# Transactional Memory: Architectural Support for Lock-Free Data Structures

Maurice Herlihy J. Eliot B. Moss[1]

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/07 December 1, 1992

## 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 that supports lock-free implementations of complex data structures in a simple and efficient way. 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 ownership-based multiprocessor cache-coherence protocol. Simulation results show that transactional memory outperforms the best known locking techniques for simple benchmarks, even in the absence of priority inversion, convoying, and deadlock.

[1] Deptartment of Computer Science, University of Massachusetts, Amherst, MA 01003 (moss@cs.umass.edu). On leave at School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213

## 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 on conventional architectures [Bershad, 1991; Herlihy, 1990; Massalin and Pu, 1991; Mellor-Crummey, 1987; Wing and Gong, 1990].

This paper introduces *transactional memory*, a new multiprocessor architecture that supports lock-free implementations of complex data structures in a simple and efficient way. 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 ownership-based multiprocessor cache-coherence protocol. Simulation results show that transactional memory outperforms the best known locking techniques for simple benchmarks, even in the absence of priority inversion, convoying, and deadlock.

In Section 2, we introduce the semantics of transactional memory by defining a collection of new machine instructions and describing a programming style that exploits them. In Section 3 we describe one way to implement transactional memory, and briefly discuss other techniques. In Section 4 we give a detailed description of a bus-based ("snoopy") protocol, and in Section 5 we describe a network-based ("directory") protocol. In Section 6 we discuss some alternative designs. In Section 7 we describe some simulation results, and in Section 8, we give a survey of related work and a short discussion.

## 2 Semantics of 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.

The notion of a transaction originated in the database literature (viz. [Gray, 1978]). Our transactions satisfy the same formal serializability and atomicity properties, but they are implemented using different techniques, and they have different intended applications. Unlike database transactions, our transactions are intended to be short-lived activities that access a relatively small number of memory locations in primary memory. The ideal size and duration of transactions is 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 a processor's cache size.

## 2.1 Transactional Memory Primitives

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, but additionally indicates that that location is likely to be updated.
- *Store-transactional* (ST) tentatively writes the value in 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.

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 read or write 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.

| Operation | Meaning |
|---|---|
| r = LOAD(x) | Reads value of location x into register r. |
| STORE(x,v) | Writes v to location x. |
| r = LT(x) | Reads value of location x into register r for transaction. |
| r = LTX(x) | Like LT, but hints that x will probably be modified. |
| ST(x,v) | Writes v to location x for transaction. |
| r = COMMIT() | Tries to commit current transaction; returns success code. |
| ABORT() | Aborts current transaction; discards ST values. |
| r = VALIDATE() | Checks read set consistency; returns transaction status. |

Figure 1: Transactional Memory Primitives

- *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.

These *transactional instructions* are summarized in Figure 1. 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.

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.

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.

[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 lock-free data structure is typically implemented the following stylized way (see Section 7 for specific examples). Instead of acquiring a lock, executing the critical section, and releasing the lock, a process typically

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

For objects where contention is high, programmers are advised to apply some kind of adaptive backoff [Anderson, 1990; Metcalfe and Boggs, 1976] before retrying. A more complex transaction, such as one that chains down a linked list, would alternate LT and VALIDATE instructions.

## 3 Implementing Transactional Memory

In this section, we give an overview of an architecture that supports transactional memory. In later sections, we give detailed and specific protocols for a bus-based architecture (Section 4) and for an architecture employing an arbitrary interconnection network (Section 5). Here, however, we focus on basic principles.

Our design satisfies the following criteria:

- Non-transactional operations do not pay a penalty: they use the same caches, cache controller logic, and consistency protocols they would have used in the absence of transactional memory.
- Custom hardware support is restricted to caches and their controllers. Except for the instructions needed to communicate with the cache controller, we make no changes to standard processor architectures.
- Transaction commit and abort are local operations. In particular, we eschew distributed commitment protocols.

Transactional memory is implemented by modifying standard *ownership-based* cache consistency protocols. Ownership implies a right to access an item. In general, ownership may be non-exclusive (permitting reads), or exclusive (permitting writes). Ownership is usually connected with cache residence, but is a conceptually distinct property. At any given time a memory location is either (1) not owned by any processor, (2) owned non-exclusively by one or more processors, or (3) owned exclusively by exactly one processor. Most (but not all) shared bus (snoopy cache) protocols and directory cache protocols incorporate some form of ownership.

The basic idea behind our design is very simple: any protocol capable of detecting ownership 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 ownership of that location. Before another processor $Q$ can store to that location, it must acquire exclusive ownership, and must therefore detect and revoke $P$'s ownership. If we replace these operations with their transactional counterparts, then it is easy to see that any ownership-based protocol also detects the 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 ownership of a transactional entry from another active transaction. This strategy is attractive if one assumes (as we do) that timer interrupts, quantum exhaustion, page faults, and similar kinds of 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 below.

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 cache is a small, fully-associative cache with additional logic to facilitate transaction commit and abort. In addition to any tags used by the regular cache protocol, each transactional cache entry has a *transactional tag*, one of: TC_INVALID (invalid), TC_NORMAL (written by a committed transaction), TC_COMMIT (discard-on-commit), or TC_ABORT (discard-on-abort).

Transactional operations cache two entries: one with transactional tag TC_COMMIT and one TC_ABORT. Modifications are made to the TC_ABORT entry. When a transaction commits, it sets the entries marked TC_COMMIT to TC_INVALID, and TC_ABORT to TC_NORMAL. When it aborts, it sets entries marked TC_ABORT to TC_INVALID, and TC_COMMIT to TC_NORMAL. Because the transactional cache is relatively small, we assume that it is feasible to provide logic to reset these transactional tags in parallel.

When the transactional cache needs space for a new entry, it first searches for an invalid entry, then for a normal entry, and finally for a discard-on-commit entry. If the discard-on-commit entry is dirty, it must be written back as required by the cache consistency protocol. Notice that discard-on-commit 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 discard-on-commit, 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.

The size of the transactional cache must be at least the minimal transaction size guaranteed by the architecture. An alternative approach is suggested by the LimitLESS directory-based cache consistency scheme of Chaiken, Kubiatowicz, and Agarwal [Chaiken *et al.*, 1991]. 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.

All caches coordinate via a common cache consistency protocol. For example, in the snoopy cache protocol described in more detail below, all caches snoop on the bus. If a non-transactional operation results in a cache miss, the regular cache broadcasts the address on the bus. If a transactional cache has that entry, it will provide the TC_NORMAL or TC_COMMIT value. As a further optimization, if one provides a fast data path between a processor's regular and transactional caches, then the regular cache could use the transactional cache as a *victim cache* [Jouppi, 1990], where entries evicted from the normal cache are moved (as TC_NORMAL entries) to the nearby victim cache instead of the faraway memory.

Let us review the design considerations given above. Non-transactional operations interact exclusively with conventional caches executing conventional cache consistency protocols. Moreover, we expect that the locations accessed by transactional and non-transactional operations are largely disjoint (shared versus non-shared variables), so the two kinds of caches should not interfere with one another. Consequently, the principal cost of supporting transactional memory is the additional "real estate" needed to implement the transactional cache. Custom hardware support is limited to the cache and memory controllers, and to the transactional cache; except for providing instruction codes for communicating with the transactional cache, the processor itself is completely unaffected. Finally, the use of custom logic to reset transactional tags in parallel ensures that commit and abort operations are fast.

## 4 A Bus-Based Protocol

We show here how to modify the simplest ownership-based protocol, Goodman's "snoopy" protocol for a shared bus [Goodman, 1983]. (In a later section, we apply the same techniques to a directory-based protocol.) The regular cache runs Goodman's protocol without modification, using the same states, bus cycles, and transitions.

Each line in both the regular and transactional cache has a *state*, as summarized in Figure 2. These states are the same as in Goodman's protocol. Each entry in the transactional cache also has a *transactional tag*, summarized in Figure 3 (and described above). If an entry is cached in the transactional cache, then it is absent or invalid in the regular cache, and vice-versa. The protocol's bus cycles appear in Figure 4.

### 4.1 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.)

| Name | Permitted | Ownership | Modified? |
|---|---|---|---|
| C_INVALID | none | none | |
| C_VALID | read | non-exclusive | no |
| C_DIRTY | read and write | exclusive | yes |
| C_RESERVED | read and write | exclusive | no |

Figure 2: Bus Protocol: Cache Line States

| Name | Meaning |
|---|---|
| TC_INVALID | not present |
| TC_NORMAL | written by committed transaction |
| TC_COMMIT | discard on commit |
| TC_ABORT | discard on abort |

Figure 3: Bus Protocol: Transactional Tags

| Name | Issued | Meaning | Ownership |
|---|---|---|---|
| READ | regular | read value | non-exclusive |
| RFO | regular | read value | exclusive |
| WRITE | both | write back | exclusive |
| T_READ | transactional | read value | non-exclusive |
| T_RFO | transactional | read value | exclusive |
| BUSY | transactional | refuse access | unchanged |

Figure 4: Bus Protocol: Bus Cycles

```
int fill_cache_line(Word address)
{
  /* Check regular cache. */
  int line = CacheLookup(cache, address);
  if (line == MISS) {
    line = GetEntry(cache);
    cache[line].state = C_INVALID;
    }
  return line;
}
```

Figure 5: Pseudocode for Filling Regular Cache Line.

Non-transactional operations behave exactly as in Goodman's original protocol (Figure 6). 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*). Each process begins by filling a transactional cache entry as illustrated by pseudocode in Figure 7. The process queries the transactional cache for an entry with either the TC_NORMAL or TC_COMMIT flags set. If it misses, it creates a dummy invalid entry. If the resulting cache entry has transactional tag TC_NORMAL, the processor creates a second copy of the entry, marking one copy TC_ABORT (discard on abort), and the other TC_COMMIT (discard on commit).

As illustrated by pseudo-code in Figure 8, a processor executing LT first fills the transactional cache entry. If the entry state is C_INVALID, then the processor issues a READ cycle. If the response is BUSY, the processor aborts the transaction and returns an arbitrary value, and otherwise, it sets the entry state to C_VALID.

Like LT, LTX fills the transactional cache entry. If the entry is not C_RESERVED or C_DIRTY, it issues a transactional read for ownership cycle (T_RFO). If the response is BUSY, the processor aborts, otherwise it sets the entry to C_RESERVED (Figure 9). ST is almost exactly the same as LTX, except that it modifies the entry data and sets the state to C_DIRTY.

COMMIT and ABORT both set TACTIVE to *False*. COMMIT invalidates all TC_COMMIT entries in the transactional cache, and sets all TC_ABORT entries to TC_NORMAL, while ABORT invalidates all TC_ABORT entries and sets all TC_COMMIT entries to TC_NORMAL. If TACTIVE is *True*, VALIDATE returns *True*, otherwise it acts like ABORT.

[2] As discussed below in Section 6, it is possible to provide stronger guarantees on values read by aborted transactions.

```
data load(address)
{
  /* Fill regular cache line. */
  line = fill_cache_entry(address);
  if (cache[line].state == C_INVALID) {
    issue_bus_cycle(READ);
    response = get_bus_cycle();
    cache[line].data = response.data;
    /* New entry is valid because memory also snoops. */
    cache[line].state = C_VALID;
    break;
  }
  return cache[line].data;
}

data store(address, value)
{
  /* Fill regular cache line. */
  line = fill_cache_entry(address);
  switch (cache[line].state)
    {
    case C_VALID:
      /* First time => write back to memory */
      cache[line].data = value;
      issue_bus_cycle(WRITE(cache[line].data));
      cache[line].state = C_RESERVED;
      break;
    case C_RESERVED:
      /* After first time => mark dirty */
      cache[line].state = C_DIRTY;
      cache[line].data = value;
      break;
    case C_DIRTY:
      cache[line].data = value;
      break;
    case C_INVALID:
      issue_bus_cycle(RFO);
      response = get_bus_cycle();
      cache[line].data = value;
      cache[line].state = C_DIRTY;
      break;
    }
}
```

Figure 6: Goodman's Protocol.

```
int fill_tcache_line(address)
{
  int new_line, old_line;
  /* Check for normal or discard-on-commit entries. */
  new_line = TransCacheLookup(tcache, address, TC_NORMAL | TC_ABORT);
  /* No entry: create tentative and backup copies. */
  if (new_line == MISS) {
    old_line = GetEntry(tcache);
    tcache[old_line].state = C_INVALID;
    tcache[old_line].tag = TC_COMMIT;
    new_line = GetEntry(tcache);
    tcache[new_line].state = C_INVALID;
    tcache[new_line].tag = TC_ABORT;
  } else {
    /* Normal entry: mark it tentative and create backup. */
    if (tcache[new_line].tag == TC_NORMAL){
      old_line = GetEntry(tcache);
      tcache[old_line].data = tcache[new_line].data;
      tcache[old_line].state = tcache[new_line].state;
      tcache[new_line].tag = TC_ABORT;
      tcache[old_line].tag = TC_COMMIT;
    }
  }
  return new_line;
}
```

Figure 7: Pseudocode for Filling Transactional Cache Line.

```
data load_trans(address)
{
  /* Fill transactional cache line. */
  new_line = fill_tcache_entry(address);
  if (tcache[new_line].state == C_INVALID) {
    issue_bus_cycle(T_READ);
    switch (response = get_bus_cycle())
      {
      case BUSY:
        /* Abort and return arbitrary value */
        TStatus = FALSE;
        return 0;
      case DATA:
        /* Install data in transactional cache. */
        /* Note: Could also preallocate one additional entry. */
        tcache[new_line].data = response.data;
        tcache[new_line].state = C_VALID;
        break;
      }
  }
  return tcache[new_line].data;
}
```

Figure 8: Pseudocode for LT.

```
data load_trans_excl(address)
{
  /* Fill transactional cache line. */
  new_line = fill_tcache_entry(address);
  if (tcache[new_line].state != C_DIRTY &&
      tcache[new_line].state != C_RESERVED ) {
    issue_bus_cycle(T_RFO);
    switch (response = get_bus_cycle())
      {
      case BUSY:
        /* Abort and return arbitrary value */
        TStatus = FALSE;
        return 0;
      case DATA:
        /* Install data in transactional cache. */
        /* Note: Could also preallocate one additional entry. */
        tcache[new_line].data = response.data;
        tcache[new_line].state = C_RESERVED;
        break;
      }
  }
  return tcache[new_line].data;
}
```

Figure 9: Pseudocode for LTX.

```
regular_cache()
{
  while (TRUE) {
    request = get_bus_cycle();
    response.to = request.from;
    line = CacheLookup(cache, request.address);
    if (line != MISS){  /* hit! */
      response.data = cache[line].data;
      switch (request)
        {
        case READ:
        case T_READ:
          if (cache[line].state ==  C_RESERVED ||
              cache[line].state ==  C_DIRTY)
            cache[line].state = C_VALID;
          break;
        case RFO:
        case T_RFO:
          cache[line].state = C_INVALID;
        }
      issue_bus_cycle(response); /* implicitly writes back to memory */
    }
  }
}
```

Figure 10: Pseudocode for Regular Snooping Cache.

## 4.2 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 (Figure 10). On a READ or T_READ, if the state is C_VALID, the cache returns the value. If the state is C_RESERVED or C_DIRTY, the cache returns the value and resets the state to C_VALID. On a RFO or T_RFO, the cache returns the data and invalidates the line.

The transactional cache behaves as follows (Figure 11). If the current transaction is aborted (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 TC_NORMAL. On T_READ, if the state is C_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 (not shown in the pseudocode). The memory responds only to READ, T_READ, RFO, and T_RFO requests that no cache responds to, and to WRITE requests.

```
trans_cache()
{
  while (TRUE) {
    request = get_bus_cycle();
    response.to = request.from;
    tline = TransCacheLookup(tcache, request.address, TC_COMMIT);
    if (tline == MISS)
      tline = TransCacheLookup(tcache, request.address, TC_ABORT);
    if (tline != MISS){ /* cache hit */
      response.data = cache[line].data;
      if ((! TStatus) || request == READ || request == RFO) {
        /* Tag must be normal - act like regular cache. */
        switch (request)
          {
          case READ:
          case T_READ:
            if (cache[line].state ==  C_RESERVED ||
                cache[line].state ==  C_DIRTY)
              cache[line].state = C_VALID;
            break;
          case RFO:
          case T_RFO:
            cache[line].state = C_INVALID;
          }
        issue_bus_cycle(response);
      } else {
        /* Act like transactional cache */
        if (request == T_READ && tcache[tline].state = C_VALID)
          issue_bus_cycle(response);
        else
          issue_bus_cycle(BUSY);
      }                         /* else */
    }                           /* if */
  }                             /* while */
}
```

Figure 11: Pseudocode for Transactional Cache.

Interrupts and transactional cache overflows set TSTATUS to *False*.

## 4.3 Discussion

For brevity, we have chosen not to specify how transactional and non-transactional operations interact when applied concurrently to the same location. We expect that such a conflict is almost always an error. One reasonable choice is to abort a transaction when a non-transactional operation tries to revoke its ownership. Another choice is to signal some kind of error condition. The transactional cache protocol presented above can easily be extended to provide either behavior.

There are many opportunities for overlapping and pipelining the protocol given above. For example, an LTX that hits in the cache but needs an RFO cycle could proceed, postponing an examination of the results of the ownership request. Similar techniques apply to ST ownership. COMMIT, ABORT, and VALIDATE are operations local to the cache, and LT, LTX, and ST have essentially the same costs as their non-transactional counterparts.

Deadlock (cyclic waiting) is impossible in this implementation because transactions never wait for one another. A high-priority transaction cannot be delayed indefinitely by a lower-priority transaction, because the latter will be aborted by a timer interrupt if it runs too long. *Starvation*, however, is still possible. We believe that the best way to avoid starvation is to advise programmers to adopt an adaptive backoff strategy: a transaction that repeatedly aborts should wait for some duration before retrying. (A backoff strategy also helps reduce contention caused by BUSY bus cycles.) In the simulation results given below, we use a simple exponential backoff scheme. To alleviate priority inversion, a high-priority transaction might try more frequently than a low-priority transaction.

We originally considered incorporating a backoff strategy in the cache coherence protocol itself. Our simulations, however, show that backoff schemes need to be tuned to perform well, and so hardware backoff seems overly inflexible. Anderson [Anderson, 1990] reports a similar experience in alleviating contention for spin locks: exponential backoff works well, but the parameters must be chosen carefully.

As an alternative to relying on timer interrupts to abort slow transactions, we also considered a "grace point" scheme, in which each transaction is allowed to refuse a fixed number of attempts to revoke ownership. A transaction aborts after exhausting its grace points, and is then restarted with a larger number, up to a fixed maximum. In the benchmarks we simulated, the performance of the grace point scheme is indistinguishable from the performance of the timeout scheme. The principal disadvantage of the grace point scheme is that it requires a more complex cache controller.

The implementation given above is *pessimistic*: a transaction does not relinquish ownership to requesters. We also considered *optimistic* policies, in which the requester can preempt ownership (aborting the holder). Our simulation results showed that this strategy did not perform as well at high levels of contention. Although exponential backoff was sufficient to avoid cyclic restart, newer transactions tended to abort older transactions, causing more work to be discarded. A *hybrid* approach is an attractive way to eliminate priority inversion completely: higher-priority transactions can (optimistically) revoke ownership from lower-priority transactions, but not vice-versa.

## 5 A Directory-Based Protocol

In this section we show how to modify Chaiken's directory-based protocol [Chaiken, 1990; Chaiken *et al.*, 1991] to support transactional memory. The regular cache runs Chaiken's protocol without modification, using the same states, messages, and transitions.

The cache line states, shown in Figure 12, are the same as in Chaiken's protocol. States come in pairs: a "network wait" state indicates that there is an outstanding request to replace that cache line. Just as in the bus protocol, the transactional cache uses additional transactional tags on each entry (Figure 3). The protocol's messages appear in Figure 13.

A directory entry consists of *data*, a *state* (summarized in Figure 14), and an array of *pointers* to caches that hold copies of the data. The memory controller can distinguish between pointers to regular and transactional caches.

Many aspects of the directory protocol are the same as the bus protocol: COMMIT and ABORT instructions are identical, and transactional instructions issued by an aborted transaction may return an arbitrary value and cause no network traffic. Each process begins by filling a transactional cache entry in the same way.

Our protocol differs from the bus protocol in one important respect. In the bus protocol, an active transaction will always refuse to relinquish ownership of an entry to another transaction. In a directory protocol, such refusals introduce complications. For example, a transaction executing a ST might need to invalidate multiple non-exclusive owners. Some of these might refuse, and some might accept. The directory protocol would have to keep track of which is which, perhaps using one bit per owner. This structure is more complex than in Chaiken's original protocol, which uses a per-entry counter to detect when all outstanding acknowledgments have been received. The implementation we simulated is a compromise: an active transaction will never relinquish exclusive ownership, but will always relinquish non-exclusive ownership, so we can still use a counter to track outstanding acknowledgments.

For brevity we will review how our directory protocol handles LT; the other operations are similar. When an active transaction issues a LT, it repeatedly polls the local transactional cache (Figure 15). If the entry is valid, the cache returns the data. Otherwise, if the entry is invalid, the cache sends a TRREQ message to the memory controller. When the memory controller receives the TRREQ message, it checks the status of the directory entry (Figure 16). If the entry is missing, the memory controller installs the entry and grants the transaction non-exclusive ownership. If the entry is owned exclusively, the controller sends the owner a transactional invalidation message (Figure 17), and enters the D_READTRANS state until the invalidation is acknowledged. If the owner refuses, the refusal is forwarded to the requesting transaction, which will abort (Figure 18). If the entry is owned non-exclusively, the controller checks whether the transaction already has ownership. If not, it grants ownership, and in either case it sends back the data. If the list of non-exclusive owners overflows, the controller revokes ownership of one processor chosen at random (Figure 19). Finally, if the directory is in the middle of a read or write transaction, it returns a BUSY message and the requester will try again later (Figure 20).

| Name | Permitted | Ownership |
|---|---|---|
| C_INVALID,C_INVALID_NETW | none | none |
| C_READONLY,C_READONLY_NETW | reading | non-exclusive |
| C_READWRITE, C_READWRITE_NETW | reading and writing | exclusive |

Figure 12: Directory Protocol: Cache Line States

| Type | Name | Meaning |
|---|---|---|
| Reg. Cache to Memory | RREQ | read request |
| | WREQ | write request |
| | REPU | evict unmodified entry |
| | REPM | evict modified entry |
| | UPDATE | acknowledge invalidation (modified) |
| | ACKC | acknowledge invalidation (unmodified) |
| Trans. Cache to Memory | TRREQ | read request |
| | TWREQ | write request |
| | REFUSE | refuse invalidation |
| | UPDATE | acknowledge invalidation (modified) |
| | ACKC | acknowledge invalidation (unmodified) |
| Memory to Either Cache | INV | invalidate |
| | TINV | transactional invalidate |
| | RDATA | read data |
| | WDATA | write data |
| | BUSY | busy signal |

Figure 13: Directory Protocol: Messages

| Name | Meaning |
|---|---|
| D_ABSENT | absent |
| D_READONLY | several non-exclusive owners |
| D_READWRITE | one exclusive owner |
| D_READTRANS | holding read request, update in progress |
| D_WRITETRANS | holding write request, invalidation in progress |

Figure 14: Directory Protocol: Directory States

```
data load_trans(address)
{
  new_line = fill_tcache_entry(address);
  while (TRUE) {
    /* Already aborted? */
    if (! TStatus) return (0);
    switch (tcache[new_line].state)
      {
      case C_INVALID:
        send_to_memory(TRREQ, address);
        tcache[new_line].state = C_INVALID_NETW;
        break;
      case C_INVALID_NETW:
        break;
      case C_READONLY:
      case C_READONLY_NETW:
      case C_READWRITE:
      case C_READWRITE_NETW:
        return(tcache[new_line].data);
      }
    /* Poll again later. */
    SUSPEND;
  }
}
```

Figure 15: Directory Protocol: Transactional Load

```
void trreq_handler(Word address, int processor)
{ directory *d = DirectoryLookup(address);
  switch (d->status) {
  case D_ABSENT:
    d->dir[0] = processor; d->type[0] = TRANS; d->dircnt = 1;
    d->state = D_READONLY;
    send_to_tcache(processor, RDATA(memory[address]));
    return;
  case D_READONLY:
    for(i = 0; i < d->dircnt; i++)
      if (d->dir[i] == processor ) { /* Present => return data */
        send_to_tcache(processor, RDATA(memory[address]));
        return; }
    if (d->dircnt < DIRECTORY_SIZE ) { /* Absent, n < p => add it */
      d->dir[d->dircnt] = processor; d->type[d->dircnt] = TRANS;
      d->dircnt++;
      send_to_tcache(processor, RDATA(memory[address]));
    } else { /* Absent, n == p => replace randomly */
      int victim = random() % DIRECTORY_SIZE;
      if (d->type[victim] == REGULAR) send_to_cache(victim, INV(address));
      else send_to_tcache(victim, INV(address));
      d->ackctr++; d->dir[victim] = processor; d->type[victim] = TRANS;
      send_to_tcache(processor, RDATA(memory[address])); }
    return;
  case D_READWRITE:
    if (d->type[0] == REGULAR) send_to_cache(victim, INV(address));
    else send_to_tcache(victim, TINV(address));
    d->ackctr++;d->dir[0] = processor;d->type[0] = TRANS;
    d->state = D_READTRANS;
    return;
  case D_READTRANS: case D_WRITETRANS:
    if (d->type[0] == REGULAR) send_to_cache(processor, BUSY);
    else send_to_tcache(processor, BUSY);
    return; }
}
```

Figure 16: Memory Controller: Transactional Load

```
void tcache_TINV(address)
{
  int line = TransCacheLookup(tcache, address, TC_NORMAL | TC_ABORT);
  if (line == MISS) {
    send_to_memory(ACKC, address);
    return;
  }
  switch (tcache[line].state)
    {
    case C_INVALID:
    case C_INVALID_NETW:
      send_to_memory(ACKC, address);
      return;
    case C_READONLY:
    case C_READONLY_NETW:
      if (tcache[line].tag == TC_ABORT) {
        int old = TransCacheLookup(address, TC_COMMIT);
        if (old != MISS)
          tcache[old].state = C_INVALID;
      }
      TStatus = FALSE;
      send_to_memory(ACKC, address);
      return;
    case C_READWRITE:
    case C_READWRITE_NETW:
      if (tcache[line].tag == TC_NORMAL) {
        send_to_memory(UPDATE(address,tcache[line].data));
        tcache[line].state = C_INVALID;
      } else {
        send_to_memory(REFUSE, address);
      }
    }
}
```

Figure 17: Directory Protocol: Transactional Cache TINV

```
void tcache_REFUSE()
{
  TStatus = FALSE;
}
```

Figure 18: Directory Protocol: Transactional Cache REFUSE

## 6 Implementation Rationale

In this section, we review a number of alternative implementation strategies.

In our first explorations we designed systems having a single unified cache. This design was convenient for exposition, but not necessarily practical for implementation. Modern caches are usually direct mapped or set associative with a reasonably small set size, and if one does not add additional mechanism to handle set overflows, the set size restricts the worst case number of memory elements that can be involved in a transaction. Further, one must add transactional cache functionality to the entire, large cache, including (for reasonable efficiency and low complexity) the single cycle commit/abort logic, which might require large/slow drivers. In any case, hardware designers might be skeptical of achieving the same cache cycle time with a unified transactional cache as they would with a simple direct mapped cache.

This leads naturally to the notion of a separate transactional cache designed specifically for transactional operations, not interfering with the design of the regular cache. We did devise a scheme that probes both caches in parallel, giving preference to the transactional cache (which can and does hold non-transactional entries, e.g., after commit). This appears to require an extra stage of discrimination and multiplexing logic that again might interfere with achieving top performance from the regular cache.

Thus we arrive at separate caches that are not probed in parallel, but sequentially. That is, for regular memory operations one probes the regular cache, and then probes the transactional cache only on a regular cache miss. Similarly, transactional operations go to the transactional cache first, and probe the regular cache only on a transactional cache miss. As previously observed, this is closely related to the victim cache concept (which helped influence our thinking). One could certainly combine the transaction and victim caches; victims would be entered with a TC_NORMAL transaction state and the appropriate cache state (e.g., C_VALID, C_DIRTY, or C_RESERVED for the bus cache described earlier). As a slight performance improvement for systems where the processor cycle time is faster than the cache cycle time one could probe both caches in parallel, but check for misses sequentially; i.e., it would take one processor cycle longer to get an item from the "wrong" cache, rather than a complete cache cycle.

We also considered designs in which each transactional item consumes a single transactional cache entry, rather than two entries as described here. As we pointed out, the main problem with having only

```
void tcache_INV(address)
{
  int line = TransCacheLookup(tcache, address, TC_NORMAL | TC_ABORT);
  if (line == MISS) {
    send_to_memory(ACKC, address);
    return;
  }
  switch (tcache[line].state)
    {
    case C_INVALID:
    case C_INVALID_NETW:
      send_to_memory(ACKC, address);
      return;
    case C_READONLY:
    case C_READONLY_NETW:
      if (tcache[line].tag == TC_ABORT) {
        old = TransCacheLookup(address, TC_COMMIT);
        if (old != MISS)
          old.state = C_INVALID;
      }
      TStatus = FALSE;
      send_to_memory(ACKC, address);
      return;
    case C_READWRITE:
    case C_READWRITE_NETW:
      send_to_memory(UPDATE(address,tcache[line].data));
      tcache[line].state = C_INVALID;
    }
}
```

Figure 19: Directory Protocol: Transactional Cache Invalidation

```
void tcache_BUSY(address)
{
  line = TransCacheLookup(tcache, address, TC_NORMAL | TC_ABORT);
  if (line == MISS) {
    send_to_memory(ACKC, address);
    return;
  }
  switch (tcache[line].state)
    {
    case C_INVALID:
    case C_INVALID_NETW:
      tcache[line].state = C_INVALID;
      return;
    case C_READONLY_NETW:
      tcache[line].state = C_READONLY;
      return;
    case C_READWRITE_NETW:
      tcache[line].state = C_READWRITE;
    }
}
```

Figure 20: Directory Protocol: Transactional Cache BUSY

one entry is that if it is in state TC_NORMAL and is dirty, then transactional instructions must write it back to memory. Having two entries allows us to roll back as well as forwards and avoids such write back bus cycles. Having two entries requires two cache cycles to set them up. This is not bad compared with doing a *memory* cycle, and the complexity of the cache controller is probably comparable. Choosing between the one entry and two entry approaches intelligently would require analyzing more details of typical program behavior and implementation costs.

However, we observe that the implementation of the two entry scheme can be improved as follows. First, for LT instructions we can guess that the item will not be modified and not allocate two entries, and force ST to do so if required. This requires adding a transactional cache state to indicate that the entry is transactional (i.e., not invalid or normal) but should be discarded on neither commit nor abort. When ST finds there is no TC_ABORT entry, it fixes things up (taking two cycles, one to fix the old entry and one to create the new one). On the other hand, LTX would create two entries, but the processor could proceed immediately after a cache hit, with the entry creation happening in the transactional cache controller. This might very well "hide" the extra cycle most of the time. ST could be similarly improved, but it may be not worth it since an ST would normally be preceded by an LTX to the same location.

Turning to a different topic, for programs to be written in a uniform and portable manner, one needs to guarantee at the instruction set architecture level the minimum transaction size that the architecture supports. At present we do not have a good feel for what such a size might be, but it should probably be between 10 and 100. Since one might not want to put a fully associative cache of this size into every implementation of the architecture, schemes that use some hardware but handle larger transactions via software traps seem to be desirable. In fact, one can avoid hard limits on transaction size by offering the software overflow mechanism with all implementations.

We have considered how to implemented transactional memory with software overflow. Here are the tasks one must accomplish: support the machine's transactional primitives; support their interaction with the non-transactional instructions (however one chooses to define those semantics); support the bus/network interactions with other caches. The main observation we make is that supporting all of these operations without too much slowdown requires some hardware support to filter out actions that definitely do not involve the transactional items being maintain by software. One way to do this is to have a vector of bits $b[0], ..., b[k-1]$ where each $b[i]$ is associated with a distinct set of addresses, and $b[i]$ is 1 if the software has any entries from the corresponding set, and is 0 otherwise. In particular, when doing a cycle, if the regular and transactional caches miss, then one calculates $i$ from the address (probably by simply extracting some subset of bits from the address) and tests $b[i]$ to determine whether or not to trap. The number of the $b[i]$ and the hash function used determine the number of traps, and in particular, the number of traps that are ultimately useless because the item is not actually present.

By aborting more transactions, we can avoid many of the traps. In particular, if the software store is not allowed to hold on to items after commit, and does not hold dirty TC_COMMIT items, then none of the items maintained in the software are needed to respond to external requests. Similarly, if we rule out non-transactional actions on transactional items, we can abort when such actions occur and need not probe the software store. Thus, only local transactional cycles need to probe the software entries. In all other cases, if we miss in both hardware caches, and the $b[i]$ bit is set, we abort. For this approach to work, we need to be able to vary the hash function so that normal loads that lie in the same $b[i]$ set as a transactional item will not get us into cyclic restart. We suggest that if the hash function is defined as selecting some subset of the bits of the address, say $m$ out of $n$ bits, we provide $m$ multiplexor circuits, each capable of selecting 1 of the $n$ bits in an address, and allow the software to change the control on each of these multiplexors independently. This allows *any* $m$ of the $n$ bits to be selected. The $m$ bits then select one of the $b[i]$ (of which there are $2^m$).

One additional mechanism is required: the processor must be able to freeze transactional cache item ownership while doing a commit, since the software will have to write out one item at a time from its store, unlike the single cycle commit we can do in hardware. Thus external requests will have to be put off, by indicating the processor is busy. This already fits into the network design, but the shared bus design would have to be adjusted a bit.

We note that transactional memory should fit well with superscalar processor designs, since it makes ordering requirements clear. Among other things, transactions do not need to be resolved until the result of a commit or validate is actually used, just as loads do not have to complete until their result is required by a later instruction.

On a different topic, there are circumstances in which one would like to drop an item from the current transaction's read set. For example, if we are chaining down a linked list looking for a particular item with the intent of unchaining it, we need only maintain at most two of the list pointers in the read set at a time, and to reduce contention as well as avoid transactional cache overflow, we should release ownership of earlier items in the list. This requires adding a new instruction, which will revert a single item, and which causes an abort if the item was modified by the transaction. One could relax that restriction as well if there are interesting uses for early and unconditional commit of some changes, but it seems dangerous unless one simultaneously validates the current transaction.

Finally, we observe that we could reduce the need for VALIDATE instructions if we guarantee 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.

## 7 Simulation Results

To gain some insight into the performance of transactional memory, we modified a copy of the Proteus simulator [Brewer *et al.*, 1991] 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 trap to the simulator, and other instructions are executed directly, augmented by cycle-counting code inserted by a preprocessor. Because

```
void Sem_P(Word *s)             /* 0 => busy, 1 => free */
{
  unsigned backoff = BACKOFF_MIN, wait;
  while (1) {
    if ((int) *s)
      if ((int) Test_and_Set(s))
        break;
    wait = random() % (01 << backoff);
    while (wait--);
    if (backoff < BACKOFF_MAX) backoff++;
  }
}
```

Figure 21: Spin Lock with Backoff

```
shared int counter;

void process(int work)
{
  int success = 0, backoff = BACKOFF_MIN, 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 22: Counting Benchmark

most of the program is executed directly by the host processor, large simulations can be run relatively quickly. Proteus does not capture the effects of instruction caches or local caches.

We implemented both the extended version of Goodman's snoopy protocol, on a 32-processor bus-based architecture, and the extended version of the Chaiken directory protocol, on a 32-processor (simulated) Alewife [Agarwal *et al.*, 1991]. In both simulations, the regular cache is a direct-mapped cache with 2048 lines of size 8 bytes, and the transactional cache has 64 8-byte lines.

We constructed three simple benchmarks, and ran them with four different synchronization mechanisms: (1) transactional memory with user-level exponential backoff, (2) test-and-test-and-set (TTS) [Rudolph, 1983] spin locks with exponential backoff [Anderson, 1990; Metcalfe and Boggs, 1976] (Figure 21), (3) LOAD_LINKED/STORE_COND (LL/SC) spin locks with exponential backoff, and (4) queue locks [Anderson, 1990; Graunke and Thakkar, 1990; Mellor-Crummey and Scott, 1991]. For a single-word counter benchmark, we also ran a LL/SC implementation directly on the shared variable. All the locking implementations 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 the three locking 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 [Anderson, 1990], 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* [Rudolph, 1983] 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 [Anderson, 1990; Mellor-Crummey and Scott, 1991]. 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 R4000 [Kane, 1989] and Digital's Alpha [Digital Equipment Corporation]. On a cache-coherent architecture, these operations are implemented as single-word transactions — a SC succeeds if the processor retains exclusive ownership of the entry read by the LL.

In a *queue lock*, each process atomically increments a shared variable, and uses the resulting value as an index into a boolean array. The process waits until the indexed value in the array becomes *True*, and then enters the critical section. When it leaves the critical section, it "informs" the process with the next higher index by setting its array value to *True*. Variations of queue locks have been proposed by Anderson [Anderson, 1990], by Mellor-Crummey and Scott [Mellor-Crummey and Scott, 1991], and by Graunke and Thakkar [Graunke and Thakkar, 1990]. This technique produces low memory contention, since processes spin on different cached locations, and has been observed to perform well on small-scale bus-based architectures. In our network implementation of this technique, we found that it was necessary

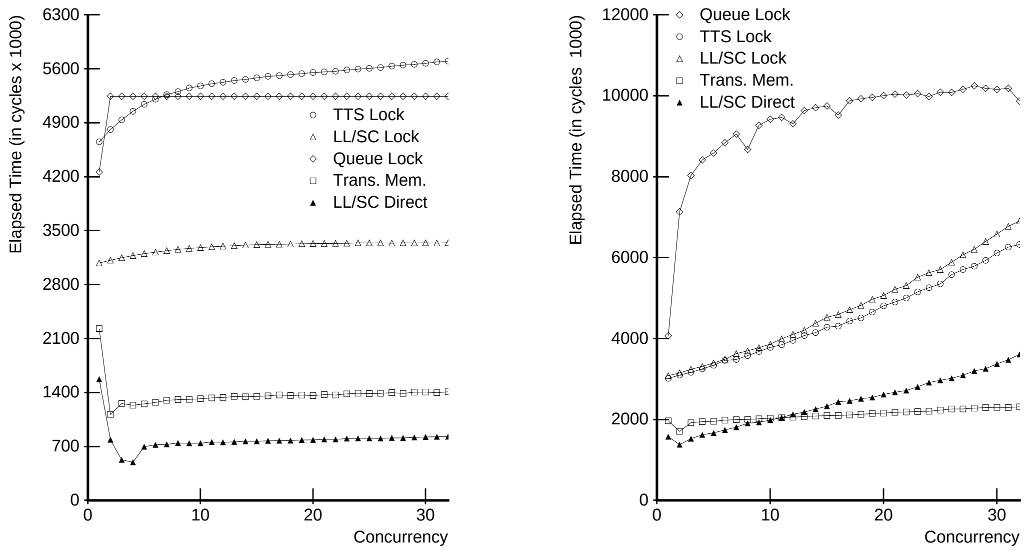

Figure 23: Counting Benchmark: Bus and Network

to scatter the array elements among the participating processors to reduce network congestion.

## 7.1 Counting Benchmark

In our first benchmark (code in Figure 22), 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 23, the vertical axis shows the number of cycles needed to complete the benchmark, and the horizontal axis shows the number of concurrent processes. Transactional memory has substantially higher throughput than any of the locking implementations, 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). The LL/SC and queue lock implementations are similar. Transactional memory, on the other hand, 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. Direct LL/SC performs well here because no commit operation is necessary. Notice, however, that the only way we can use LL/SC for the other benchmarks is as a spin lock, since they use shared data structures

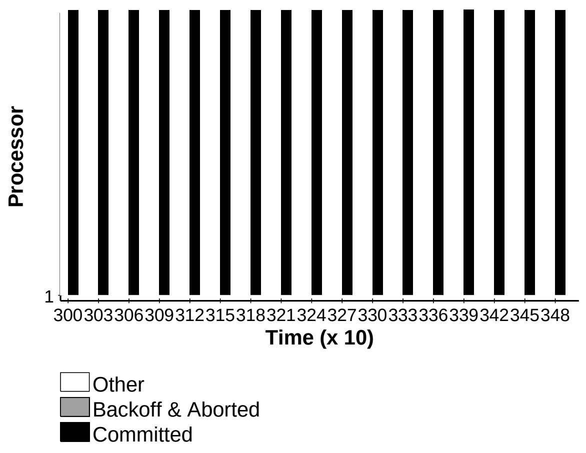

Figure 24: Single-Processor Time Line: Transactional Memory

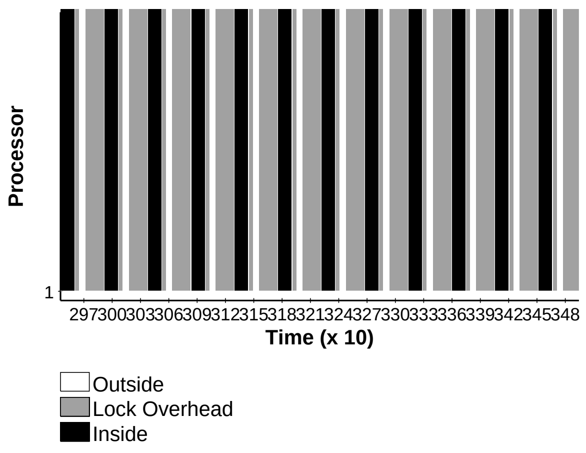

Figure 25: Single-Processor Time Line: TTS Spin Lock

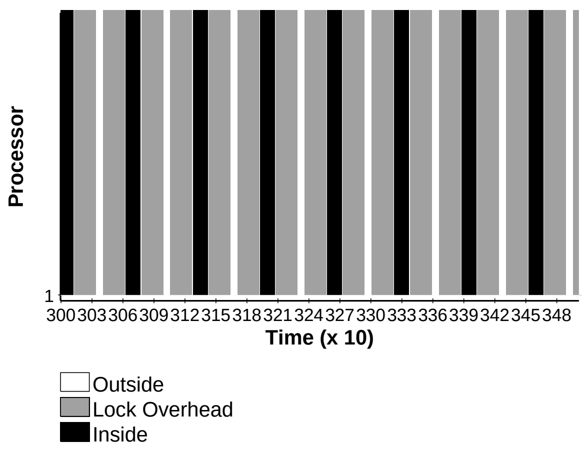

Figure 26: Single-Processor Time Line: Queue Lock

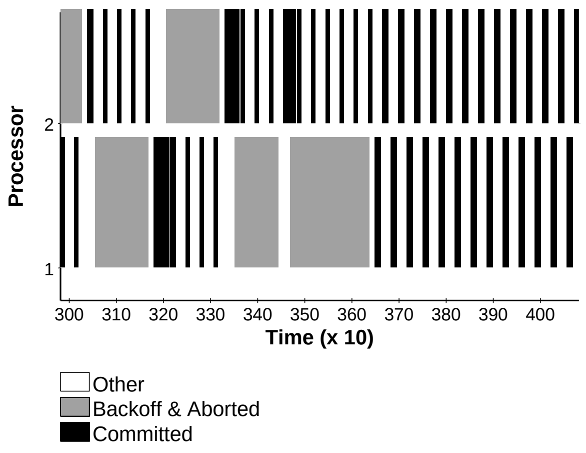

Figure 27: Two-Processor Time Line: Transactional Memory

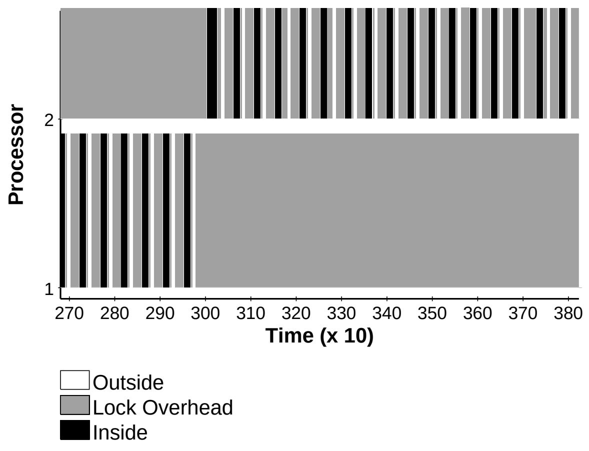

Figure 28: Two-Processor Time Line: TTS Spin Lock

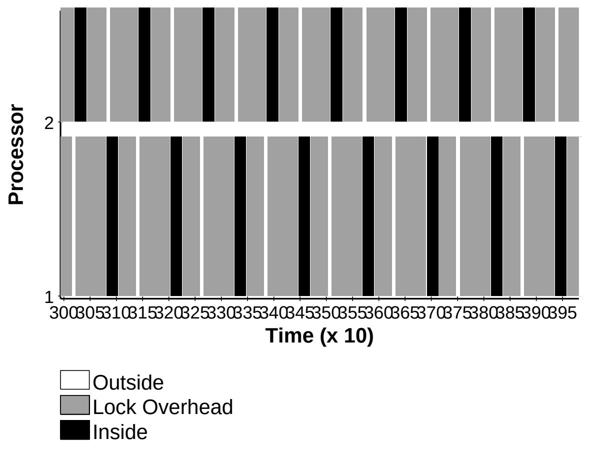

Figure 29: Two-Processor Time Line: Queue Lock

that occupy multiple words of memory. These results suggest that it might be effective to provide a single operation that atomically performs a transactional store and attempts to commit the current transaction.

As an aside, we note that the LL/SC spin lock substantially outperforms the TTS spin lock in the bus architecture, but not in the network architecture. This difference is an artifact of Goodman's snoopy cache protocol. The first time a location is modified, Goodman's protocol marks it *reserved* and writes it back. This technique works well for some variables, but it is poorly suited for spin locks, since they are about to be written again (when they are released). Both the LL/SC protocol and Chaiken's directory protocol avoid this inefficiency, leaving the lock variable dirty in the cache.

The behaviors of the different schemes are illustrated in Figures 24, 25, and 26, which show fragments of the time lines for the single-processor counting benchmarks for transactional memory, TTS spin lock, and queue lock on the bus architecture. Each time line shows the durations of three kinds of activities: (1) useful work: committed transactions or critical sections, (2) overhead: aborted transactions, lock management, or backoff, (3) other: primarily loop indexing. These figures show that the transactional memory implementation, which does no lock management, is able to pack more committed transactions into an interval than the locking schemes, which have a substantial synchronization overhead.

Figures 27, 28, and 29 show time line fragments for the two-processor counting benchmark. Although we now see the effects of transaction aborts and backoffs, the transaction length is short enough that the transactional memory implementation can still sometimes overlap loop overhead with transactions (see the right-hand side of Figure 27). In the TTS and queue lock implementations, such an overlap is impossible because of the lock management overhead.

## 7.2 Producer/Consumer Benchmark

In the *producer/consumer* benchmark (code in Figure 30), $n$ 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 31), 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.

## 7.3 Doubly-Linked List Benchmark

In the *doubly-linked list* (code in Figure 32), $n$ processes share a doubly-linked list anchored by *head* and *tail* pointers. Each process dequeues an item by removing the item pointed to by *head*, and then enqueues it by threading it onto the list at *tail*. A process that removes the last item sets both *head* and *tail* to *NULL*, and a process that inserts an item into an empty list sets both *head* and *tail* to point to the new item. The benchmark finishes when $2^{16}$ 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

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

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

Figure 30: Part of Producer/Consumer Benchmark

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 *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 33. The locking implementations have substantially lower throughput, primarily because they because never allow enqueues and dequeues to overlap.

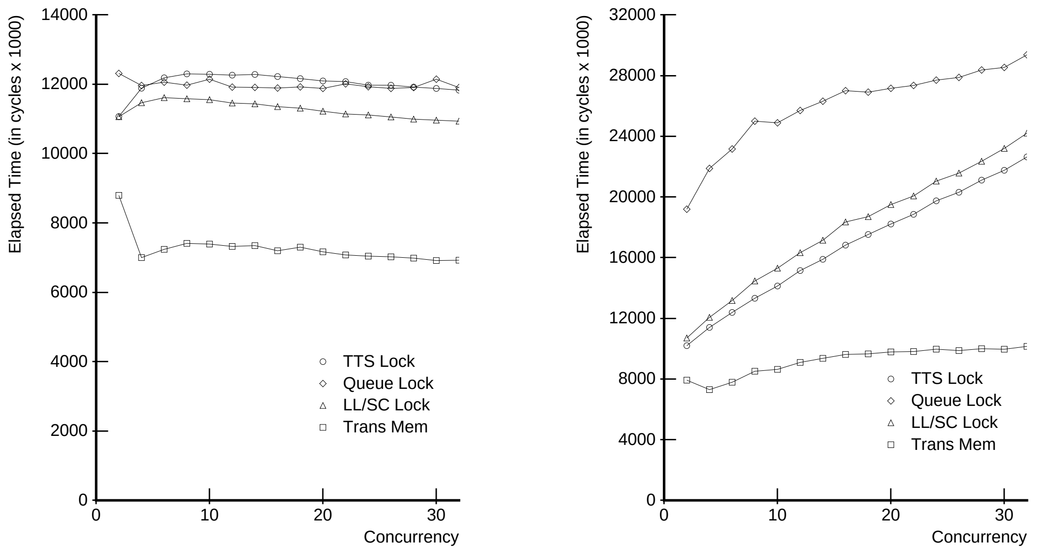

Figure 31: Producer/Consumer Benchmark: Bus and Network

## 8 Discussion and Related Work

Transactional memory is a direct generalization of the load-linked (LL) and store-conditional (SC) instructions proposed by Jensen et al. [Jensen *et al.*, 1987], and since incorporated into the MIPS R4000 [Kane, 1989] and Digital's Alpha [Digital Equipment Corporation]. The LL instruction is essentially the same as LTX, and SC is a combination of ST and COMMIT. The LL/SC 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 atomic operations on multiple locations. For example, the Motorola 68000 provides a COMPARE&SWAP2 operation that operates on two independent locations. This instruction was used by Massalin and Pu [Massalin and Pu, 1991] for lock-free list manipulation in an operating system kernel. Transactional memory provides more powerful support for this "lock-free" style of programming.

The literature includes a variety of memory consistency models, including sequential consistency [Lamport, 1979], processor consistency [Goodman, 1989], weak consistency [Dubois *et al.*, 1986; Dubois and Scheurich, 1990], and release consistency [Gharachorloo *et al.*, 1990][3]. Of these, transactional memory most closely resembles release consistency. LT, LTX, and ST are similar to ACQUIRE, and COMMIT is similar to RELEASE. Perhaps the most significant difference is that transactions can abort, and do so consistently and cleanly. We distinguish transactional and non-transactional memory operations, and private memory

[3]See Gharachorloo et al. [Gharachorloo *et al.*, 1991] for concise descriptions of these models as well as performance comparisons.

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

shared entry *Head, *Tail;

void list_enq(entry* new) {

  entry *old_tail;
  unsigned backoff = BACKOFF_MIN, 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 32: Part of Doubly-Linked List Benchmark

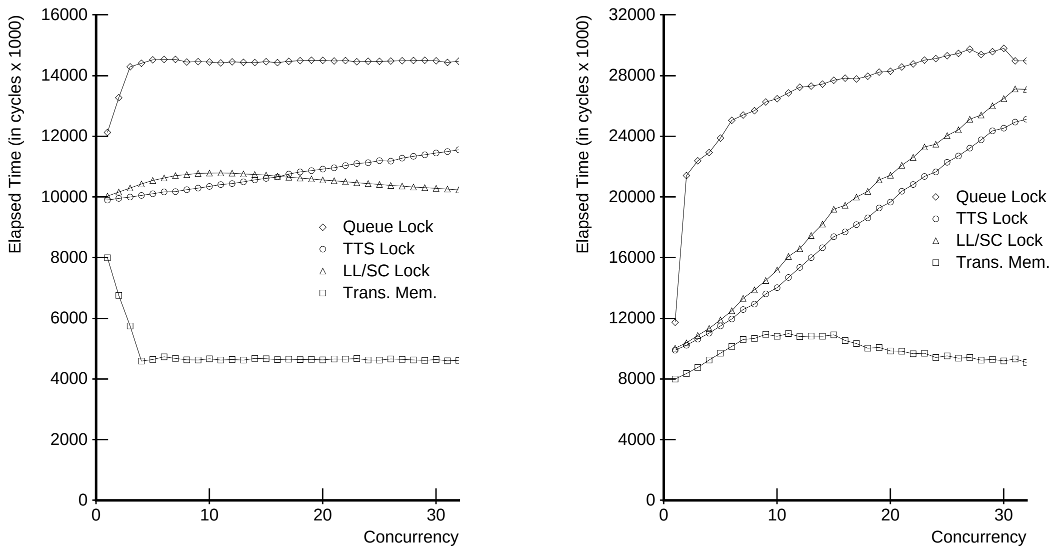

Figure 33: Doubly-Linked List Benchmark: Bus and Network

accesses are not affected by transactional operations. Also, the transactional memory primitives include synchronization, while ACQUIRE and RELEASE are primarily intended for use within critical sections.

Like the models cited above, transactional memory imposes certain constraints on instruction issue and reordering. For example, LT, LTX, ST, LOAD, and STORE instructions referring to different locations may be reordered freely, but LT and LTX instructions may not be reordered with respect to successful VALIDATE instructions, and LT, LTX, and ST may not be reordered with respect to successful COMMIT instructions. While transactional memory obviates certain common uses of memory access barrier instructions, barriers are still useful for non-transactional data. For example, one might construct a list element using non-transactional operations, and thread it onto a list using transactional operations. The element's initialization must take effect before that transaction commits, suggesting that COMMIT and VALIDATE should perhaps implicitly perform barriers.

Other researchers who have investigated architectural support for multi-word synchronization include Knight [Knight, 1986], who suggests using cache consistency protocols to add parallelism to "mostly functional" LISP programs, and the IBM 801 [Chang and Mergen, 1988], 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 [Anderson, 1990], Bershad [Bershad, 1991], Graunke and Thakkar

[Graunke and Thakkar, 1990], and Mellor-Crummey and Scott [Mellor-Crummey and Scott, 1991].

## References

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

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

[Bershad, 1991] B.N. Bershad. Practical considerations for lock-free concurrent objects. Tech. Rep. CMU-CS-91-183, Carnegie Mellon University, Pittsburgh, PA, Sept. 1991.

[Brewer *et al.*, 1991] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel architecture simulator. Tech. Rep. MIT/LCS/TR-516, Massachusetts Institute of Technology, Sept. 1991.

[Chaiken, 1990] D.L. Chaiken. Cache coherence protocols for large-scale multiprocessors. Tech. Rep. MIT/LCS/TR-489, MIT LCS, 545 Tech Square, Cambridge MA 02139, Sept. 1990.

[Chaiken *et al.*, 1991] 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* (Apr. 1991), ACM, pp. 224–234.

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

[Digital Equipment Corporation] Digital Equipment Corporation. Alpha system reference manual.

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

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

[Gharachorloo *et al.*, 1991] 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* (Apr. 1991), pp. 245–257.

[Gharachorloo *et al.*, 1990] 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* (June 1990), pp. 15–26.

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

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

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

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

[Herlihy, 1990] 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* (Mar. 1990), pp. 197–206.

[Jensen *et al.*, 1987] E.H. Jensen, G.W. Hagensen, and J.M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Tech. Rep. UCRL-97663, Lawrence Livermore National Laboratory, Nov. 1987.

[Jouppi, 1990] 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* (June 1990), ACM SIGARCH, p. 364.

[Kane, 1989] G. Kane. *MIPS RISC Architecture*. Prentice Hall, New York, 1989.

[Knight, 1986] T. Knight. An achitecture for mostly functional languages. In *Conference on Lisp and Functional Programming* (Aug. 1986), pp. 105–112.

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

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

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

[Mellor-Crummey and Scott, 1991] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (Feb. 1991), 21–65.

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

[Rudolph, 1983] L. Rudolph. Decentralized cache scheme for an MIMD parallel processor. In *11th Annual Computing Architecture Conference* (1983), pp. 340–347.

[Wing and Gong, 1990] J. Wing and C. Gong. A library of concurrent objects and their proofs of correctness. Tech. Rep. CMU-CS-90-151, Carnegie Mellon University, Pittsburgh, PA, July 1990.