Appears in the proceedings of the 34th International Symposium on Microarchitecture (MICRO), Dec. 3-Dec. 5, 2001, Austin, Texas.

Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution

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

Abstract

Serialization of threads due to critical sections is a fundamental bottleneck to achieving high performance in multithreaded programs. Dynamically, such serialization may be unnecessary because these critical sections could have safely executed concurrently without locks. Current processors cannot fully exploit such parallelism because they do not have mechanisms to dynamically detect such false inter-thread dependences.

We propose Speculative Lock Elision (SLE)*, a novel micro-architectural technique to remove dynamically unnecessary lock-induced serialization and enable highly concurrent multithreaded execution. The key insight is that locks do not always have to be acquired for a correct execution. Synchronization instructions are predicted as being unnecessary and elided. This allows multiple threads to concurrently execute critical sections protected by the same lock. Misspeculation due to inter-thread data conflicts is detected using existing cache mechanisms and rollback is used for recovery. Successful speculative elision is validated and committed without acquiring the lock.*

SLE can be implemented entirely in microarchitecture without instruction set support and without system-level modifications, is transparent to programmers, and requires only trivial additional hardware support. SLE can provide programmers a fast path to writing correct high-performance multithreaded programs.

1 Introduction

Explicit hardware support for multithreaded software, either in the form of shared memory multiprocessors or hardware multithreaded architectures, is becoming increasingly common [9, 21, 38, 2]. As such support becomes available, application developers are expected to exploit these developments by employing multithreaded programming. While server workloads have traditionally displayed abundant thread-level parallelism, increasing evidence indicates desktop applications may also display such parallelism if programmers focus efforts on exploiting these emerging architectures.

In multithreaded programs, synchronization mechanisms—usually locks—are often used to guarantee threads have exclusive access to shared data for a critical section of code. A thread acquires the lock, executes its critical section, and releases the lock. All other threads wait for the lock until the first thread has completed its critical section, serializing access and thus making the entire critical section appear to execute atomically.

For a variety of reasons, concurrent accesses to a shared data structure by multiple threads within a critical section may in fact not conflict, and such accesses do not require serialization. Two such examples are shown in Figure 1. Figure 1a shows an example from a multithreaded application `ocean` [40]. Since a store instruction (line 3) to a shared object is present, the lock is required. However, most dynamic executions do not perform the store operation and thus do not require the lock. Additionally, multiple threads may update different fields of a shared object, while holding the shared object lock, and often these updates do not conflict. Such an example, involving updates of a hash table, is shown in Figure 1b. This example is similar to a thread-safe hash-table implementation from SHORE, a database object repository [5].

In these examples, conventional speculative execution in out-of-order processors cannot take advantage of the parallelism present because the threads will first wait for a free lock and then acquire the lock in a serialized manner.

a)

```
1.LOCK(locks->error_lock)
2.if (local_error > multi->err_multi)
3.      multi->err_multi = local_err;
4.UNLOCK(locks->error_lock)
```

b)

```
        Thread 1                              Thread 2
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(X)
if (!var)
    hash_tbl.add(X);
UNLOCK(hash_tbl.lock)
                                   LOCK(hash_tbl.lock)
                                   var = hash_tbl.lookup(Y)
                                   if (!var)
                                       hash_tbl->add(Y);
                                   UNLOCK(hash_tbl.lock)
```

Figure 1. ***Two examples of potential parallelism masked by dynamically unnecessary synchronization.***

No mechanisms currently exist to detect this parallelism. Frequent serialization hurts performance of even tuned multithreaded applications [37, 20], and the degradation can be much worse in the presence of conservative synchronization.

In developing threaded programs, programmers must make trade-offs between performance and code development time. Although multithreaded programming can improve throughput, a certain level of expertise is required to ensure correct interplay among program threads. Such required expertise is generally higher than for most single-threaded programs because sharing of data structures among threads is often subtle and complex. Programmers may avoid much of the complexity while ensuring correctness by using conservative techniques [17]. Doing so provides a faster and easier path to a correctly working program, but limits thread-level parallelism in programs because of unnecessary, synchronization-induced serialization constraints on thread execution: in the dynamic execution, no data hazards may have existed among the threads.

Ideally, programmers would be able to use frequent and conservative synchronization to write obviously correct multithreaded programs, and a tool would automatically remove all such conservative use. Thus, even though programmers use simple schemes to write correct code, synchronization would be performed only when necessary for correctness; and performance would not be degraded by the presence of dynamically unnecessary synchronization.

In this paper, we show how hardware techniques can be used to remove dynamically unnecessary serialization from an instruction stream and thereby increase concurrent execution. In *Speculative Lock Elision* (SLE), the hardware dynamically identifies synchronization operations, predicts them as being unnecessary, and elides them. By removing these operations, the program behaves as if synchronization were not present in the program. Of course, doing so can break the program in situations where synchronization is required for correctness. Such situations are detected using pre-existing cache coherence mechanisms and *without executing synchronization operations*. In this case, recovery is performed and the lock is explicitly acquired. Synchronization is performed only when the hardware determines that serialization is required for correctness.

Safe dynamic lock removal is performed by exploiting a property of locks and critical sections as they are commonly implemented. If memory operations between the lock acquire and release appear to occur atomically, the two writes corresponding to the lock acquire and release can be elided because the second write (of the lock release) undoes the changes of the first write (of the lock acquire). Section 3.3 discusses this concept in detail. Atomicity violations, discussed further in Section 5.3, can be detected using cache coherence protocols already implemented in most modern processors.

SLE has the following key features:

1. Enables highly concurrent multithreaded execution:

Multiple threads can concurrently execute critical sections guarded by the same lock. Additionally, correctness is determined without acquiring (or modifying) the lock.

2. Simplifies correct multithreaded code development:

Programmers can use conservative synchronization to write correct multithreaded programs without significant performance impact. If the synchronization is not required for correctness, the execution will behave as if the synchronization were not present.

3. Can be implemented easily:

SLE can be implemented entirely in the microarchitecture, without instruction set support and without system-level modifications (e.g., no coherence protocol changes are required) and is transparent to programmers. Existing synchronization instructions are identified dynamically. Programmers do not have to learn a new programming methodology and can continue to use well understood synchronization routines. The technique can be incorporated into modern processor designs, independent of the system and the cache coherence protocol.

To our knowledge, this is the first proposed technique for removing dynamically unnecessary and conservative synchronization operations from a dynamic execution *without* performing the lock-acquire and release operations, and *without* requiring exclusive ownership of the lock variable.

In Section 3 and Section 4 we discuss the idea of SLE and provide implementation strategies in Section 5. Much of the additional functionality required is either present in modern microarchitectures or involves well understood techniques developed for other microarchitectural optimizations.

2 Background

In this section, we provide a background into locking and performance/complexity trade-offs in multithreaded programming that can benefit from SLE.

2.1 Performance/complexity trade-offs in multithreaded programming

Conservative locking. In any multithreaded programming effort, programming complexity is the most significant problem to consider [7] and care is required to ensure correct synchronization and interaction among threads. Lack of appropriate synchronization can result in an incorrect program execution. To ensure correctness, program-

mers rely on conservative locking, often at the expense of performance.

Locking granularity. A careful program design must choose appropriate levels of locking to optimize the tradeoff between performance and ease of reasoning about program correctness. Early parallel programs were typically designed with few locks and programmers did not have to reason about correctness because all memory was protected by these locks. As parallelism increased and locking became frequent, performance degraded and finer grained locks were used. Finer granularity locking can improve performance but introduces programming complexity and makes program management difficult.

Thread-unsafe legacy libraries. An important source of serialization in some environments is the presence of nonreentrant legacy software binaries. If a thread calls a library not equipped to deal with threads, a global locking mechanism is used to prevent conflicts, thus serializing access and resulting in a performance degradation.

2.2 Mutual exclusion in microprocessors

Mutual exclusion is commonly implemented using atomic read-modify-write primitives such as SWAP, COMPARE&SWAP, LOAD-LOCKED/STORE-CONDITIONAL (LL/SC), and EXCHANGE. These instructions allow processors to—sometimes conditionally—atomically swap a register value and a memory location.

The simplest mutual exclusion construct is a TEST&SET lock. TEST&SET performs an atomic swap on a memory location. TEST&TEST&SET [32], an extension to TEST&SET, performs a read of the lock to test it, before attempting the TEST&SET operation. An example implementation of the TEST&TEST&SET sequence is shown in Figure 2.

While numerous lock constructs, both hardware and software, have been proposed, the simplicity and portability of TEST&TEST&SET locks make them quite popular. Hardware architecture manuals recommend [8, 10, 33, 18], and database vendors are advised [22] to use these simple locks as a portable locking mechanism (of course, a few other software primitives are also used when circumstances dictate their use). The POSIX threads standard recommends synchronization be implemented in library calls such as

```
TEST         { L1:1.ldl t0, 0(t1)    #t0 = lock
                  2.bne t0, L1:      #if not free, goto L1
TEST & SET   {    3.ldl_l t0, 0(t1)  #load locked, t0 = lock
                  4.bne t0, L1:      #if not free, goto L1
                  5.lda t0, 1(0)     #t0 = 1
                  6.stl_c t0, 0(t1)  #conditional store, lock = 1
                  7.beq t0, L1:      #if stl_c failed, goto L1,
               8-15.<critical section>
UNSET        {   16.stl 0 ,0(t1)     #lock = 0, release lock
```

Figure 2. *Typical code for lock-acquire and releases using the Alpha ISA [9].* *This is a TEST&TEST&SET lock construct. Inst. 1 through 7 form the lock-acquire, inst. 16 is the lock release.*

`pthread_mutex_lock()` and these calls implement the TEST&SET or TEST&TEST&SET locks.

3 Enabling concurrent critical sections

In this section, we discuss the concept of Speculative Lock Elision. We use the code sequence in Figure 2 and LL/SC for ease of explanation; the ideas are readily applied using other synchronization primitives. We first discuss lock-enforced false dependences. In Section 3.2 and Section 3.3 we show how these dependences can be overcome and we step through an example in Section 3.4.

3.1 How locks impose false dependences

A lock is a control variable determining whether a thread can execute a critical section—it enforces a control dependence among threads but does not produce any useful result. Additionally, the lock forms a data dependence within the single thread—the value of the lock determines the control flow for the thread. The lock-enforced control dependence is manifested as a data dependence because the lock is a memory location checked and operated upon by the threads. This dependence is analogous to data dependences where an instruction waits for a logically preceding instruction, on which it may depend, to complete.

3.2 How to remove false dependences due to locks

Atomicity means all changes performed within a critical section appear to be performed instantaneously. The *appearance* of instantaneous change is key. By acquiring a lock, a thread can prevent other threads from observing any memory updates being performed within the critical section. While this conventional approach trivially guarantees atomicity of all updates in the critical section, it is only one way to guarantee atomicity.

Locks can be elided and critical sections concurrently executed if atomicity can be guaranteed for all memory operations within the critical sections by other means. For guaranteeing atomicity, the following conditions must hold within a critical section:

1. Data read within a speculatively executing critical section is not modified by another thread before the speculative critical section completes.
2. Data written within a speculatively executing critical section is not accessed (read or written) by another thread before the speculative critical section completes.

A processor can provide the *appearance of atomicity* for memory operations within the critical section without acquiring the lock by ensuring that partial updates performed by a thread within a critical section are *not* observed by other threads. The entire critical section appears to have executed atomically, and program semantics are maintained.

The key observation is that a lock does not always have to be acquired for a correct execution if hardware can provide the appearance of atomicity for all memory operations within the critical section. If a data conflict occurs, i.e., two threads compete for the same data other than for reading, atomicity cannot be guaranteed and the lock needs to be acquired. Data conflicts among threads are detected using existing cache protocol implementations as demonstrated in Section 5. Any execution not meeting the above two conditions is not retired architecturally, thus guaranteeing correctness.

Algorithmically, the sequence is this:

1. When a lock-acquire operation is seen, the processor *predict*s that memory operations in the critical section will occur atomically and elides the lock acquire.
2. Execute critical section speculatively and buffer results.
3. If hardware cannot provide atomicity, trigger misspeculation, recover and explicitly acquire lock.
4. If the lock-release is encountered, then atomicity was not violated (else a misspeculation would have been triggered earlier). Elide lock-release operation, commit speculative state, and exit speculative critical section.

Eliding the lock acquire leaves the lock in a FREE state, allowing other threads to apply the same algorithm and also speculatively enter the critical section. Even though the lock was not modified, either at the time of the acquire or the release, critical section semantics are maintained.

In step 3, the processor can alternatively try to execute the algorithm again a finite number of times before explicitly acquiring the lock. We call this number the *restart threshold*. Forward progress is always guaranteed because after the restart threshold is reached, the lock is explicitly acquired.

The above algorithm requires processors to recognize lock-acquire and release operations. As discussed in Section 2.2, lock-acquires are implemented using low-level synchronization instructions. These instructions may not always be used only for implementing lock-acquires. Additionally, lock-releases are implemented using a normal store operation. Thus, the processor lacks *precise* information about an operation being a lock-acquire or release but only observes a series of loads, stores, and low-level synchronization primitives and can only *predict* them to be lock operations. In the next section we show why predicted lock-acquires and releases can *still* be removed without precise semantic information.

3.3 Removing lock-acquires and releases by eliding silent store-pairs

The lock-acquire and release contain store operations. If the lock is FREE, the lock-acquire *writes* to the lock marking it HELD. A lock-release *writes* to the lock marking it FREE. Figure 3 shows memory references under SLE in three columns. Instructions are numbered in program order. The first column shows the programmers view, the second column shows the operations performed by the processor, and the third column shows the value of location `_lock_` as seen by different threads.

Program Semantic	Instruction Stream	Value of _lock_ as seen by self	as seen by other threads
TEST _lock_	`L1:i1 ldl t0, 0(t1)`	FREE	FREE
	`i2 bne t0, L1:`		
TEST _lock_	`i3 ldl_l t0, 0(t1)`	FREE	FREE
&	`i4 bne t0, L1:`		
SET _lock_	`i5 lda t0, 1(0)`		
	`i6 stl_c t0, 0(t1)`	HELD	FREE
	`i7 beq t0, l1:`		
critical section	*i8-i15*		
RELEASE _lock_	`i16 stl 0, 0(t1)`	FREE	FREE

Figure 3. ***Silent store-pair elision.*** *Inst. i6 and i16 can be elided if i16 restores the value of _lock_ to its value prior to i6 (i.e., value returned by i3), and i8 through i15 appear to execute atomically w.r.t. other threads. Although the speculating thread elides i6, it still observes the HELD value itself (because of program order requirements within a single thread) but others observe a FREE value.*

If i3 returns FREE, i6 writes HELD to location `_lock_`. i16 releases the lock by marking it FREE. After the lock-release (i16), the value of `_lock_` is the same as it was at the start of the lock-acquire (i.e., before i6)—i16 restores the value of `_lock_` to its value prior to i6. We exploit this property of synchronization operations to elide lock acquires and releases. If critical section memory operations occur atomically, then stores i6 and i16 form a *silent pair*. The architectural changes performed by i6 are undone by i16. When executed as a pair, the stores are silent; individually, they are not. Location `_lock_` must not be modified by another thread, or i6 and i16 cannot form a silent pair. Note other threads can read memory location `_lock_`.

The above observation means the SLE algorithm need not depend on program semantic information, specifically whether an operation is a lock-acquire or lock-release. The lock elision can be done by simply observing load and store sequences and the values read and to be written. If any instruction sequence matches the pattern of columns 2 and 3 in Figure 3, the location `_lock_` is not modified by another thread, and the memory operations in the critical section appear to execute atomically, the two stores corresponding to i6 and i16 are elided. The location `_lock_` is never modified, and other threads can proceed without being serialized on the value of `_lock_`.

Thus, an additional prediction is added to the algorithm of Section 3.2. On a store predicted to be a lock-acquire, the processor predicts that the changes performed by the store will be undone shortly by another store, and no other thread will modify the location in question. If this is so, and since the entire sequence is globally observed to occur

atomically, the two stores can be elided. We use a filter to determine candidate load/store pairs. For example, in our implementation, only instructions `ldl_l` and `stl_c` (they normally occur as a pair) are considered. The `stl_c` store is elided speculatively, and a future store matching the pattern of Figure 3 is identified.

The complete algorithm for SLE is this:

1. If candidate load (`ldl_l`) to an address is followed by store (`stl_c` of the lock acquire) to same address, predict another store (lock release) will shortly follow, restoring the memory location value to the one prior to this store (`stl_c` of the lock acquire).
2. Predict memory operations in critical sections will occur atomically, and elide lock acquire.
3. Execute critical section speculatively and buffer results.
4. If hardware cannot provide atomicity, trigger misspeculation, recover and explicitly acquire lock.
5. If second store (lock release) of step 1 seen, atomicity was not violated (else a misspeculation would have been triggered earlier). Elide lock-release store, commit state, and exit speculative critical section.

Note, in the above revised algorithm, the hardware needs no semantic information about whether the memory access is to a lock variable. The hardware only tracks changes in values and observes requests from other threads. If the store of step 5 does not match the value requirement outlined in step 1, the store is simply performed. If, on the completion of the store, atomicity was still maintained, the critical section can be safely exited.

3.4 Speculative Lock Elision algorithm example

Figure 4 shows the application of SLE to our earlier example of Figure 1a. The modified control flow is shown on the right with instructions 6 and 16 elided. All threads proceed without serialization. Instructions 1 and 3 bring the _lock_ into the cache in a shared state. Instruction 6 is elided and the modified control flow is speculatively executed. The location `_lock_` is monitored for writes by other threads. All loads executed by the processor are recorded. All stores executed are temporarily buffered. If instruction 16 is reached without any atomicity violations, SLE is successful.

If the thread cannot record accesses between the two stores, or the hardware cannot provide atomicity, a misspeculation is triggered, and execution restarts from instruction 6. On a restart, if the restart threshold has been reached, the execution occurs non-speculatively and the lock is acquired.

4 Why does SLE work correctly?

We now discuss why SLE guarantees a correct execution even in the absence of precise information from the software and independent of nesting levels and memory ordering. As mentioned earlier, SLE involves two predictions:

1. On a store, predict that another store will shortly follow and undo the changes by this store. The prediction is resolved without stores being performed but it requires the memory location (of the stores) to be monitored. If the prediction is validated, the two stores are elided.
2. Predict that all memory operations within the window bounded by the two elided stores occur atomically. This prediction is resolved by checking for conditions outlined in Section 3.2 using cache coherence mechanisms described in Section 5.3.

The above predictions do not rely on semantics of the program (a lock-predictor is used to identify loads/stores as

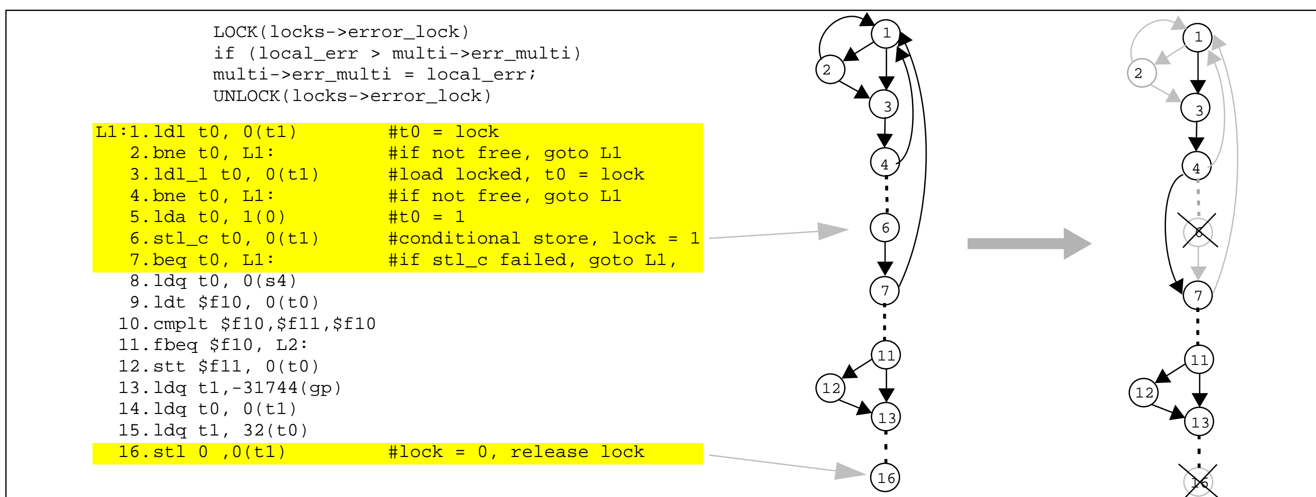

Figure 4. ***Speculative Lock Elision algorithm example.*** *Often, branch 11 is taken thus skipping the store inst. 12. The greyed portion on the right graph is not executed. Inst. 6 and 16 are elided and the code sequence executes with no taken branches between i1 and i8.*

candidates for prediction 1 but is not integral to the idea and software could alternatively provide these hints). In addition, no partial updates are made visible to other threads. Doing so guarantees critical section semantics. Store elision works because the architectural state remains the same. The architectural state at the end of the second elided store is the same, with or without SLE.

If another thread explicitly acquires the lock by writing to it, a misspeculation is triggered because the write will be automatically observed by all speculating threads. This trivially guarantees correctness even when one thread is speculating and another thread acquires the lock.

Nested locks. While it is possible to apply the elision algorithm to multiple nested locks, we apply it to only one level (can be any level and not necessarily the outermost) and any lock operations within this level are treated as speculative memory operations.

Memory consistency. No memory ordering problems exist because speculative memory operations under SLE have the appearance of atomicity. Regardless of the memory consistency model, it is always correct for a thread to insert an *atomic* set of memory operations into the global order of memory operations.

5 Implementing SLE

We have shown how SLE can be used to remove unnecessary synchronization operations dynamically. Now, we will show how SLE can be implemented using well understood and commonly used techniques.

SLE is similar to branch prediction, and other techniques for speculative execution. The elided lock acquire can be viewed as a predicted branch, and the elided lock release is similar to the resolution of the branch. However, SLE does not require the processor to support out-of-order execution but simply the ability to *speculatively retire* instructions. In other words, inter-instruction dependence information need not be maintained.

The four aspects of implementing SLE are 1) initiating speculation, 2) buffering speculative state, 3) misspeculation conditions and their detection, and 4) committing speculative memory state.

5.1 Initiating speculation

A filter is used to detect candidates for speculation (e.g., `ldl_l/stl_c` pairs) and is indexed by the program counter. Additionally, a confidence metric is assigned to each of these pairs. If the processor predicts a lock to be held, it assumes another processor had to acquire the lock because of its inability to elide the lock. In this case, the processor does not initiate speculation. This is a conservative approach but helps prevent performance degradation under pathological cases. Better confidence estimation is an important area of future research.

5.2 Buffering speculative state

To recover from an SLE misspeculation, register and memory state must be buffered until SLE is validated.

Speculative register state. Two simple techniques for handling register state are:

1. *Reorder buffer (ROB):* Using the reorder buffer [35] has the advantage of using recovery mechanisms already used for branch mispredictions. However, the size of the ROB places a limit on the size of the critical section (in terms of dynamic instructions).
2. *Register checkpoint:* This may be of dependence maps (there may be certain restrictions on how physical registers are freed) or of the architected register state itself. On a misspeculation, the checkpoint is restored. Using a checkpoint removes the limitation on the size of critical sections: instructions can safely update the register file, speculatively retire, and be removed from the ROB because a correct architected register checkpoint exists for recovery in case of misspeculation. Importantly, only *one* such checkpoint is required and is taken at the start of the SLE sequence.

Speculative memory state. Although most modern processors support speculative load execution, they do not retire stores speculatively (i.e., write to the memory system speculatively). For supporting SLE, we augment existing processor write-buffers (between the processor and L1 cache) to buffer speculative memory updates. Speculative data is not committed from the write-buffer into the lower memory hierarchy until the lock elision is validated. On a misspeculation, speculative entries in the write-buffer are invalidated.

As an additional benefit, under SLE, speculative writes can now be merged in the write-buffer, *independent* of the memory consistency model. This is possible because, for successful speculation, all memory accesses are guaranteed to appear to complete *atomically*. Only the write-buffer size limits the number of unique cache lines modified in the critical section and does not restrict the dynamic number of executed store instructions in the critical section.

5.3 Misspeculation conditions and their detection

Two reasons for misspeculation to occur are 1) atomicity violations and 2) violations due to limited resources.

Atomicity violations. Atomicity violations (Section 3.2) can be detected using existing cache coherence schemes. Cache coherence is a mechanism to propagate memory updates to other caches and make memory operations visible to others. Invalidation-based coherence protocols guarantee an exclusive copy of the memory block in the local cache when a store is performed. Most modern processors already implement some form of invalidation-based coher-

ency as part of their local cache hierarchy. Thus, the basic mechanism for detecting conflicts among memory operations from different processors already exists. One now needs a mechanism to record memory addresses read and written within a critical section.

In some processors—such as the MIPS R10K, and the Intel Pentium 4—the load/store queue (LSQ) is snooped on any external invalidation received by the cache to allow for aggressive memory consistency implementations [12]. If the ROB approach is used for SLE, no additional mechanisms are required for tracking external writes to memory locations speculatively read—the LSQ is already snooped.

If the register checkpoint approach is used, the LSQ alone cannot be used to detect load violations for SLE because loads may speculatively retire and leave the ROB. In this case, the cache can be augmented with an *access bit* for each cache block. Every memory access executed during SLE marks the access bit of the corresponding block. On an external request, this bit is checked in parallel with the cache tags. Any external invalidation to a block with its access bit set, or an external request to a block in exclusive state with its access bit set, triggers a misspeculation. The bit can be stored with the tags and there is no overhead for the bit comparison because, for maintaining coherency, a tag lookup is already performed to snoop the cache.

The scheme is independent of the number of cache levels because all caches maintain coherency and any update is propagated to all coherent caches automatically by the existing protocols.

On misspeculations and commits, the access bit is unset for all blocks in the cache; this can be done using techniques such as flash invalidation [26]. For aggressive out-of-order processors, when a candidate store (for the lock-acquire elision) is decoded, any subsequent load issued by the processor marks the appropriate access bit. The load may not actually be part of the critical section but marking it conservatively will always be correct. Multiple candidate stores (predicted lock-acquires) can be allowed in the pipe and as long as there is a candidate store in the core, loads conservatively mark the access bit.

Violations due to resource constraints. Limited resources may force a misspeculation if either there is not enough buffer space to store speculative updates, or it is not possible to monitor accessed data to provide atomicity guarantees. Four such conditions for misspeculation are:

1. Finite cache size. If register checkpoint is used, the cache may not be able to track all memory accesses.
2. Finite write-buffer size. The number of unique cache lines modified exceeds the write-buffer size.
3. Finite ROB size. If the checkpoint approach is used, the ROB size is not a problem.
4. Uncached accesses or events (e.g., some system calls) where the processor cannot track requests.

It may not always be necessary to restart for conditions 1, 2, and 3. The processor could simply write to the lock marking it acquired. When the write completes, if atomicity has been maintained, speculation can be committed and the processor can continue without restarting.

5.4 Committing speculative memory state

We have discussed recovering and committing architected register state, buffering speculative store state in the write-buffer, and detecting misspeculation conditions using existing cache coherence protocols. Committing memory state requires ensuring that speculatively buffered writes be committed and made visible *instantaneously* to the memory system (to provide atomicity).

Caches have two aspects: 1) state, and 2) data. The cache coherence protocol determines state transitions of cache block state. Importantly, these state transitions can occur speculatively as long as the data is not changed speculatively. This is how speculative loads and exclusive prefetches (operations that bring data in exclusive state into caches) are issued in modern processors. We use these two aspects in performing atomic memory commit *without* making any change to the cache coherence protocol.

When a speculative store is added to the write-buffer, an exclusive request is sent to the memory system. This request initiates pre-existing state transitions in the coherence protocol and brings the cache block into the local cache in the exclusive state. Note the *cache block data is not speculative*—speculative data is buffered in the write-buffer. When the critical section ends, all speculative entries in the write-buffer will have a corresponding block in exclusive state in the cache, otherwise a misspeculation would have been triggered earlier. At this point, the write-buffer is marked as having the latest architectural state.

The write-buffer requires an additional functionality of being able to source data for requests from other threads. This is not in the critical path and the write-buffer can be lazily drained into the cache as needed. The instantaneous commit is possible because the process of marking the write-buffer as having the latest state involves setting one bit—exclusive permissions have already been obtained for all speculatively updated and buffered blocks.

Figure 5 shows two design points: (a) uses the ROB to store speculative state, and (b) uses an extra register checkpoint and access bits with the cache tags.

6 Evaluation methodology

A multithreaded benchmark can have different control flow depending upon the underlying coherence mechanism and performance benefits are highly dependent on the underlying protocol implementation. To address this issue, we evaluate multiple configurations. Table 1 shows the

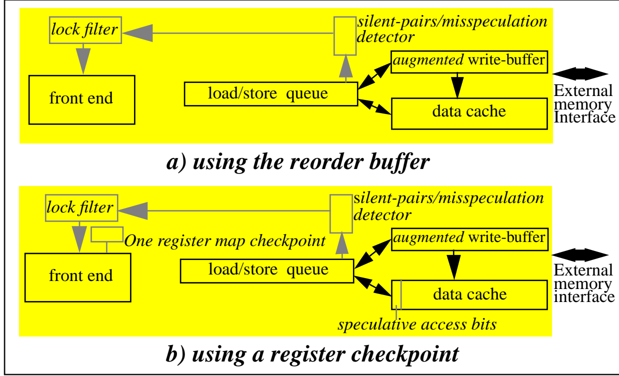

Figure 5. ***Two design points for speculative lock elision.*** *The shaded box is the processor. Modifications and additional data paths are shown italicized and in grey lines.*

parameters for three multiprocessor systems: a) a chip multiprocessor (CMP), b) a more conventional bus system (SMP), and c) a directory system (DSM). The bus protocol is based on the Sun Gigaplane [6] and the directory protocol is based on the SGI Origin 2000 [25]. The processor implements Total Store Ordering (TSO) as a memory consistency model. Retired stores are written to the write-buffer from which they are made architecturally visible in program order. On a coherency event, all in-flight loads are snooped and replayed if necessary. We use a single register checkpoint for SLE register recovery and a 32 entry lock predictor indexed by the program counter.

6.1 Simulation environment

We use SimpleMP, an execution-driven simulator for running multithreaded binaries. The simulator is derived from the Simplescalar toolset [4]. Our simulator is rewritten to model accurately an out-of-order processor 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 TSO implementations.

6.2 Benchmarks

We evaluate our proposal using a simple microbenchmark and six applications (Table 2). The microbenchmark consists of N threads, each incrementing a unique counter (2^{16})/N times, and all N counters are protected by a single lock. This is a worst case behavior for conventional locking but clearly demonstrates the potential of our scheme. For the six applications, we use `mp3d`, `barnes`, and `cholesky` from the SPLASH [34] suite and `radiosity`, `water-nsq`, and `ocean` from the SPLASH2 [40] suite.

Application	Type of Simulation	Inputs	Type of Critical Sections
Barnes	N-Body	4K bodies	cell locks, nested
Cholesky	Matrix factoring	tk14.O	task queues, col. locks
Mp3D	Rarefied field flow	24000 mols, 25 iter.	cell locks
Radiosity	3-D rendering	-room, batch mode	task queues, nested
Water-nsq	Water molecules	512 mols, 3 iter.	global structure
Ocean-cont	Hydrodynamics	x130	conditional updates

Table 2: ***Benchmarks***

These applications have been selected for their varying lock behavior, memory access patterns, and critical section behavior. These benchmarks have been appropriately padded to reduce false sharing. A locking version of `Mp3d` was used in order to study the impact of SLE on a lock-intensive benchmark [20]. This version of `Mp3d` does frequent synchronization to largely uncontended locks and lock access latencies cannot be hidden by a large reorder buffer. `Cholesky` and `radiosity` have work queues that are accessed frequently. `Ocean-cont` has conditional update code sequences. `Barnes` has high lock and data contention, while `water-nsq` has little contention.

These benchmarks have been optimized for sharing and thus have little communication in most cases. We are interested in determining the robustness and potential of our proposal even for these well-tuned benchmarks.

Processor L1 caches	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/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 (each entry 64-bytes wide) 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. minimum 1 cycle occupancy for request/response queues between l1 and l2.
CMP	Sun Gigaplane-type MOESI protocol between L1s, split transaction. Address bus: broadcast network, snoop latency of 20 cycles, 120 outstanding transactions. L2 cache: perfect, 12 cycle access. Data network: point-to-point, pipelined, transfer latency: 20 cycles
SMP	Sun Gigaplane-type MOESI protocol between L2s, split transaction. Address bus: broadcast network, snoop latency of 30 cycles, 120 outstanding transactions. L2 cache: unified, 4-MB, 4-way, 12 cycle access, 16 pending misses. Data network: point-to-point, pipelined, 70 cycles transfer latency. Memory access: 70 cycles for 64 bytes.
DSM	SGI Origin-2000-type MESI protocol between L2s. L2 cache: unified, 4-MB, 4-way, 12 cycle access, 16 pending misses. Directory: full mapped, 70 cycle access (overlapped with memory access). Network latencies: processor to local directory (70 ns), directory and remote route (50 ns). Some uncontended latencies: read miss to local memory: ~130 ns, read miss to remote memory: ~230 ns, read miss to remote dirty cache: ~360 ns.

Table 1. ***Simulated machine parameters.***

7 Results

Significant performance benefits can accrue from SLE in applications that perform frequent synchronization. We first discuss the performance of SLE in a microbenchmark and then analyze the performance for a set of applications.

7.1 Microbenchmark results

Figure 6 plots the execution time (for the CMP configuration) of the microbenchmark on the y-axis and varying processor count on the x-axis. As expected, the behavior under conventional locking quickly degrades because of severe contention. Even though the counter updates do not conflict, out-of-order processors cannot exploit this aspect because the lock-acquire sequence masks this parallelism and the performance is limited by lock-acquires. However, with SLE, the hardware automatically detects that the lock is not required and elides it. Perfect scalability is achieved because SLE does not require the lock to be acquired (by writing to it) for validating the speculation.

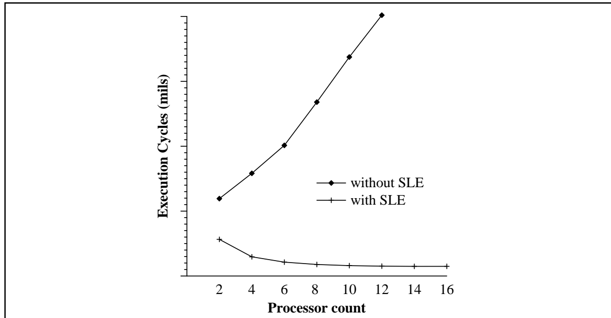

Figure 6. ***Microbenchmark result for CMP.***

7.2 Benchmark results

A parameter that can be varied under SLE is the restart threshold. This determines how many misspeculations a processor can experience before explicitly acquiring the lock in order to perform the critical section. We ran experiments for varying thresholds and present results for a threshold of 1—a processor restarts once after an atomicity violation while in SLE mode, and tries to elide the lock again. Note that if a lock is held by a processor, other processors will not attempt SLE but rather will spin (or wait) on the lock (waiting for a free lock before attempting SLE) as determined by the original control flow sequence of the program. Doing so guarantees that the processor holding the lock will not observe critical section data access interference by other processors.

Elided locks. Figure 7 shows the percentage of dynamic lock acquire/release pairs elided for a restart threshold of 1. A large fraction of dynamic lock acquires are elided. This reduction does not always translate to better performance because these operations may not have been on the critical path of the program, but it demonstrates the effectiveness

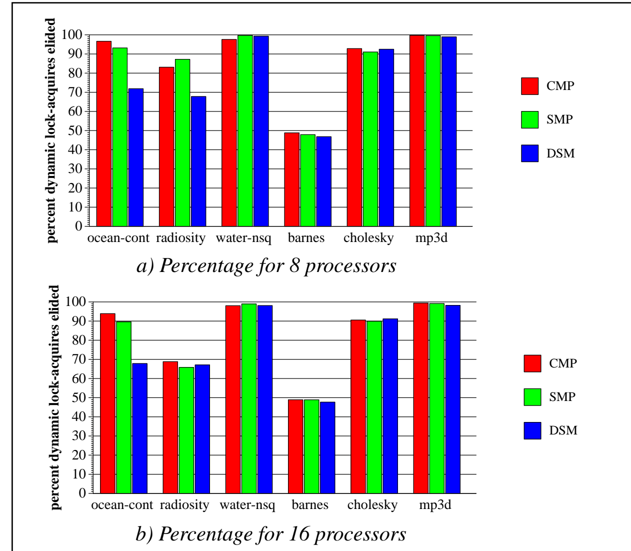

Figure 7. ***Percentage of dynamic lock acquires/releases elided***

of our technique. A threshold of 0 (restart on the first misspeculation) resulted in 10-30% fewer lock acquires being elided. In `barnes`, there is high contention for the protected data, and repeated restarts tend to end in conflicts. Thus, for `barnes`, the number of locks elided is low.

Performance. Figure 8 show the normalized execution times for 8 and 16 processors. The y-axis is normalized execution time (parallel execution cycles with SLE/parallel execution cycles without SLE). *A number below 1 corresponds to a speedup*. For each bar, the upper portion corresponds to the contribution due to lock variable accesses (LOCK-PORTION), and the lower portion corresponds to the rest (NON-LOCK-PORTION). The fractions on top of the bar-pairs are normalized execution times for the SLE case.

For some configurations, the NON-LOCK-PORTION for the optimized case is larger than the corresponding 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 get exposed and stall the processor.

Three primary reasons for the observed performance gains are: 1) concurrent critical section execution, 2) reduced observed memory latencies, and 3) reduced memory traffic.

Concurrent critical section execution. In `ocean-cont`, `radiosity`, and `cholesky`, even though locks are contended, critical sections can sometimes be correctly executed without acquiring the lock and thread execution is not serialized on the lock. The gains increase with larger memory latencies because the serialization delays induced due to lock acquire latencies are also greater.

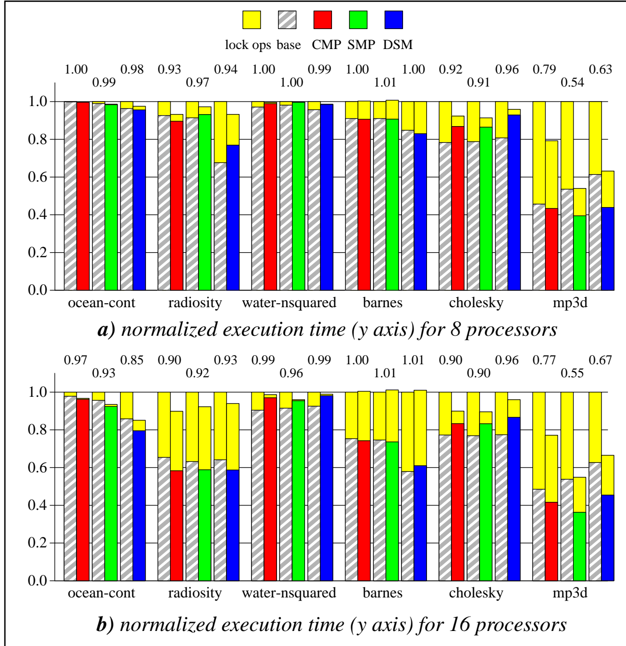

Figure 8. *Normalized execution time for 8 and 16 threads for CMP, SMP, and DSM configurations*. *The y-axis is normalized parallel execution time. Three pairs of bars, corresponding to CMP, SMP, and DSM respectively, are shown for each benchmark. The first bar of each pair corresponds to the base case. The second bar of each pair corresponds to the SLE case. For each bar, the upper portion corresponds to the contribution due to lock variable accesses (LOCK-PORTION), and the lower portion corresponds to the rest (NON-LOCK-PORTION). The fractions on top of the bar-pairs are normalized execution times for the SLE case. The LOCK-PORTION also contains time spent waiting for a lock. All normalizations are w.r.t. the base case (left bar of each pair).*

Reduced observed memory latencies. Often, a lock acquire causes a cache miss and a write request to the memory system, the latencies of which cannot be completely overlapped with misses within the critical section. SLE permits locks to remain shared in the local cache, and thus the processor does not observe the lock acquire misses. Nearly all benchmarks benefit from this. `Water-nsq` does not benefit much because after lock elision, the overlapped critical section misses get exposed.

Reduced memory traffic. If a lock is acquired and kept in exclusive state in a processor's cache, a request for the lock from another processor will require bus/network messages (one for reading the lock and another for setting it). Thus, for benchmarks with frequent synchronization, removing requests to locks can help due to reduced memory traffic. By not acquiring the lock, the lock is kept in shared state locally on the various processors and miss traffic is eliminated. The benchmark gaining the most from this effect is `mp3d` because of elision of frequent synchronization. Some lock access latency remains because some locks do undergo cache miss latencies that could not be overlapped. The SMP and DSM versions gain more than CMP because their large caches can hold the working set and thus have fewer read misses (and memory traffic) for the lock. For the CMP, the absence of a large cache hurts and thus there are more evictions of locks in clean state because the L1 suffers conflict and capacity misses.

Misspeculation effects. In our experiments, misspeculation due to capacity and cache conflicts (due to limited associativity) occurred in less than 0.5% of all cases. For atomicity conflicts we do an early restart and acquire the lock and reduce the misspeculation penalties. The NON-LOCK-PORTION also contains some of the misspeculation-induced memory latencies.

Dependence on restart threshold. A restart threshold of 0 gave up to 25% lesser speedups than those for a threshold of 1. Increasing the threshold resulted in greater performance improvements for some benchmarks. However, for `barnes` with 16 processors, performance degraded as much as 10% for a threshold of 5. This is because misspeculating processors introduce coherence protocol interference, thereby increasing the observed latency for data within critical sections. Selecting a low threshold (0 or 1) has the advantage of minimizing the degradation that can occur due to repeated misspeculations. For these benchmarks and a threshold of 1, we rarely degrade performance (loss of 1%). Increasing the threshold may sometimes lead to slightly worse performance, even though more locks are elided, because of data access interference within a critical section. Predictors for picking restart thresholds dynamically is an area of future work.

8 Related work

Lamport introduced lock-free synchronization [24] and gave algorithms to allow multiple threads to work on a data structure without a lock. Operations on lock-free data structures support concurrent updates and do not require mutual exclusion. Lock-free data structures have been extensively investigated [3, 15]. Experimental studies have shown software implementations of lock-free data structures do not perform as well as their lock-based counterparts primarily due to excessive data copying involved to enable rollback, if necessary [1, 16].

Transactional memory [16] and the Oklahoma update protocol [37] were the initial proposals for hardware support for implementing lock-free data structures. Both provided programmers with special memory instructions for accessing these data structures. Although conceptually powerful, the proposals required instruction set support

and programmer involvement. The programmer had to learn the correct use of new instructions and the proposal required coherence protocol extensions. Additionally, existing program binaries could not benefit. The proposals relied on software support for guaranteeing forward progress. These proposals were both direct generalizations of the LOAD-LINKED and STORE-CONDITIONAL instructions originally proposed by Jensen et al. [19]. The LOAD-LINKED/STORE-CONDITIONAL combination allows for atomic read-modify-write sequences on a word.

In contrast to the above proposals, our proposal does not require instruction set changes, coherence protocol extensions, or programmer support. As a result, we can run unmodified binaries in a lock-free manner in most cases when competing critical section executions have no conflict. We do not have to provide special support for forward progress because, for conflicts, we simply fall back to the original code sequence, acquiring and releasing the lock in the conventional way.

Extensive research has been conducted in databases on concurrency control and Thomasian [39] provides a good summary and further references. Optimistic Concurrency Control (OCC) was proposed by Kung and Robinson [23] as an alternative to locking in database management systems. 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. In spite of extensive research, OCC is not common as a concurrency control mechanism for database systems. An excellent discussion regarding the issues involved with OCC approaches and their shortcomings which makes it unattractive for high performance database systems is provided by Mohan [28]. Special requirements and guarantees required by database systems [29] make OCC hard to use for high performance. To provide these guarantees, substantial state information must be stored in software resulting in large overheads. In addition, with OCC, the validation phase is serialized.

Our proposal is quite different from database OCC proposals. We are not providing an alternative to lock-based synchronization: we detect dynamic instances when these synchronization operations are unnecessary and *remove* them. The requirements imposed on critical sections are far less strict than those for database systems. Since we do not require explicit acquisition of a lock to determine success, we do not have a serialized validation phase.

Prior work exists in microarchitectural support for speculative retirement [31, 13] and buffering speculative data in caches [11, 14, 36]. Our work can leverage these techniques and coexist with them. However, none of these earlier techniques dynamically remove conservative synchronization from the dynamic instruction stream. Predicting lock-acquires and releases has been proposed earlier and we use similar techniques [30].

Our scheme of silent pair elision is an extension to the *silent store* proposal of Lepak and Lipasti [27]. While they squashed individual silent store operations, we elide pairs of stores that individually are not silent but when executed as a pair are silent.

9 Concluding remarks

We have proposed a microarchitectural technique to remove unnecessary serialization from a dynamic instruction stream. The key insight is that locks do not have to be acquired but only need to be observed. With our technique, the control dependence implied by the lock operation is converted to a true data dependence among the various concurrent critical sections. As a result, the potential parallelism masked by dynamically unnecessary and conservative locking imposed by a programmer-based static analysis is exposed by a hardware-based dynamic analysis.

The technique proposed does not require any coherence protocol changes. Additionally, no programmer or compiler support and no instruction set changes are necessary. The key notion of atomicity of memory operations enables the technique to be incorporated in any processor without regard to memory consistency as correctness is guaranteed without any dependence on memory ordering.

We view our proposal as a step towards enabling high performance multithreaded programming. With multiprocessing becoming more common, it is necessary to provide programmers with support for exploiting these multiprocessing features for functionality and performance. Speculative Lock Elision permits programmers to use frequent and conservative synchronization to write *correct* multithreaded code easily; our technique automatically and dynamically removes unnecessary instances of synchronization. Synchronization is performed only when necessary for correctness; and performance is not degraded by the presence of such synchronization. Since SLE is a purely microarchitectural technique, it can be incorporated into any system without dependence on coherence protocols or system design issues.

Acknowledgements

We would like to thank Mark Hill, Mikko Lipasti, and David Wood for valuable discussions regarding the ideas in the paper. We thank Brian Fields, Adam Butts, Trey Cain, Timothy Heil, Mark Hill, Herbert Hum, Milo Martin, Paramjit Oberoi, Manoj Plakal, Eric Rotenberg, Dan Sorin, Vijayaraghavan Soundararajan, David Wood, and Craig Zilles for comments on drafts of this paper. Jeffrey Naughton and C. Mohan provided us with information regarding OCC in database systems.

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] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowaczyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[3] 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.

[4] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. TR #1342, Computer Sciences Department, University of Wisconsin, Madison, June 1997.

[5] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, and M.J. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on the Management of Data*, pages 383–394, May 1994.

[6] A. Charlesworth, A. Phelps, R. Williams, and G. Gilbert. Gigaplane-XB: Extending the Ultra Enterprise Family. In *Proceedings of the Symposium on High Performance Interconnects V*, pages 97–112, August 1997.

[7] S-E. Choi and E.C. Lewis. A Study of Common Pitfalls in Simple Multi-Threaded Programs. In *Proceedings of the Thirty-First ACM SIGCSE Technical Symposium on Computer Science Education*, March 2000.

[8] *Alpha Architecture Handbook Version 4*, October 1998.

[9] Intel Corporation. Hyper-Threading Technology. http://developer.intel.com/technology/hyperthread/, 2001.

[10] International Business Machines Corporation. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufman, San Francisco, CA, 1998.

[11] M. Franklin and G.S. Sohi. A Hardware Mechanism for Dynamic Reordering of Memory References. In *IEEE Transactions on Computers, 45(5)*, May 1996.

[12] 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.

[13] 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.

[14] S. Gopal, T.N. Vijaykumar, J.E. Smith, and G.S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

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

[16] 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.

[17] M.D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, 1998.

[18] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual*, January 1996.

[19] 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.

[20] 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.

[21] J. Kahle. A Dual-CPU Processor Chip. In *Proceedings of the 1999 International Microprocessor Forum*, October 1999.

[22] T.S. Kawaf, D.J. Shakshober, and D.C. Stanley. Performance Analysis Using Very Large Memory on the 64-bit AlphaServer System. *Digital Technical Journal*, 8(3), 1996.

[23] 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.

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

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

[26] D.D. Lee and R.H. Katz. Using Cache Mechanisms to Exploit Nonrefreshing DRAM's for On-Chip Memories. *IEEE Journal of Solid-State Circuits*, 26(4), April 1991.

[27] K.M. Lepak and M.H. Lipasti. On the Value Locality of Store Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 182–191, June 2000.

[28] 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.

[29] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), March 1992.

[30] 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.

[31] 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.

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

[33] Silicon Graphics Inc., Mountain View, CA. *MIPS R10000 Microprocessor User's Manual Version 2.0*, 1996.

[34] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[35] J.E. Smith and A.R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[36] D.J. Sorin, M.M.K. Martin, M.D. Hill, and D.A. Wood. Fast Checkpoint/Recovery to Support Kilo-Instruction Speculation and Hardware Fault Tolerance. TR #1420, Computer Sciences Department, University of Wisconsin, Madison, October 2000.

[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] S. Storino and J. Borkenhagen. A Multi-Threaded 64-bit PowerPC Commercial RISC Processor design. In *Proceedings of the 11th Annual International Symposium on High-Performance Chips*, August 1999.

[39] A. Thomasian. Concurrency Control: Methods, Performance, and Analysis. *ACM Computing Surveys*, 30(1), March 1998.

[40] 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.