Transactional Memory: Architectural Support for Lock-Free Data Structures

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Laboratory
Cambridge MA 02139
herlihy@crl.dec.com

J. Eliot B. Moss
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
moss@cs.umass.edu

Abstract

A shared data structure is *lock-free* if its operations do not require mutual exclusion. If one process is interrupted in the middle of an operation, other processes will not be prevented from operating on that object. In highly concurrent systems, lock-free data structures avoid common problems associated with conventional locking techniques, including priority inversion, convoying, and difficulty of avoiding deadlock. This paper introduces *transactional memory*, a new multiprocessor architecture intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion. Transactional memory allows programmers to define customized read-modify-write operations that apply to multiple, independently-chosen words of memory. It is implemented by straightforward extensions to any multiprocessor cache-coherence protocol. Simulation results show that transactional memory matches or outperforms the best known locking techniques for simple benchmarks, even in the absence of priority inversion, convoying, and deadlock.

1 Introduction

A shared data structure is *lock-free* if its operations do not require mutual exclusion. If one process is interrupted in the middle of an operation, other processes will not be prevented from operating on that object. Lock-free data structures avoid common problems associated with conventional locking techniques in highly concurrent systems:

- *Priority inversion* occurs when a lower-priority process is preempted while holding a lock needed by higher-priority processes.
- *Convoying* occurs when a process holding a lock is descheduled, perhaps by exhausting its scheduling quantum, by a page fault, or by some other kind of interrupt. When such an interruption occurs, other processes capable of running may be unable to progress.
- *Deadlock* can occur if processes attempt to lock the same set of objects in different orders. Deadlock avoidance can be awkward if processes must lock multiple data objects, particularly if the set of objects is not known in advance.

A number of researchers have investigated techniques for implementing lock-free concurrent data structures using software techniques [2, 4, 19, 25, 26, 32]. Experimental evidence suggests that in the absence of inversion, convoying, or deadlock, software implementations of lock-free data structures often do not perform as well as their locking-based counterparts.

This paper introduces *transactional memory*, a new multiprocessor architecture intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion. Transactional memory allows programmers to define customized read-modify-write operations that apply to multiple, independently-chosen words of memory. It is implemented by straightforward extensions to multiprocessor cache-coherence protocols. Simulation results show that transactional memory is competitive with the best known lock-based techniques for simple benchmarks, even in the absence of priority inversion, convoys, and deadlock.

In Section 2, we describe transactional memory and how to use it. In Section 3 we describe one way to implement transactional memory, and in Section 4 we discuss some

alternatives. In Section 5 we present some simulation results, and in Section 6, we give a brief survey of related work.

2 Transactional Memory

A *transaction* is a finite sequence of machine instructions, executed by a single process, satisfying the following properties:

- *Serializability*: Transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another. Committed transactions are never observed by different processors to execute in different orders.
- *Atomicity*: Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either *commits*, making its changes visible to other processes (effectively) instantaneously, or it *aborts*, causing its changes to be discarded.

We assume here that a process executes only one transaction at a time. Although the model could be extended to permit overlapping or logically nested transactions, we have seen no examples where they are needed.

2.1 Instructions

Transactional memory provides the following primitive instructions for accessing memory:

- *Load-transactional* (LT) reads the value of a shared memory location into a private register.
- *Load-transactional-exclusive* (LTX) reads the value of a shared memory location into a private register, "hinting" that the location is likely to be updated.
- *Store-transactional* (ST) tentatively writes a value from a private register to a shared memory location. This new value does not become visible to other processors until the transaction successfully commits (see below).

A transaction's *read set* is the set of locations read by LT, and its *write set* is the set of locations accessed by LTX or ST. Its *data set* is the union of the read and write sets.

Transactional memory also provides the following instructions for manipulating transaction state:

- *Commit* (COMMIT) attempts to make the transaction's tentative changes permanent. It *succeeds* only if no other transaction has updated any location in the transaction's data set, and no other transaction has read any location in this transaction's write set. If it succeeds, the transaction's changes to its write set become visible to other processes. If it *fails*, all changes to the write set are discarded. Either way, COMMIT returns an indication of success or failure.
- *Abort* (ABORT) discards all updates to the write set.
- *Validate* (VALIDATE) tests the current transaction status. A *successful* VALIDATE returns *True*, indicating that the current transaction has not aborted (although it may do so later). An *unsuccessful* VALIDATE returns *False*, indicating that the current transaction has aborted, and discards the transaction's tentative updates.

By combining these primitives, the programmer can define customized read-modify-write operations that operate on arbitrary regions of memory, not just single words. We also support *non-transactional* instructions, such as LOAD and STORE, which do not affect a transaction's read and write sets.

For brevity, we leave undefined how transactional and non-transactional operations interact when applied to the same location.[1] We also leave unspecified the precise circumstances that will cause a transaction to abort. In particular, implementations are free to abort transactions in response to certain interrupts (such as page faults, quantum expiration, etc.), context switches, or to avoid or resolve serialization conflicts.

2.2 Intended Use

Our transactions are intended to replace short critical sections. For example, a lock-free data structure would typically be implemented in the following stylized way (see Section 5 for specific examples). Instead of acquiring a lock, executing the critical section, and releasing the lock, a process would:

1. use LT or LTX to read from a set of locations,
2. use VALIDATE to check that the values read are consistent,
3. use ST to modify a set of locations, and
4. use COMMIT to make the changes permanent. If either the VALIDATE or the COMMIT fails, the process returns to Step (1).

[1] One sensible way to define such interactions is to consider a LOAD or STORE as a transaction that always commits, forcing any conflicting transactions to abort.

A more complex transaction, such as one that chains down a linked list (see Figure 3), would alternate LT and VALIDATE instructions. When contention is high, programmers are advised to apply adaptive backoff [3, 28] before retrying.

The VALIDATE instruction is motivated by considerations of software engineering. A set of values in memory is *inconsistent* if it could not have been produced by any serial execution of transactions. An *orphan* is a transaction that continues to execute after it has been aborted (i.e., after another committed transaction has updated its read set). It is impractical to guarantee that every orphan will observe a consistent read set. Although an orphan transaction will never commit, it may be difficult to ensure that an orphan, when confronted with unexpected input, does not store into out-of-range locations, divide by zero, or perform some other illegal action. All values read before a successful VALIDATE are guaranteed to be consistent. Of course, VALIDATE is not always needed, but it simplifies the writing of correct transactions and improves performance by eliminating the need for *ad-hoc* checks.

Our transactions satisfy the same formal serializability and atomicity properties as database-style transactions (viz. [18]), but they are intended to be used very differently. Unlike database transactions, our transactions are short-lived activities that access a relatively small number of memory locations in primary memory. The ideal size and duration of transactions are implementation-dependent, but, roughly speaking, a transaction should be able to run to completion within a single scheduling quantum, and the number of locations accessed should not exceed an architecturally-specified limit.

3 Implementation

In this section, we give an overview of an architecture that supports transactional memory. An associated technical report [20] gives detailed protocols for both bus-based (snoopy cache) and network-based (directory) architectures.

Our design satisfies the following criteria:

- Non-transactional operations use the same caches, cache controller logic, and coherence protocols they would have used in the absence of transactional memory.
- Custom hardware support is restricted to primary caches and the instructions needed to communicate with them.
- Committing or aborting a transaction is an operation local to the cache. It does not require communicating with other processes or writing data back to memory.

Transactional memory is implemented by modifying standard multiprocessor cache coherence protocols. We exploit access rights, which are usually connected with cache residence. In general, access may be non-exclusive (shared) permitting reads, or exclusive, permitting writes. At any time a memory location is either (1) not immediately accessible by any processor (i.e., in memory only), (2) accessible non-exclusively by one or more processors, or (3) accessible exclusively by exactly one processor. Most cache coherence protocols incorporate some form of these access rights.

The basic idea behind our design is simple: any protocol capable of detecting accessibility conflicts can also detect transaction conflict at no extra cost. Before a processor P can load the contents of a location, it must acquire non-exclusive access to that location. Before another processor Q can store to that location, it must acquire exclusive access, and must therefore detect and revoke P's access. If we replace these operations with their transactional counterparts, then it is easy to see that any protocol that detects potential access conflicts also detects the potential transaction conflict between P and Q.

Once a transaction conflict is detected, it can be resolved in a variety of ways. The implementation described here aborts any transaction that tries to revoke access of a transactional entry from another active transaction. This strategy is attractive if one assumes (as we do) that timer (or other) interrupts will abort a stalled transaction after a fixed duration, so there is no danger of a transaction holding resources for too long. Alternative strategies are discussed in [20].

3.1 Example implementation

We describe here how to extend Goodman's "snoopy" protocol for a shared bus [15] to support transactional memory. (See [20] for similar extensions to a directory-based protocol.) We first describe the general implementation strategy, the various cache line states, and possible bus cycles. We then describe the various possible actions of the processor and the bus snooping cache logic.

3.1.1 General approach

To minimize impact on processing non-transactional loads and stores, each processor maintains two caches: a *regular cache* for non-transactional operations, and a *transactional cache* for transactional operations. These caches are exclusive: an entry may reside in one or the other, but not both. Both caches are primary caches (accessed directly by the processor), and secondary caches may exist between them and the memory. In our simulations, the regular cache is a conventional direct-mapped cache. The transactional

Name	Access	Shared?	Modified?
INVALID	none	—	—
VALID	R	Yes	No
DIRTY	R, W	No	Yes
RESERVED	R, W	No	No

Table 1: Cache line states

Name	Meaning
EMPTY	contains no data
NORMAL	contains committed data
XCOMMIT	discard on commit
XABORT	discard on abort

Table 2: Transactional tags

cache is a small, fully-associative cache with additional logic to facilitate transaction commit and abort. The overall hardware organization is similar to that used by Jouppi for the *victim cache* [22], and indeed one can readily extend the transactional cache to act as a victim cache as well.

The idea is that the transactional cache holds all the tentative writes, *without* propagating them to other processors or to main memory unless the transaction commits. If the transaction aborts, the lines holding tentative writes are dropped (invalidated); if the transaction commits, the lines may then be snooped by other processors, written back to memory upon replacement, etc. We assume that since the transactional cache is small and fully associative it is practical to use parallel logic to handle abort or commit in a single cache cycle.

3.1.2 Cache line states

Following Goodman, each cache line (regular or transactional) has one of the states in Table 1. The possible accesses permitted are reads and/or writes; the "Shared?" column indicates whether sharing is permitted; and the "Modified?" column indicates whether the line may differ from its copy in main memory.

The transactional cache augments these states with separate *transactional tags* shown in Table 2, used as follows. Transactional operations cache two entries: one with transactional tag XCOMMIT and one XABORT. Modifications are made to the XABORT entry. When a transaction commits, it sets the entries marked XCOMMIT to EMPTY, and XABORT to NORMAL. When it aborts, it sets entries marked XABORT to EMPTY, and XCOMMIT to NORMAL.

When the transactional cache needs space for a new entry, it first searches for an EMPTY entry, then for a NORMAL entry, and finally for an XCOMMIT entry. If the XCOMMIT entry is DIRTY, it must be written back. Notice that XCOMMIT entries are used only to enhance performance. When a ST tentatively updates an entry, the old value must be retained in case the transaction aborts. If the old value is resident in the transactional cache and dirty, then it must either be marked XCOMMIT, or it must be written back to memory. Avoiding such write-backs can substantially enhance performance when a processor repeatedly executes transactions that access the same locations. If contention is low, then the transactions will often hit dirty entries in the transactional cache.

Name	Kind	Meaning	New access
READ	regular	read value	shared
RFO	regular	read value	exclusive
WRITE	both	write back	exclusive
T_READ	trans	read value	shared
T_RFO	trans	read value	exclusive
BUSY	trans	refuse access	unchanged

Table 3: Bus cycles

3.1.3 Bus cycles

The various kinds of bus cycles are listed in Table 3. The READ (RFO (read-for-ownership)) cycle acquires shared (exclusive) ownership of the cache line. The WRITE cycle updates main memory when the protocol does write through; it is also used when modified items are replaced. Further, memory snoops on the bus so if a modified item is read by another processor, the main memory version is brought up to date. These cycles are all as in Goodman's original protocol. We add three new cycles. The T_READ and T_RFO cycles are analogous to READ and RFO, but request cache lines transactionally. Transactional requests can be *refused* by responding with a BUSY signal. BUSY helps prevent transactions from aborting each other too much. When a transaction receives a BUSY response, it aborts and retries, preventing deadlock or continual mutual aborts. This policy is theoretically subject to starvation, but could be augmented with a queueing mechanism if starvation is a problem in practice.

3.1.4 Processor actions

Each processor maintains two flags: the *transaction active* (TACTIVE) flag indicates whether a transaction is in progress, and if so, the *transaction status* (TSTATUS) flag indicates whether that transaction is active (*True*) or aborted

(*False*). The TACTIVE flag is implicitly set when a transaction executes its first transactional operation. (This implicit approach seems more convenient than providing an explicit *start transaction* instruction.) Non-transactional operations behave exactly as in Goodman's original protocol. Transactional instructions issued by an aborted transaction cause no bus cycles and may return arbitrary values.[2]

We now consider transactional operations issued by an active transaction (TSTATUS is *True*). Suppose the operation is a LT instruction. We probe the transactional cache for an XABORT entry, and return its value if there is one. If there is no XABORT entry, but there is a NORMAL one, we change the NORMAL entry to an XABORT entry, and allocate a second entry with tag XCOMMIT and the same data.[3] If there is no XABORT or NORMAL entry, then we issue a T_READ cycle. If it completes successfully, we set up two transactional cache entries, one tagged XCOMMIT and one XABORT, both with whatever state the Goodman protocol would get on a READ cycle. If we get a BUSY response, we abort the transaction (set TSTATUS to *False*, drop all XABORT entries, and set all XCOMMIT entries to NORMAL) and return arbitrary data.

For LTX we use a T_RFO cycle on a miss rather than a T_READ, and change the cache line state to RESERVED if the T_RFO succeeds. A ST proceeds like a LTX, except it updates the XABORT entry's data. The cache line state is updated as in the Goodman protocol with LT and LTX acting like LOAD and ST acting like STORE.

The VALIDATE instruction returns the TSTATUS flag, and if it is *False*, sets the TACTIVE flag to *False* and the TSTATUS flag to *True*. The ABORT instruction discards cache entries as previously described, and sets TSTATUS to *True* and TACTIVE to *False*. Finally, COMMIT returns TSTATUS, sets TSTATUS to *True* and TACTIVE to *False*, drops all XCOMMIT cache entries, and changes all XABORT tags to NORMAL.

Interrupts and transactional cache overflows abort the current transaction.

3.1.5 Snoopy cache actions

Both the regular cache and the transactional cache snoop on the bus. A cache ignores any bus cycles for lines not in that cache. The regular cache behaves as follows. On a READ or T_READ, if the state is VALID, the cache returns the value. If the state is RESERVED or DIRTY, the cache returns the value and resets the state to VALID. On a RFO or T_RFO, the cache returns the data and invalidates the line.

The transactional cache behaves as follows. If TSTATUS is *False*, or if the cycle is non-transactional (READ and RFO), the cache acts just like the regular cache, except that it ignores entries with transactional tag other than NORMAL. On T_READ, if the state is VALID, the cache returns the value, and for all other transactional operations it returns BUSY.

Either cache can issue a WRITE request when it needs to replace a cache line. The memory responds only to READ, T_READ, RFO, and T_RFO requests that no cache responds to, and to WRITE requests.

[2] As discussed below in Section 4, it is possible to provide stronger guarantees on values read by aborted transactions.

[3] Different variations are possible here. Also, allocating an entry may involve replacing a dirty cache entry, in which case it must be written back, as previously mentioned.

4 Rationale

It would be possible to use a single cache for both transactional and non-transactional data. This approach has two disadvantages: (1) modern caches are usually set associative or direct mapped, and without additional mechanisms to handle set overflows, the set size would determine the maximum transaction size, and (2) the parallel commit/abort logic would have to be provided for a large primary cache, instead of the smaller transactional cache.

For programs to be portable, the instruction set architecture must guarantee a minimum transaction size, thus establishing a lower bound for the transactional cache size. An alternative approach is suggested by the LimitLESS directory-based cache coherence scheme of Chaiken, Kubiatowicz, and Agarwal [6]. This scheme uses a fast, fixed-size hardware implementation for directories. If a directory overflows, the protocol traps into software, and the software emulates a larger directory. A similar approach might be used to respond to transactional cache overflow. Whenever the transactional cache becomes full, it traps into software and emulates a larger transactional cache. This approach has many of the same advantages as the original LimitLESS scheme: the common case is handled in hardware, and the exceptional case in software.

Other transactional operations might be useful. For example, a simple "update-and-commit" operation (like STORE_COND) would be useful for single-word updates. It might also be convenient for a transaction to be able to drop an item from its read or write set. Naturally, such an operation must be used with care.

One could reduce the need for VALIDATE instructions by guaranteeing that an orphan transaction that applies a LT or LTX instruction to a variable always observes some value previously written to that variable. For example, if a shared variable always holds a valid array index, then it would not be necessary to validate that index before using it. Such a change would incur a cost, however, because an orphan transaction might sometimes have to read the variable's value from memory or another processor's cache.

5 Simulations

Transactional memory is intended to make lock-free synchronization as efficient as conventional lock-based techniques. In this section, we present simulation results suggesting that transactional memory is competitive with well-known lock-based techniques on simple benchmarks. Indeed, transactional memory has certain inherent advantages: for any object that spans more than a single word of memory, techniques based on mutual exclusion must employ an explicit lock variable. Because transactional memory has no such locks, it typically requires fewer memory accesses.

We modified a copy of the Proteus simulator [5] to support transactional memory. Proteus is an execution-driven simulator system for multiprocessors developed by Eric Brewer and Chris Dellarocas of MIT. The program to be simulated is written in a superset of C. References to shared memory are transformed into calls to the simulator, which manages the cache and charges for bus or network contention. Other instructions are executed directly, augmented by cycle-counting code inserted by a preprocessor. Proteus does not capture the effects of instruction caches or local caches.

We implemented two versions of transactional memory, one based on Goodman's snoopy protocol for a bus-based architecture, and one based on the Chaiken directory protocol for a (simulated) Alewife machine [1]. Our motive in choosing these particular protocols was simply ease of implementation: the Proteus release includes implementations of both. As noted below, a more complex snoopy protocol could make spin locks more efficient.

Both simulated architectures use 32 processors. The regular cache is a direct-mapped cache with 2048 lines of size 8 bytes, and the transactional cache has 64 8-byte lines. In both architectures, a memory access (without contention) requires 4 cycles. The network architecture uses a two-stage network with wire and switch delays of 1 cycle each.

The ability to commit and abort transactions quickly is critical to the performance of transactional memory. In our simulations, each access to the regular or transactional cache, including transaction commit and abort, is counted as a single cycle. Single-cycle commit requires that the transactional cache provide logic to reset the transactional tag bits in parallel. Moreover, commit must not force newly-committed entries back to memory. Instead, in the implementations simulated here, committed entries are gradually replaced as they are evicted or invalidated by the ongoing cache coherence protocol.

We constructed three simple benchmarks, and compared transactional memory against two software mechanisms and two hardware mechanisms. The software

```
shared int counter;

void process(int work)
{
  int success = 0, backoff = BACKOFF_MIN;
  unsigned wait;

  while (success < work) {
    ST(&counter, LTX(&counter) + 1);
    if (COMMIT()) {
      success++;
      backoff = BACKOFF_MIN;
    }
    else {
      wait = random() % (01 << backoff);
      while (wait--);
      if (backoff < BACKOFF_MAX)
        backoff++;
    }
  }
}
```

Figure 1: Counting Benchmark

```
typedef struct {
  Word deqs;
  Word enqs;
  Word items[QUEUE_SIZE];
} queue;

unsigned queue_deq(queue *q) {
  unsigned head, tail, result;
  unsigned backoff = BACKOFF_MIN
  unsigned wait;
  while (1) {
    result = QUEUE_EMPTY;
    tail = LTX(&q->enqs);
    head = LTX(&q->deqs);
   /* queue not empty? */
    if (head != tail) {
      result =
       LT(&q->items[head % QUEUE_SIZE]);
      /* advance counter */
      ST(&q->deqs, head + 1);
    }
    if (COMMIT()) break;
    /* abort => backoff */
    wait = random() % (01 << backoff);
    while (wait--);
    if (backoff < BACKOFF_MAX)
      backoff++;
  }
  return result;
}
```

Figure 2: Part of Producer/Consumer Benchmark

```
typedef struct list_elem{
  /* next to dequeue */
  struct list_elem *next;
  /* previously enqueued */
  struct list_elem *prev;
  int value;
} entry;

shared entry *Head, *Tail;

void list_enq(entry* new) {

  entry *old_tail;
  unsigned backoff = BACKOFF_MIN;
  unsigned wait;

  new->next = new->prev = NULL;

  while (TRUE) {
    old_tail = (entry*) LTX(&Tail);
    if (VALIDATE()) {
      ST(&new->prev, old_tail);
      if (old_tail == NULL) {
        ST(&Head, new);
      } else {
        ST(&old_tail->next, new);
      }
      ST(&Tail, new);
      if (COMMIT()) return;
    }
    wait = random() % (01 << backoff);
    while (wait--);
    if (backoff < BACKOFF_MAX)
      backoff++;
  }
```

Figure 3: Part of Doubly-Linked List Benchmark

mechanisms were (1) test-and-test-and-set (TTS) [30] spin locks with exponential backoff [3, 28], and (2) software queueing [3, 17, 27]. The hardware mechanisms were (1) LOAD_LINKED/STORE_COND (LL/SC) with exponential backoff, and (2) hardware queueing [16]. For a single-word counter benchmark, we ran the LL/SC implementation directly on the shared variable, while on the others we used LL/SC to implement a spin lock. Both software mechanisms perform synchronization in-line, and all schemes that use exponential backoff use the same fixed minimum and maximum backoff durations. We now give a brief review of these techniques.

A *spin lock* is perhaps the simplest way to implement mutual exclusion. Each processor repeatedly applies a *test-and-set* operation until it succeeds in acquiring the lock. As discussed in more detail by Anderson [3], this naïve technique performs poorly because it consumes excessive amounts of processor-to-memory bandwidth. On a cache-coherent architecture, the *test-and-test-and-set* [30] protocol achieves somewhat better performance by repeatedly rereading the cached value of the lock (generating no memory traffic), until it observes the lock is free, and then applying the *test-and-set* operation directly to the lock in memory. Even better performance is achieved by introducing an exponential delay after each unsuccessful attempt to acquire a lock [3, 27]. Because Anderson and Mellor-Crummey et al. have shown that TTS locks with exponential backoff substantially outperform conventional TTS locks on small-scale machines, it is a natural choice for our experiments.

The LL operation copies the value of a shared variable to a local variable. A subsequent SC to that variable will succeed in changing its value only if no other process has modified that variable in the interim. If the operation does not succeed, it leaves the shared variable unchanged. The LL/SC operations are the principal synchronization primitives provided by the MIPS II architecture [29] and Digital's Alpha [31]. On a cache-coherent architecture, these operations are implemented as single-word transactions — a SC succeeds if the processor retains exclusive access to the entry read by the LL.

In *software queuing*, a process that is unable to acquire a lock places itself on a software queue, thus eliminating the need to poll the lock. Variations of queue locks have been proposed by Anderson [3], by Mellor-Crummey and Scott [27], and by Graunke and Thakkar [17]. Our simulations use the algorithm of Mellor-Crummey and Scott. In *hardware queuing*, queue maintenance is incorporated into the cache coherence protocol itself. The queue's head is kept in memory, and unused cache lines are used to hold the queue elements. The directory-based scheme must also keep the queue tail in memory. Our simulations use a

queuing scheme roughly based on the QOSB mechanism of Goodman et al. [16].

5.1 Counting Benchmark

In our first benchmark (code in Figure 1), each of n processes increments a shared counter $2^{16}/n$ times, where n ranges from 1 to 32. In this benchmark, transactions and critical sections are very short (two shared memory accesses) and contention is correspondingly high. In Figure 4, the vertical axis shows the number of cycles needed to complete the benchmark, and the horizontal axis shows the number of concurrent processes. With one exception, transactional memory has substantially higher throughput than any of the other mechanisms, at all levels of concurrency, for both bus-based and directory-based architectures. The explanation is simple: transactional memory uses no explicit locks, and therefore requires fewer accesses to shared memory. For example, in the absence of contention, the TTS spin lock makes at least five references for each increment (a read followed by a test-and-set to acquire the lock, the read and write in the critical section, and a write to release the lock). Similar remarks apply to both software and hardware queueing.

By contrast, transactional memory requires only three shared memory accesses (the read and write to the counter, and the commit, which goes to the cache but causes no bus cycles). The only implementation that outperforms transactional memory is one that applies LL/SC directly to the counter, without using a lock variable. Direct LL/SC requires no commit operation, and thus saves a cache reference. In the other benchmarks, however, this advantage is lost because the shared object spans more than one word, and therefore the only way to use LL/SC is as a spin lock.

Several other factors influence performance. Our implementation of hardware queuing suffers somewhat from the need to access memory when adjusting the queue at the beginning and end of each critical section, although this cost might be reduced by a more sophisticated implementation. In the bus architecture, the TTS spin lock suffers because of an artifact of the particular snoopy cache protocol we adapted [15]: the first time a location is modified, it is marked *reserved* and written back. TTS would be more efficient with a cache protocol that leaves the location *dirty* in the cache.

5.2 Producer/Consumer Benchmark

In the *producer/consumer* benchmark (code in Figure 2), n processes share a bounded FIFO buffer, initially empty. Half of the processes produce items, and half consume them. The benchmark finishes when 2^{16} operations have completed. In the bus architecture (Figure 5), all throughputs are essentially flat. Transactional memory has higher throughputs than the others, although the difference is not as dramatic as in the counting benchmark. In the network architecture, all throughputs suffer somewhat as contention increases, although the transactional memory implementations suffers least.

5.3 Doubly-Linked List Benchmark

In the *doubly-linked list* benchmark (code in Figure 3) n processes share a doubly-linked list anchored by *head* and *tail* pointers. Each process dequeues an item by removing the item pointed to by *tail*, and then enqueues it by threading it onto the list at *head*. 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} operations have completed.

This example is interesting because it has potential concurrency that is difficult to exploit by conventional means. When the queue is non-empty, each transaction modifies *head* or *tail*, but not both, so enqueuers can (in principle) execute without interference from dequeuers, and vice-versa. When the queue is empty, however, transactions must modify both pointers, and enqueuers and dequeuers conflict. This kind of state-dependent concurrency is not realizable (in any simple way) using locks, since an enqueuer does not know if it must lock the *tail* pointer until after it has locked the *head* pointer, and vice-versa for dequeuers. If an enqueuer and dequeuer concurrently find the queue empty, they will deadlock. Consequently, our locking implementations use a single lock. By contrast, the most natural way to implement the queue using transactional memory permits exactly this parallelism. This example also illustrates how VALIDATE is used to check the validity of a pointer before dereferencing it.

The execution times appear in Figure 6. The locking implementations have substantially lower throughput, primarily because they never allow enqueues and dequeues to overlap.

5.4 Limitations

Our implementation relies on the assumption that transactions have short durations and small data sets. The longer a transaction runs, the greater the likelihood it will be aborted by an interrupt or synchronization conflict[4]. The larger the data set, the larger the transactional cache needed, and (perhaps) the more likely a synchronization conflict will occur.

[4]The identical concerns apply to current implementations of the LOAD_LINKED and STORE_COND instructions [31, Appendix A].

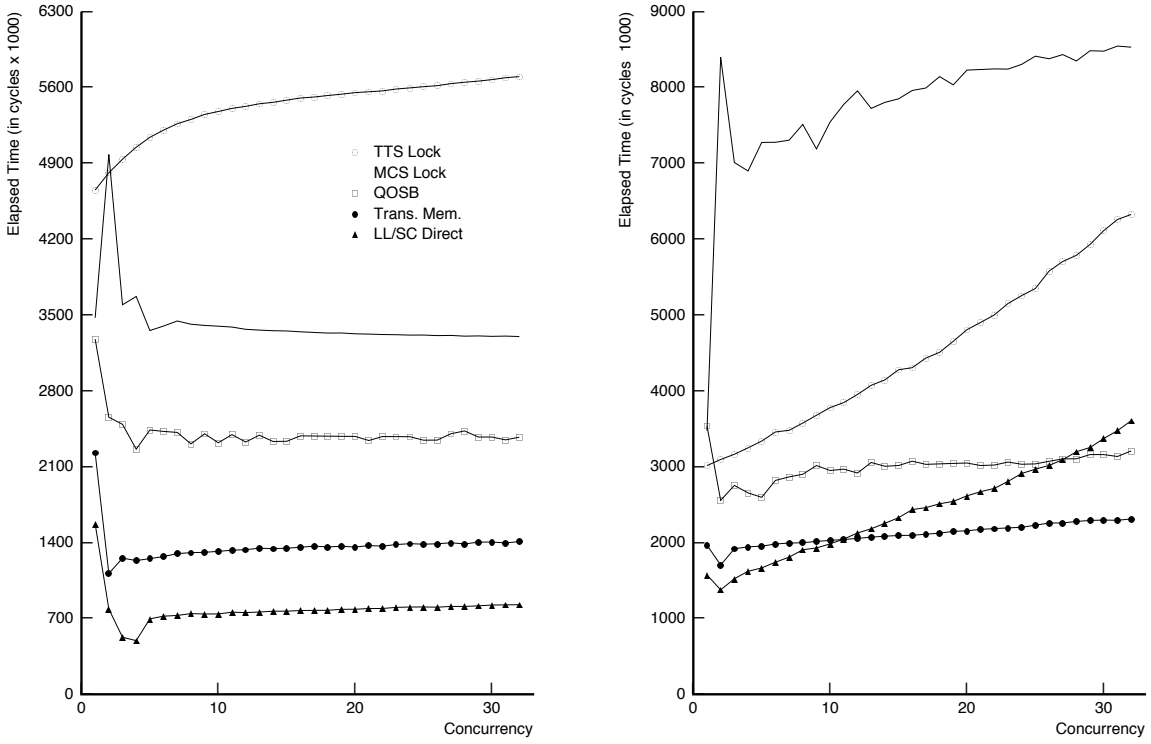

Figure 4: Counting Benchmark: Bus and Network

Such size and length restrictions are reasonable for applications that would otherwise have used short critical sections, but not for applications that would otherwise lock large objects for a long time (such as navigating a B-link tree with a large node size). Support for larger and longer transactions would require more elaborate hardware mechanisms.

The implementation described here does not guarantee forward progress, relying instead on software-level adaptive backoff to reduce the abort rate by spacing out conflicting transactions. Our simulations suggest that adaptive backoff works reasonably well when conflicting transactions have approximately the same duration. If durations differ, however, then longer transactions will be more likely to abort. Some kind of hardware queueing mechanism [16] might alleviate this limitation.

The cache coherence protocols used in our simulations provide a sequentially consistent memory [24]. A number of researchers have proposed weaker notions of correctness that permit more efficient implementations. These alternatives include processor consistency [14], weak consistency [9, 8], release consistency [13], and others[5]. Most of these models guarantee that memory will appear to be sequentially consistent as long as the programmer executes a *barrier* (or *fence*) instruction at the start and finish of each critical section. The most straightforward way to provide transactional memory semantics on top of a weakly-consistent memory is to have each transactional instruction perform an implicit barrier. Such frequent barriers would limit performance. We believe our implementation can be extended to require barriers only at transaction start, finish, and validate instructions.

[5] See Gharachorloo et al. [12] for concise descriptions of these models as well as performance comparisons.

6 Related Work

Transactional memory is a direct generalization of the LOAD_LINKED and STORE_COND instructions originally proposed by Jensen et al. [21], and since incorporated into the MIPS II architecture [29] and Digital's Alpha [31]. The LOAD_LINKED instruction is essentially the same as LTX, and STORE_COND is a combination of ST and COMMIT. The LOAD_LINKED/STORE_COND combination can implement any read-modify-write operation, but it is restricted to a single word. Transactional memory has the same flexibility, but can operate on multiple, independently-chosen words.

We are not the first to observe the utility of performing

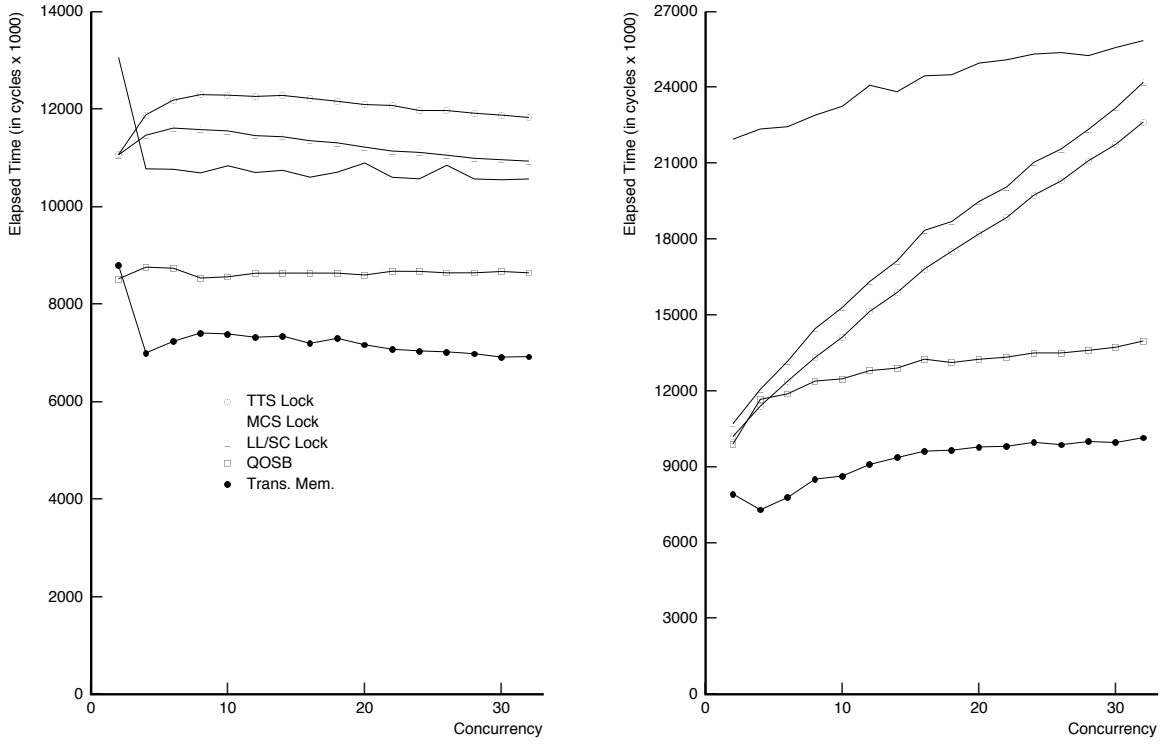

Figure 5: Producer/Consumer Benchmark: Bus and Network

atomic operations on multiple locations. For example, the Motorola 68000 provides a COMPARE&SWAP2 that operates on two independent locations. Massalin and Pu [25] use this instruction for lock-free list manipulation in an operating system kernel. Transactional memory provides more powerful support for this "lock-free" style of programming.

Other work that uses after-the-fact conflict detection to recognize violations of desired correctness conditions include Gharachorloo and Gibbons [11], who propose an implementation of release consistency that exploits an underlying invalidation-based cache protocol to detect violations of sequential consistency, and Franklin and Sohi [10], who propose a hardware architecture that optimistically parallelizes sequential code at runtime.

Other researchers who have investigated architectural support for multi-word synchronization include Knight [23], who suggests using cache coherence protocols to add parallelism to "mostly functional" LISP programs, and the IBM 801 [7], which provides support for database-style locking in hardware. Note that despite superficial similarities in terminology, the synchronization mechanisms provided by transactional memory and by the 801 are intended for entirely different purposes, and use entirely different techniques.

Our approach to performance issues has been heavily influenced by recent work on locking in multiprocessors, including work of Anderson [3], Bershad [4], Graunke and Thakkar [17], and Mellor-Crummey and Scott [27].

7 Conclusions

The primary goal of transactional memory is to make it easier to perform general atomic updates of multiple independent memory words, avoiding the problems of locks (priority inversion, convoying, and deadlock). We sketched how it can be implemented by adding new instructions to the processor, adding a small auxiliary, transactional cache (without disturbing the regular cache), and making straightforward changes to the cache coherence protocol. We investigated transactional memory for its added functionality, but our simulations showed that it outperforms other techniques for atomic updates. This is primarily because transactional memory uses no explicit locks and thus performs fewer shared memory accesses. Since transactional memory offers both improved functionality and better performance, it should be considered in future processor architectures.

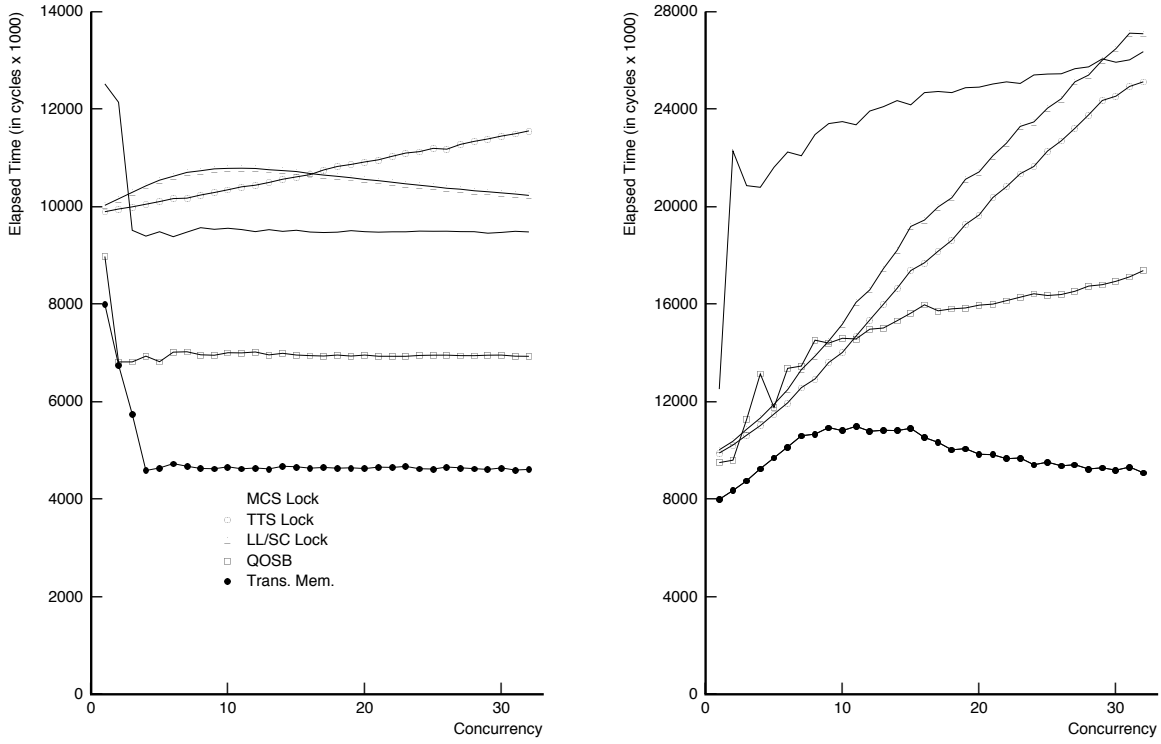

Figure 6: Doubly-Linked List Benchmark: Bus and Network

References

[1] A. Agarwal et al. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report TM-454, MIT Lab for Computer Science, 545 Technology Square, Cambridge MA 02139, March 1991. Extended version submitted for publication.

[2] J. Allemany and E.W. Felton. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 125–134. ACM, August 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, January 1990.

[4] B.N. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, Carnegie Mellon University, Pittsburgh, PA, September 1991.

[5] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[6] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: a scalable cache coherence scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Langauges and Operating Systems*, pages 224–234. ACM, April 1991.

[7] A. Chang and M.F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[8] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.

[9] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[10] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67. IEEE, May 1992.

[11] K. Gharachorloo and P. Gibbons. Detecting violations of sequential consistency. In *Proceedings*

of the 2nd Annual Symposium on Parallel Algorithms and Architectures, pages 316–326, July 1991.

[12] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[13] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[14] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.

[15] J.R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 124–131. IEEE, June 1983.

[16] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languates and Operating Systems*, pages 64–75, April 1989.

[17] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[18] J.N. Gray. *Notes on Database Operating Systems*, pages 393–481. Springer-Verlag, Berlin, 1978.

[19] M.P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, March 1990.

[20] M.P. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report 92/07, Digital Cambridge Research Lab, One Kendall Square, Cambridge MA 02139, December 1992.

[21] 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, November 1987.

[22] N. Jouppi. Improving direct mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual Internationall Symposium on Computer Architecture*, page 364. ACM SIGARCH, June 1990.

[23] T. Knight. An achitecture for mostly functional languages. In *Conference on Lisp and Functional Programming*, pages 105–112, August 1986.

[24] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[25] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University Computer Science Dept., 1991.

[26] J.M. Mellor-Crummey. Practical fetch-and-phi algorithms. Technical Report Technical Report 229, Computer Science Dept., University of Rochester, November 1987.

[27] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[28] R. Metcalfe and D. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[29] MIPS Computer Company. The MIPS RISC architecture.

[30] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[31] R.L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Maynard, MA, 1992.

[32] J. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(2), February 1993.