranteed behavior will need to be aware of precise specifications. For a critical section to be executed in a wait-free manner, the lock must be positively identified. TLR uses SLE, which must be implemented to identify all locks that satisfy a certain idiom. The spin-wait loop of the lock acquire will only be reached if TLR has failed thus giving the programmer a method of detecting when wait-freedom has not been achieved. This is an area of future work.

Multiple nested locks can also be elided if hardware for tracking these elisions is sufficient. If some inner lock cannot be elided due to an inability to track multiple elisions, the inner lock is treated as data. This does not change TLR's properties: the execution is still lock-free and lower priority threads will be deferred by higher priority threads temporarily. The outermost lock controls whether TLR's properties are met [29].

TLR provides support for restartable critical sections because failure atomicity is provided by TLR. Sometimes operating system may want to restart certain threads—e.g., if threads are deadlocked. Locks makes such termination difficult because the thread might be in a critical section and may have modified shared memory. TLR provides hardware support for buffering speculative updates within critical sections and exposes these values only at the time the critical section execution is committed. Thus, if a thread is terminated during TLR execution, the speculative updates are discarded. Restartable critical section are a useful functionality for operating systems to exploit.

Restartable critical sections allow the underlying blocking synchronization primitive to be made non-blocking. Non-blocking synchronization primitives allow a system as a whole makes progress despite individual halting failures or delays. TLR makes the critical section execution non-blocking because TLR provides a lock-free execution. If a process is de-scheduled, a misspeculation is triggered and the lock is left free with all speculative updates within the critical section discarded. Other threads scheduled continue to operate on the protected data. The wait-free behavior follows from the non-blocking behavior discussed above but subject to a stronger guarantee of starvation freedom.

5 Evaluation Methodology

We evaluate TLR using microbenchmarks and applications. We evaluate four configurations—1) BASE: base system, 2) BASE+SLE: base system with SLE optimization [30], 3) BASE+SLE+TLR: base system with SLE and TLR optimizations (this paper), and 4) MCS: system with MCS locks [26]. MCS locks are scalable software-queue locks that perform well under contention. For convenience we will refer to these four schemes in text as BASE, SLE, TLR, and MCS respectively. BASE, SLE, and TLR use the same benchmark executable employing the test&test&set lock.

5.1 Microbenchmarks

The three microbenchmarks capture three different locking and critical section data conflict behaviors—coarse-grain/no-conflicts, fine-grain/high-conflicts, and fine-grain/dynamic conflicts.

Coarse-grain/no-conflicts. The `multiple-counter` microbenchmark consists of n counters protected by a single lock. Each processor uniquely updates only one of n counters $2^{24}/n$ times. While a single lock protects the counters, there is no dependence across the various critical sections for the data itself and hence no conflicts.

Fine-grain/high-conflicts. The `single-counter` microbenchmark corresponds to critical sections operating on a single cache line. One counter is protected by a lock and n processors increment the counter $2^{16}/n$ times. No inherent exploitable parallelism exists as all processors operate upon the same data (and cache line).

Fine-grain/dynamic-conflicts. The `doubly-linked list` microbenchmark consists of a doubly-linked list with `Head` and `Tail` pointers protected by one lock. Each processor dequeues an item by removing the item pointed to by `Head`, and then enqueues it by adding it to `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 and dequeue operations have completed. A non-empty queue can support concurrent enqueue and dequeue operations. When the queue is non-empty, each transaction modifies `Head` or `Tail`, but not both, so enqueuers can execute without interference from dequeuers, and vice versa. Transactions must modify both pointers for an empty queue. This concurrency is difficult to exploit 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 [13, 33]. The critical sections are non-trivial involving pointer manipulations and multiple cache line accesses.

Processors execute critical sections in a loop for a fixed number of iterations. Special care was taken in designing these microbenchmarks. We use a methodology similar to that used by Kumar et al. [19]. To ensure fairness, we introduce delays after a lock release operations. After releasing the lock, the processor waits a minimum random interval before proceeding to ensure another processor has an opportunity to acquire the lock before a successive local lock re-acquire, thus reducing unfairness.

5.2 Applications

We use `barnes`, `cholesky`, and `mp3d` from `SPLASH` [34] and `radiosity`, `water-nsq`, `ocean-cont`, and `ray-trace` from `SPLASH2` [39]. A locking version of `mp3d` is used to study the impact of TLR on a lock-intensive benchmark [16]. This version of `mp3d` does frequent synchronization to largely uncontended locks and lock access latencies cannot be hidden by a large reorder buffer. Table 1 presents details. These applications have been selected for their fine-grain locking and critical section behavior. `Barnes`, `cholesky`, `radiosity`, `raytrace`, and `ocean-cont` have lock contention. `Water-nsq` and `mp3d` do not have lock contention but perform frequent synchronization. These benchmarks are optimized for sharing, employ fine-grain locks, and have little communication in most cases. Where appropriate, the data structures are padded to eliminate false sharing.

5.3 System configuration

The target system configuration is shown in Table 2. It is a chip-multiprocessor configuration with snooping L1 caches inter-

Application	Type of Simulation	Inputs	Type of Critical Sections
Barnes	N-Body	4K bodies	tree node locks
Cholesky	Matrix factoring	tk15.O	task queue & col. locks
Mp3D	Rarefied field flow	24000 mols, 25 iter.	cell locks
Radiosity	3-D rendering	-room, batch mode	task queue & buffer locks
Water-nsq	Water molecules	512 mols, 3 iter.	global structure locks
Ocean-cont	Hydrodynamics	128x128, 2 days	counter locks
Raytrace	Image rendering	car	work list & counter locks

Table 1. Benchmarks

connected together. All coherence traffic occurs between the L1s. The L2 cache is shared. The 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. The PC-indexed predictor for optimizing read-modify-write sequences is used for all experiments (BASE, SLE, TLR, and MCS).

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 accurately, 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. Care must be taken in evaluating multithreaded workloads because of non-deterministic executions. Random latency perturbations are introduced in the simulator (similar to [1]).

6 Results

In Section 6.1 we provide an intuition behind why TLR may improve performance. We present microbenchmark results in Section 6.2 and application results in Section 6.3.

6.1 Performance intuition

With TLR, processors request data without acquiring locks and the data request is appropriately queued using the coherence protocol and a timestamp-based conflict resolution scheme. Figure 7 shows four processors P0, P1, P2, and P3 requesting the same cache line *A* thus exhibiting true data conflict. For simplicity, assume the conflict resolution scheme orders priorities as follows: P0, P1, P2, and P3 (P0 has highest priority). P0 is currently executing its optimistic lock-free transaction and has accessed cache line *A* (in modified state M). P0 receives, defers (and buffers) P1's request for *A*. P2's request is buffered by P1 and P2's request is buffered by P3. P0 operates on *A*, complete its critical

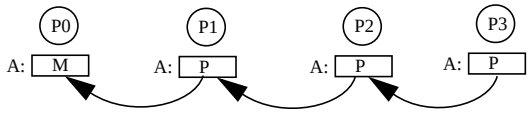

Figure 7. *Queue maintenance and data transfer.*

section and then responds to P1's request with the latest data for *A*. Subsequently, P1 operates upon the data, execute its own transaction, and on completion, respond to P2's request with the latest data for *A*, and so on. Thus, while processors execute transactions conflicting on data accessed, they are 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. No transaction requires to restart in the above example. Further, while P0 is operating on *A*, other processors wait for the latest copy rather than introduce contention in the system by repeatedly requesting locks and data. The behavior is similar to hardware queue locks [9] but now the queue is constructed using the data itself and no lock requests are generated. TLR, by removing explicit lock requests and locking overhead under contention, reduces network contention and latency.

If the order of timestamps is different from the order in which the respective requests are ordered by the coherence protocol, additional latency may be introduced due to misspeculation—a processor may have to restart and service a higher priority request making the performance sub-optimal. Processors in TLR mode restart only if the order of requests processed by the coherence protocol are different from the timestamp order. Such situations can be addressed if coherence protocol support was added. In the TLR algorithm in this paper, no coherence protocol state transition changes are made or special protocol support is required. As we see later TLR always outperforms the base system.

6.2 Microbenchmarks

Figure 8, Figure 9, and Figure 10 present microbenchmark results. The y-axis shows wall-clock time for completing the parallel execution of the microbenchmarks. The x-axis shows processor counts. Each data point in the graphs represents the same amount of work. 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 8 shows results for `multiple-counters`. The base scheme degrades performance as more threads run concurrently because of severe contention for the lock. MCS, as expected is scalable under high contention but experiences a fixed software overhead. TLR and SLE behave identically because of the absence of any data conflicts and both outperform BASE and

Processor	1 GHz (1 ns clock), 128 entry reorder buffer, 64 entry load/store queue, 16 entry instruction fetch queue, 3-cycle branch mispredict redirection penalty, out-of-order issue/execution and commit of 8 inst. per cycle, issue loads as early as possible, 8-K entry combining predictor, 8-K entry 4-way BTB, 64 entry return address stack. Pipelined functional units, 8 alus, 2 multipliers, 4 floating point units, 3 memory ports. Write-buffer: 64-entry (64 byte wide). 128-entry PC indexed predictor for collapsing read-modify-write sequences within critical sections into single request. Synchronization primitive: load-linked/store-conditional. Memory Model: Total Store Ordering (aggressive implementation [8]).
L1 caches	Instruction cache: 64-KByte, 2-way, 1 cycle access, 16 pending misses. Data cache: 128-KByte. 4-way associative, write-back, 1-cycle access, 16 pending misses. Line size of 64 bytes.
TLR parameters	64 entry silent store-pair predictor table, support for upto 8 store-pair elisions at any time (~lock nesting depth of 8)
Coherence protocol	Sun Gigaplane-type MOESI protocol between L1s, split transaction. Address bus: broadcast network, snoop latency of 20 cycles, 120 outstanding transactions. L2 cache: 4MB 12 cycle access. Memory access: 70 cycles. Data network: point-to-point, pipelined, latency: 20 cycles.

Table 2. Simulated machine parameters.

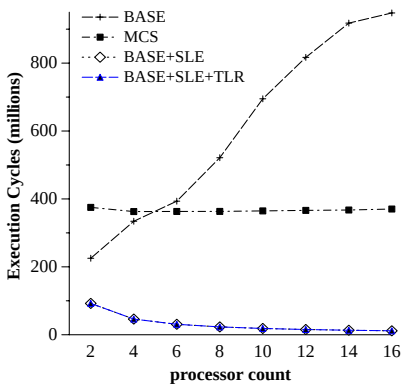

Figure 8. *Multiple counter results: coarse-grain/no-conflicts.*

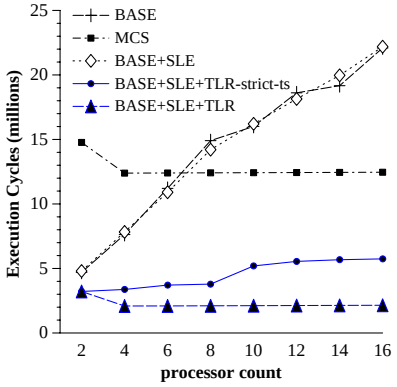

Figure 9. *Single counter results: fine-grain/high-conflict*

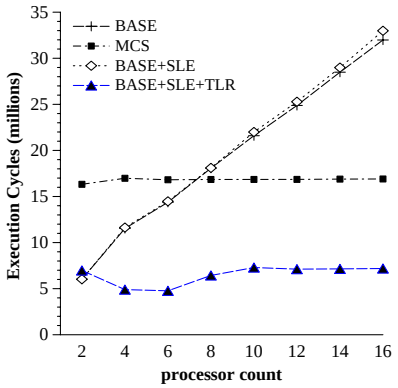

Figure 10. *Doubly-linked list results: fine-grain/dynamic-conflicts*

MCS. They experience no lock overhead and true concurrency is exploited. Perfect scalability is achieved.

Figure 9 shows results for `single-counter`. BASE performance degrades with increasing threads because of severe contention for the lock and data. SLE behaves similar to BASE because SLE detects frequent data conflicts, turns off speculation, and falls back to the BASE scheme. MCS is scalable but experiences a fixed software overhead. Following our performance intuition discussion in Section 6.1, we get ideal queued behavior for TLR and increasing concurrent threads does not degrade performance the way the other schemes do. No explicit lock requests are made under TLR and TLR performs ideally; no processor restarts and all transactions complete with a single cache miss. Also shown is the *TLR-strict-ts* case without the single cache block optimization of Section 3.2 Under TLR-strict-ts, timestamps are always enforced even though there is no danger of deadlock because only one data block is being contended for. The performance gap between TLR and TLR-strict-ts is because sometimes the order in which requests reach the coherence protocol is different from the order of the respective timestamps resulting in some misspeculation. The mismatch of protocol orders and timestamp orders results in a sub-optimal ordering and additional latencies (Section 6.1).

Figure 10 shows results for `doubly-linked list`. Performance for BASE degrades similar to the other microbenchmarks because of severe lock contention. SLE does not perform well either (and performs similar to BASE) 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 data conflicts. Any concurrency SLE exploits is offset by locking overhead when SLE needs to acquire the lock. MCS again is scalable but experiences a fixed software overhead. TLR performs well and can exploit enqueue/dequeue concurrency. For two processors, BASE performs slightly better than TLR because of fairness issues for that one run.

In summary, TLR outperforms both BASE and MCS. TLR exploits dynamic concurrency while both BASE and MCS are limited by synchronization performance. MCS performs a constant factor worse than TLR while BASE performance degrades quite substantially with increasing contention. Poor behavior of BASE under lock contention occurs because of repeated access to the lock variable by multiple processors racing for the lock and data thus introducing a large amount of traffic into the network [16]. MCS is scalable because processors form an orderly queue in software rather than repeatedly race for the variable and data.

6.3 Applications

Figure 11 shows application performance for 16 processors. The y-axis is normalized execution time. All bars are normalized to BASE. Each benchmark has three bars: the first bar is BASE. The second bar is SLE and the third bar is TLR. Each bar is divided into two parts: contributions due to lock variable accesses (loads and stores) and the remaining contributions. The accounting is performed at instruction commit time—the instruction that stalls commit is charged the stall. The breakup is approximate since accounting for stall cycles due to individual operations is difficult and not accurate. For some benchmarks, the non-lock portion for the optimized case is larger than the non-lock portion for the base case. This is because sometimes removing locks puts other memory operations on the critical path. Speculative loads issued for data within critical sections that were earlier overlapped with the lock-acquire operation now are exposed and stall the processor. Since we assume fast network latencies and an aggressive memory system, communication among processors is fast and thus the stalls due to lock operations is small.

All experiments employ the instruction-based predictor for reducing latencies in critical sections and discussed earlier (Section 3.1.2. and Table 2). This results in a highly-optimized base system execution and the performance numbers for TLR are thus conservative. Later, we discuss the effect this predictor has on the base system and present performance numbers to give an idea of how much better TLR would do against a more conventional base case. The speedup for technique *X* over technique *Y* is the ratio of the benchmark parallel cycle count with technique *Y* to that of the benchmark parallel cycle count with technique *X*. A speedup value greater than 1 is better.

`Ocean-cont` and `water-nsq` do not show much performance benefits. While `ocean-cont` has lock contention and opportunities for concurrent critical section execution, the performance impact on our target system is not much because lock accesses do not contribute much to performance loss. `Water-nsq` has frequent uncontended lock acquires. While the bars for BASE show potential for performance, removing locks does not result in a corresponding performance gain because now the data cache misses within the critical section, that were earlier overlapped with lock access misses, now are exposed and account for the stalls. For, TLR speedup over BASE for `water-nsq` is 1.01 and for `ocean-cont` is 1.02. MCS speedup over BASE for `ocean-cont` is 0%, and for `water-nsq` is 0.96. The performance loss for MCS for `water-nsq` is due to the software overhead for uncontended locks.

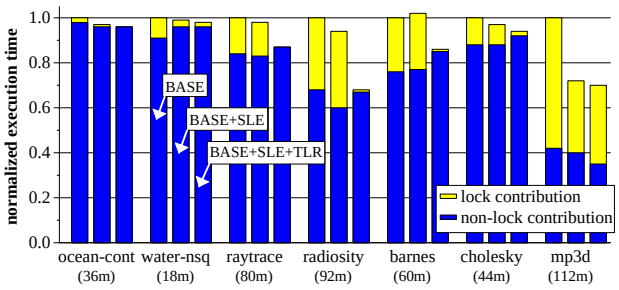

Figure 11. *Application performance for 16 processors. The y-axis is normalized execution time. All bars are normalized to the performance of BASE. Benchmarks are on the x-axis. Each benchmark has three bars: first bar is BASE, second bar is BASE+SLE and third bar is BASE+SLE+TLR. Each bar is divided into two parts: contributions due to lock variables (load and store instructions) and the remaining contributions. The number in parentheses below the benchmark name is the parallel execution cycle count, in millions, for the BASE shown as the first of three bars for each benchmark.*

For `radiosity`, speedup of TLR over BASE is 1.47 and nearly all locking overhead disappears. Speedup of MCS over BASE is 1.35. The task queue critical section was most contended for in `radiosity` and accounted for most conflict-induced restarts under TLR.

For `raytrace`, the speedup of TLR over BASE is 1.17. MCS performance is similar to TLR and its speedup over BASE is also 1.17. For `raytrace` (`car` input) on our system, lock contribution to execution time is 16%—much less than those reported earlier on systems with larger latencies, slower memory systems and different cache coherence protocols [19, 16].

For `barnes` TLR speedup over BASE is 1.16. However, MCS speedup over BASE is 1.21. MCS performs 4% better than TLR—the only application where MCS performs better than TLR. `Barnes` is based on a hierarchical octree representation of space in three dimensions and each node in the tree has its own lock. The root of this tree represents a space cell containing all bodies in the system. The tree is built by adding particles to the initially empty root cell and subdividing a cell into its eight children as soon as it contains more than single body. Most locking occurs in the tree building phase. Each process loads its bodies in the octree using locks to ensure atomic updates of the cell nodes. These locks tend to be contended and have data conflicts resulting in TLR restarting frequently. TLR's restarts are due to sub-optimal ordering discussed earlier in Section 6.1. MCS constructs an ordered software queue and thus performs better than TLR.

`Cholesky`, with the `tk15.0` input set, is the only benchmark that cannot fit one critical section's data within the local cache. About 3.7% of dynamic critical section executions resulted in resource limitations for local buffering (write-buffer limitations). This occurs at three functions (`ScatterUpdate`, `CompleteSuperNode`, and `ModifyColumn`) where a column in the matrix is locked and the algorithm then writes to the column entries resulting in buffer limitations (80% due to write buffer and 20% due to cache). TLR nevertheless achieves a speedup of 1.05 over BASE. MCS performs slightly worse than BASE (0.97).

`Mp3d` has frequent lock accesses but these locks are largely uncontended. The 128K data cache is unable to hold all locks and hence the processor suffers miss latency to locks. With TLR, significant lock contribution still remains. TLR achieves a speedup of 1.40 over BASE. BASE performs better than MCS (speedup over MCS: 1.47) because MCS pays a software overhead even for uncontended locks. This overhead adds up significantly if locking is frequent. TLR outperforms MCS by achieving a speedup of 2.06 because TLR pays no software overhead.

The performance gaps between MCS and TLR for `barnes` and the TLR restarts in the applications suggests more optimizations are possible for TLR where coherence protocol support can be used. A similar gap (between TLR and an ideal TLR execution) was also observed in Figure 9 in Section 6.2.

Coarse-grain vs. fine-grain experiment. With `mp3d`, a noticeable locking overhead remained and we investigated it further. We conjectured replacing the per-cell fine-grain locks in `mp3d` by one single coarse-grain lock should provide better performance because the data foot-print reduces and the memory system behavior should improve substantially. We replaced the individual cell locks in `mp3d` with a single lock. This is bad for BASE (and MCS) because now the benchmark has severe contention. As expected, TLR with one lock for all cells in `mp3d` outperforms BASE with fine-grain per-cell locks by 58% (speedup 2.40) and outperforms TLR with fine-grain per-cell locks by 41% (speedup 1.70). Thus, using coarse-grain locks can improve performance significantly over fine-grain locks.

Read-modify-write prediction effects. The performance we report for the BASE case uses the instruction-based predictor for collapsing read-modify-write sequences within predicted critical sections. We give speedups of BASE with the predictor (the results in Figure 11) with respect to BASE without the predictor (BASE-no-opt: a more conventional base case). The speedup is calculated as the ratio of the parallel cycle count for BASE and parallel cycle count for BASE-no-opt. A speedup value greater than 1 is better. The speedups are—`ocean-cont`: 1.00, `water-nsq`: 1.04, `raytrace`: 1.28, `radiosity`: 1.05, `barnes`: 1.04, `cholesky`: 1.33, and `mp3d`: 1.13. With the optimization, the time spent waiting for lock operations increases because critical section data latencies are reduced. Thus, our speedups in Figure 11 would be much larger if we assumed a more conventional base case without the predictor. For all benchmarks, a 128 entry PC-indexed predictor was sufficient (only `radiosity` used more than 30 entries—using just under 100) and most of the remaining benchmarks used less than 20 entries).

7 Related work

We discuss related work under three categories: lock-free and non-blocking methods, database concurrency control, and lock-based synchronization.

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 [21]. Herlihy gave a theoretical framework for constructing wait-free objects [12, 11]. Software lock-free schemes using lock-free data structures have been proposed to address the inherent limitations of locking [12, 38, 4, 27]. Lock-free schemes provide optimistic concurrency 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. Software only lock-free schemes have been shown to perform poorly as compared to lock-based schemes because of high software overheads and excessive data

copying to allow roll-back [2, 6]. With TLR, programmers continue using the familiar lock-based critical section while obtaining the benefits of lock-free data structures.

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 [13] and the Oklahoma update [36] were generalization of the LL/SC primitives outlined above. Both schemes required special instructions, programmer support, and coherence protocol extensions to provide mechanisms to write transactional code. Transactional memory is not strictly 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 requests. Software transactional memory [33] uses software primitives to implement transactions but performs poorly with respect to its lock-based counterparts. Speculative Lock Elision [30] dynamically elides lock acquire and release operations from an execution stream but requires lock acquisitions in the presence of conflicts.

Improving performance of software non-blocking schemes have been studied previously [27, 4, 38]. Software proposals have been made to make lock-based critical sections non-blocking [37] and thread scheduling that is aware of blocking locks [18, 28].

Database concurrency control. Transactions are well understood and studied in database literature [10]. The use of timestamps for resolving conflicts and ordering transactions in database systems has been well studied [5, 32]. Optimistic concurrency control (OCC) was proposed as an alternative to locking in database management systems [20]. OCC involves a read phase where objects are accessed (with possible updates to a private copy of these objects) followed by a serialized validation phase to check for data conflicts (read/write conflicts with other transactions). This is followed by the write phase if the validation is successful. TLR does not have a serialized validation phase and exploits hardware techniques to provide transactional behavior.

Lock-based synchronization. Lock-based synchronization has been extensively studied in literature. These techniques attempt to optimize the lock and data transfer operations [9, 3, 26, 16, 31]. The techniques are not lock-free. These techniques suffer from locking overhead and serialization due to lock acquisitions.

Martínez and Torrellas introduced *Speculative Locks*, allowing speculative threads to bypass a held lock and enter a critical section [24]. At any time the lock is always acquired by one thread which is non-speculative. Speculative threads could then become non-speculative after a lock was released by the non-speculative thread if no data conflicts were detected by the speculative threads and the speculative threads had completed their critical sections. In the presence of data conflicts, speculative threads always restart and retry the above sequence, competing for the lock. A free lock is always written to and acquired explicitly by a thread. In *Speculative Synchronization* [25], Speculative Locks is extended to include the SLE mechanism to be used in the absence of data conflicts. In the presence of data conflicts, rather than falling back on the underlying scheme as SLE does, it adapts by employing Speculative Locks as described above. These schemes provide the same forward progress guarantees as SLE. These schemes are not lock-free, experience the limitations of locks, and do not provide the guarantees provided by TLR.

Delaying responses to requests for lock variables for a short time and thus emulating hardware queued locks was proposed earlier [31]. TLR generalizes that notion by applying deferrals to data and to multiple cache blocks simultaneously.

8 Concluding Remarks

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

TLR is a step in the direction towards high-performance and highly reliable concurrent multithreaded execution. We summarize the contributions of our mechanism under 3 categories:

- **Programmability**. Reasoning about granularity of locks is not required because ordering decisions are dynamically made based on actual data conflicts and independent of lock granularity. Thus, a critical problem in reasoning about writing multithreaded programs is solved. Coarse granularity locking and frequent locking can be employed without paying a performance penalty.
- **Stability**. Since the software wait on locks is eliminated, properties of lock-free and wait-free execution are achieved transparently. This results in improved system wide interactions, non-blocking behavior, and improved stability.
- **Performance**. Since serialization decisions are made only when data conflicts occur, the performance of the finest granularity locking is automatically obtained independent of locking granularity. Since a queue of requestors is constructed in hardware using the coherence protocol, data transfers are efficient and low overhead. Programmers can focus on writing correct code while hardware automatically extracts performance.

TLR is the first proposal to combine these properties. While TLR does tradeoff hardware for these properties, the hardware cost is modest. Additionally, we address the inherent limitations of the locking construct automatically while maintaining the well understood critical section abstraction for the programmer.

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 wait-free critical section execution. Further, operating systems can exploit the notion of transactional execution to provide improved behavior 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.

Acknowledgements

We would like to especially thank Maurice Herlihy for extensive discussions regarding the ideas in the paper and comments on drafts of the paper. We thank Ras Bodik, Trey Cain, Mike Dahlin, Joel Emer, Brian Fields, Timothy Heil, Mark Hill, Milo Martin, Manoj Plakal, Eric Rotenberg, Shai Rubin, Dan Sorin, and David Wood for discussions and comments, and the reviewers for their comments. We thank Michael Scott for answering queries regarding MCS behavior, Alaa Alameldeen for discussions regarding multithreaded workload evaluation issues, and Alain Kägi and Sanjeev Kumar for the MCS lock code. This work is supported in part by National Science Foundation grant CCR-9810114.

References

[1] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, Feb. 2002.

[2] J. Allemany and E. 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, Aug. 1992.

[3] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.

[4] 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.

[5] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[6] 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, Sept. 1991.

[7] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 11:624–633, 1976.

[8] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, Aug. 1991.

[9] 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, Apr. 1989.

[10] J. Gray. The transaction concept: Virtues and limitations. In *Seventh International Conference on Very Large Data Bases*, pages 144–154, Sept. 1981.

[11] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

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

[13] 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.

[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, Nov. 1987.

[15] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[16] 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.

[17] S. Kaxiras and J. 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, Jan. 1999.

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

[19] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 23–34, May 1999.

[20] H. Kung and J. T. Robinson. On optimistic methods of concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[21] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[23] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[24] J. F. Martínez and J. Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues*, June 2001.

[25] J. F. Martínez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[26] 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, Feb. 1991.

[27] 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*, pages 267–275, May 1996.

[28] 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.

[29] R. Rajwar. *Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs*. PhD thesis, University of Wisconsin, Madison, WI, 2002.

[30] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, Dec. 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, Jan. 2000.

[32] D. J. Rosenkrantz, 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.

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

[34] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

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

[36] 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, Nov. 1993.

[37] 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*, pages 212–222, Aug. 1992.

[38] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rochester Institute of Technology, Rochester, NY, 1995.

[39] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. 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.