Optimizing Memory Transactions

Tim Harris[†] Mark Plesko[†] Avraham Shinnar[‡] David Tarditi[†]

Microsoft Research[†] Harvard University[‡]

tharris@microsoft.com markples@microsoft.com shinnar@eecs.harvard.edu dtarditi@microsoft.com

Abstract

Atomic blocks allow programmers to delimit sections of code as 'atomic', leaving the language's implementation to enforce atomicity. Existing work has shown how to implement atomic blocks over *word-based transactional memory* that provides scalable multiprocessor performance without requiring changes to the basic structure of objects in the heap. However, these implementations perform poorly because they interpose on all accesses to shared memory in the atomic block, redirecting updates to a thread-private log which must be searched by reads in the block and later reconciled with the heap when leaving the block.

This paper takes a four-pronged approach to improving performance: (1) we introduce a new 'direct access' implementation that avoids searching thread-private logs, (2) we develop compiler optimizations to reduce the amount of logging (e.g. when a thread accesses the same data repeatedly in an atomic block), (3) we use runtime filtering to detect duplicate log entries that are missed statically, and (4) we present a series of GC-time techniques to compact the logs generated by long-running atomic blocks.

Our implementation supports short-running scalable concurrent benchmarks with less than 50% overhead over a non-thread-safe baseline. We support long atomic blocks containing millions of shared memory accesses with a 2.5-4.5x slowdown.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Algorithms, Languages, Performance

Keywords Atomicity, Critical Regions, Transactional Memory

1. Introduction

Atomic blocks provide a promising simplification to the problem of writing concurrent programs [12]. A code block is marked `atomic` and the compiler and runtime system ensure that operations within the block, including function calls, appear atomic. The programmer no longer needs to worry about manual locking, low-level race conditions, or deadlocks. Atomic blocks can also provide *exception recovery*, whereby a block's side effects are rolled back if an exception terminates it [13]. This is valuable even in a single-threaded application: error handling code is often difficult to write and to test [29]. Implementations of atomic blocks scale to large multiprocessor machines [12] because they are *parallelism preserving*: atomic blocks can execute concurrently so long as a location being updated in one block is not being accessed in any of the others. This preserves the kind of sharing allowed in a conventional data cache.

Although they scale well, current implementations of atomic blocks introduce substantial runtime overhead [12]. They are built using *word-based software transactional memory* (STM) which allows a series of memory accesses made via the STM library to be performed atomically. There are three main reasons for the runtime overhead, which we discuss in more detail in Section 2:

- **STM implementations typically create private shadow copies of memory updated in atomic blocks.** This introduces lookups on all read operations in atomic blocks and slows down write operations when there is no contention. Furthermore, the cost of these lookups precludes the use of atomic blocks across longer-running sections of code.
- **STM is implemented as a library.** Calls to STM operations are introduced late in compilation and are treated as opaque calls. This misses many optimization opportunities.
- **STM operations are used unnecessarily.** Accesses to heap data are blindly redirected through the STM without consideration of whether or not an object is thread-local.

We address these problems with a novel STM implementation that is more tightly integrated with the compiler and runtime system. We make a number of contributions:

- **Direct-access STM**. Our STM is the first to allow objects to be updated directly in the heap rather than working on private shadow copies of objects, or via extra levels of indirection between an object reference and the current object contents. This optimizes for transactions that commit successfully.
- **Decomposed STM interface**. Section 3 describes how we decompose transactional memory operations to expose opportunities for classical optimizations. For instance, a transactional store `obj.field = x` is split into steps that (a) record that `obj` is being updated by the current thread, (b) log the old value that `field` held, and (c) store the new value `x` into the field. These three steps are then handled separately by the compiler and (a) and (b) can often be hoisted from a loop while (c) cannot.
- **Compile-time optimizations.** Section 4 describes additional optimizations to reduce the number of calls to the STM interface. For instance, by further decomposing the logging operations we can amortize the cost of checking for space across a series of stores into the log.
- **Integrated transactional versioning.** Our STM is the first to integrate transactional versioning with an existing object header word. Earlier STMs, even those integrated in a managed runtime environment, either used external tables of versioning records [12], additional header words [13], or made

PLDI'06 June 10–16, 2006, Ottawa, Ontario, Canada

programmer-visible changes to the object model to add levels of indirection between object references and current object contents [16, 10, 23].

- **Runtime filtering.** Not all unnecessary operations can be identified statically, so we add complementary runtime filtering – e.g. to remove updates to transaction-local objects. (Section 5).
- **Garbage collection (GC) integration.** Our implementation is the first to allow the GC to reclaim objects that become unreachable within a still-running transaction; earlier work would hold onto such objects until the transaction that allocated them has committed or aborted. (Section 6).

Our work is implemented in Bartok, an optimizing ahead-of-time research compiler and runtime system for Common Intermediate Language (CIL) programs with performance competitive to the Microsoft .NET Platform. The runtime system is implemented in CIL, including the garbage collectors and the new STM.

As our results in Section 7 show, the combined effect of our techniques is that short blocks (e.g. updates to red-black trees or skip lists) run with less than 50% overhead compared with non-thread-safe uniprocessor code. Furthermore, our techniques allow blocks to scale to contain millions of memory accesses, running between 2.5x and 4.5x slower than uniprocessor code.

As we conclude in Section 9, our work sheds light on how future hardware can improve the performance of atomic blocks. It is important that the problem be tackled after exploring the opportunities for optimizing a purely software-based implementation and with careful consideration of how hardware support fits with all of the other parts of the runtime system.

1.1 Semantics

This paper focuses on the performance of atomic blocks. There are many interesting questions about their exact semantics [11], including the interaction of atomic blocks with locking code and combining I/O operations with atomic blocks. These are important questions but orthogonal to the performance questions we consider.

1.2 Design assumptions

As with any performance-based work on language design, we are faced with a chicken-and-egg problem in terms of benchmarking. We therefore make some assumptions about how `atomic` blocks will be used.

Our key assumption is that *most transactions commit successfully*. We believe this is reasonable. First, the use of a parallelism-preserving STM means that transactions will not abort 'spontaneously' or because of conflicts that the programmer cannot understand (in an earlier system we built, conflicts were detected based on hash values, which could collide giving unintuitive performance characteristics [12]). Second, the programmer already has a strong incentive to avoid contention because of the cost of ping-ponging data between caches. Traditional techniques such as handing high-contention operations off to work queues managed by a single thread remain valuable.

Our second assumption is that *reads significantly outnumber updates* in atomic blocks. This is borne out by observations of current programs, and attempts to develop transactional versions of them [5, 3]. This makes us careful to keep the overhead of transactional reads low: reads involve merely logging the address of the object being read and the contents of its header word.

Our final assumption is that *transaction size cannot be bounded*. This lets us retain compositionality and suggests that the STM implementation needs to scale well as the length of transactions grows. In our design, the space overhead grows with the number of objects that a transaction accesses and the number of words that it updates. It does not grow with the number of accesses made.

In this paper, we distinguish informally between transactions that are *short* and transactions that are *long*. A *short* transaction is likely to run without requiring any memory allocation by our STM, meaning that it can access up to 1024 words in our experiments. Short transactions are also likely to be supported by traditional hardware transactional memory designs [17]. In contrast, when we refer to *long* transactions, we mean those which are likely to require memory allocation within an STM, and which are unlikely to be accommodated in hardware without complicated extensions [26].

2. Atomic blocks and STM

In this section we introduce the conventional interface for word-based transactional memory. We show how `atomic` blocks are built over it and highlight the problems of that approach. Much of the discussion applies equally to object-based STMs [16, 15, 23] where many of the same fundamental problems occur.

2.1 Word-based transactional memory

Word-based STM provides the following two sets of operations [12]:

```
void TMStart()
void TMAbort()
bool TMCommit()
bool TMIsValid()

word TMRead(addr a)
void TMWrite(addr a, word value)
```

The first set is used to manage transactions: `TMStart` starts a transaction in the current thread. `TMAbort` aborts the current thread's transaction. `TMCommit` attempts to commit the current thread's transaction. If the transaction cannot commit (because a concurrent transaction has updated one of the locations it accessed) then `TMCommit` returns `false` and the current transaction is discarded. Otherwise, `TMCommit` returns `true` and the updates are atomically propagated to the shared heap. `TMIsValid` returns `true` iff the current thread's transaction could commit at the point of the call. The second set of operations performs data accesses: `TMRead` returns the current value of the specified location, or the most recent value written by `TMWrite` in the current transaction.

2.2 Building atomic blocks over STM

Programming directly with STM is cumbersome because the programmer must ensure that `TMRead` and `TMWrite` are used for all memory accesses made during a transaction. It is straightforward to automate this process by having a compiler rewrite memory accesses in atomic blocks to use STM operations, and having it generate specialized versions of any methods called.

2.3 Problems with the STM interface

This design suffers from a number of problems which limit its applicability. Figure 1(a) shows a running example that illustrates this. The example iterates through the elements of a linked list between sentinel nodes `this.Head` and `this.Tail`. It sums the `Value` fields of the nodes and stores the result in `this.Sum`. Figure 1(b) shows how `Sum` could be implemented using traditional word-based STM operations. Several performance problems occur:

- **Searching transaction logs will not scale to support large transactions**. `TMRead` must see earlier stores by the same transaction, so it must search the transaction log that holds tentative updates. The performance depends on the length of the transaction log and the effectiveness of auxiliary index structures.
- **Opaque calls to the TM library hinder optimization** – e.g. it is no longer possible to hoist reading `this.Tail` from the loop.
- **Monolithic TM operations cause repeated work**. For instance, repeated searches when accessing a field in a loop.

```
public int Sum() {
 Node n = this.Head;
 int t = 0;
 do {
  t += n.Value;
  if (n==this.Tail)
  {
   this.Sum = t;
   return t;
  }
  n = n.Next;
 }
}
```

(a) Original code.

```
public void Sum() {
 Node n = TMRead(&this.Head);
 int t = 0;
 do {
  t += TMRead(&n.Value);
  if (n==TMRead(&this.Tail))
  {
   TMWrite(&this.Sum, t);
   return t;
  }
  n = TMRead(&n.Next);
 }
}
```

(b) Monolithic operations.

```
public int Sum() {
 tm_mgr tx = DTMGetTMMgr();
 DTMOpenForRead(tx, this);
 Node n = this.Head;
 int t = 0;
 do {
  DTMOpenForRead(tx, n);
  t += n.Value;
  DTMOpenForRead(tx, this);
  if (n==this.Tail) {
   DTMOpenForUpdate(tx, this);
   DTMLogFieldStore(tx, this,
          offsetof(List.Sum));
   this.Sum = t;
   return t;
  }
  DTMOpenForRead(tx, n);
  n = n.Next;
 }
}
```

(c) Decomposed operations.

```
public int Sum() {
 tm_mgr tx = DTMGetTMMgr();
 DTMOpenForUpdate(tx, this);
 Node n = this.Head;
 int t = 0;
 do {
  DTMOpenForRead(tx, n);
  t += n.Value;
  if (n==this.Tail) {
    DTMLogFieldStore(tx, this,
           offsetof(List.Sum));
    this.Sum = t;
    return t;
  }
  n = n.Next;
 }
}
```

(d) Optimized operations.

Figure 1. Running example with explicit STM calls (in reality, these are added during compilation, not as a source-to-source transformation).

3. Decomposed direct-access STM

This section introduces a new interface that lets us solve the problems with the monolithic word-based STM.

The first problem is solved by designing the system so that a transaction can perform read and write operations directly to the heap, letting a read naturally see a preceding transactional store without any searching. Of course, logs are still needed for rolling back a transaction that aborts and to detect conflicts at commit time. However, for short transactions, these logs are append-only, and searching is never required for any transaction size.

The second problem is solved by introducing TM operations early during compilation and extending the subsequent analysis and optimization phases to be aware of their semantics.

Finally, the third problem is solved by decomposing the monolithic TM operations into separate steps so that repeated work can be avoided. For instance, we separate the management of the transaction logs from the actual data accesses, often allowing log management to be hoisted from loops.

The result is a new form of STM interface which can be seen as a hybrid that combines ideas from pure word-based and object-based designs. As with object-based STM, objects must be *opened* by a transaction before they can be accessed. However, as with a word-based STM, subsequent accesses are performed with reference to an ordinary memory address rather than with reference to a handle returned when the object was opened. Avoiding the use of handles reduces the number of live variables at most points in a transaction's execution.

The new interface decomposes the transactional memory operations into four sets:

```
tm_mgr DTMGetTMMgr()

void DTMStart(tm_mgr tx)
void DTMAbort(tm_mgr tx)
bool DTMCommit(tm_mgr tx)
bool DTMIsValid(tm_mgr tx)

void DTMOpenForRead(tm_mgr tx, object obj)
void DTMOpenForUpdate(tm_mgr tx, object obj)
object DTMAddrToSurrogate(tm_mgr tx, addr a)

void DTMLogFieldStore(tm_mgr tx, object obj, int offset)
void DTMLogAddrStore(tm_mgr tx, addr a)
```

The first two sets are straightforward, providing `DTMGetTMMgr` to get the current thread's *transaction manager* and the usual transaction management operations. Each thread has its own transaction manager that survives for the lifetime of the thread. Making the transaction manager an explicit parameter of the `DTM*` operations allows us to reduce the number of accesses to per-thread storage.

The third set provides contention detection. Most field accesses are performed with respect to an object reference. These cases are handled directly by `DTMOpenForRead` and `DTMOpenForUpdate` which indicate that the specified object will be accessed in read-only mode or that it may subsequently be updated. Update access subsumes read access so it is sufficient to open an object for update before a series of reads and writes to its fields. Static field accesses and indirect field accesses do not ordinarily involve an object reference: these cases are handled by `DTMAddrToSurrogate` which maps an address to a *surrogate object* that is used for contention detection on behalf of the address[1].

The final set of operations maintains an undo log, needed to roll back updates on abort. `DTMLogFieldStore` deals with stores to objects and `DTMLogAddrStore` deals with stores to any address.

Calls to these operations must be correctly sequenced to provide atomicity. There are three rules: (a) a location must be open for read (or for update) when it is read, (b) a location must be open for update when it is updated or a store logged for it, (c) a location's old value must have been logged before it is updated. In practice this means that a call to `TMRead` could be rewritten as `DTMGetTMMgr`, `DTMAddrToSurrogate` and then `DTMOpenForRead`. `TMWrite` is `DTMGetTMMgr`, `DTMAddrToSurrogate`, `DTMOpenForUpdate` and then `DTMLogAddrStore`.

Figure 1(c) shows how our running example can be written using this decomposed interface and Figure 1(d) illustrates the optimization opportunities that are available.

3.1 Runtime system

In this section we describe the implementation of the decomposed direct-access STM. In overview, a transaction uses strict two-phase locking for updates, and it records version numbers for objects that it reads from so it can detect conflicting updates. A roll-back log is used for recovery upon conflict or deadlock.

The use of pessimistic update locking is motivated by our workload assumption that conflicts are rare: locking enables the owning thread to update objects in place. The use of optimistic concurrency control on reads is motivated by our goal to offer scalable performance: all of the cache lines holding read-only data can remain in

[1] For an indirect field access this means converting an interior pointer into a reference to the containing object.

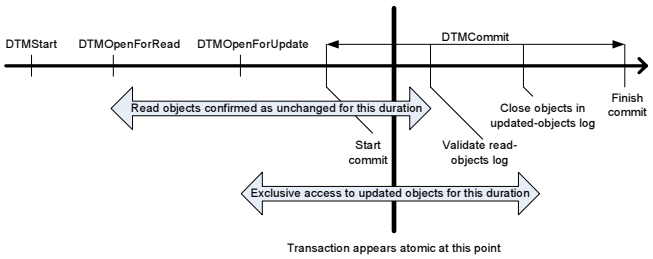


Figure 2. Ensuring atomicity: the commit operation checks that objects read are unchanged during the upper gray arrow. Object updated are held under exclusive access during the lower arrow.

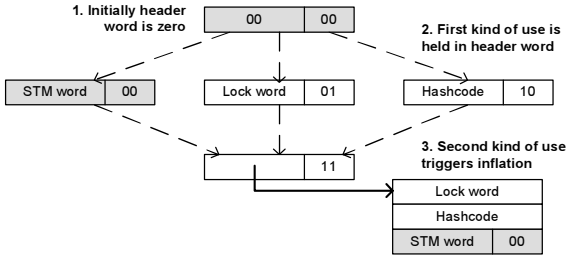


Figure 3. Multi-use word states: the STM word is held explicitly where shaded and is implicitly 0 in objects that have not yet been opened for update in a transaction.

shared mode in all of the readers' data caches. This would not be possible with a traditional lock because even an MRSW variant will use atomic read-modify-write operations for synchronization.

Note that this combination of forms of concurrency control means that update locks must be acquired on objects that a thread allocates within a transaction: this ensures that a second thread encountering a reference to such an object in shared memory is aware that the first thread has exclusive update access to it. However, as we discuss in Sections 4.2 and 5.1, we can streamline the way that log entries are managed for this kind of transactionally-allocated object.

Figure 2 illustrates the operations performed by a simple transaction: it calls `DTMStart`, then opens objects for reading and for update, and it concludes by calling `DTMCommit` to attempt to perform those accesses atomically. Each call to `DTMOpenForRead` records a version number for the object in question. Each call to `DTMOpenForUpdate` acquires an update lock on the object.

Internally, the commit operation begins by attempting to *validate* the objects that have been opened for reading by checking that the recorded version numbers are still current. This ensures that no updates have been made to them by other transactions since they were opened. If validation fails then a conflict has been detected: the transaction's updates are rolled back and the objects it opened for update are *closed*, whereupon they can be opened by other transactions. If validation succeeds then the transaction has executed without conflicts: the objects that it opened for update are closed, retaining the updates.

Validation checks that there were no conflicting updates to the objects that the transaction read during the timespan indicated by the upper gray arrow on Figure 2. Holding locks on objects open for update prevents conflicts during the timespan of the lower gray arrow. Consequently, *there was no conflicting access to any of the objects opened during the intersection of these timespans*; the transaction appears to take place atomically in this interval. This *linearizability* [18] argument is common in transactional memory systems [10].

We present the details of this implementation in three sections. We show how we extend the objects' structure to support the version numbers and locks used by our STM. We then show how we implement the `DTMOpen*` and `DTMLog*` operations. Finally, we present the `DTMCommit` operation.

3.1.1 Object structure

We now turn to the structures used to support the validation of read-only objects and the open and close operations on objects that are updated. The STM requires two abstract entities on each object: an *STM word*, used to coordinate which transaction has the object open for update, and an *STM snapshot*, used in fast-path code to detect conflicting updates to objects the transaction has read:

```
word GetSTMWord(Object o)
bool OpenSTMWord(Object o, word prev, word next)
void CloseSTMWord(Object o, word next)

snapshot GetSTMSnapshot(Object o)
word SnapshotToWord(snapshot s)
```

An object's *STM word* has two fields. The first, a single bit, indicates whether or not the object is currently open for update by any transaction. If set then the remainder of the word identifies the owning transaction. Otherwise the remainder of the word holds a version number. `OpenSTMWord` attempts an atomic compare-and-swap on the STM word (from `prev` to `next`). `CloseSTMWord` updates the word to a specified value.

An object's *STM snapshot* provides a hint about the object's transactional state. The implementation must guarantee that the snapshot changes whenever `CloseSTMWord` is called on the object – that is, whenever a thread releases update-access to the object. As we shall see, this provides sufficient information to detect conflicts.

The Bartok runtime associates a single *multi-use header word* with each object, using this to associate locks and hashcodes with objects. As Figure 3 shows, we extend this design with an additional state to hold the STM word of objects that have ever been opened for update in a transaction. If the multi-use word is needed for more than one of these purposes then it is *inflated* and an external structure holds the object's lock word, hashcode, and STM word.

The STM snapshot is simply the value of the object's multi-use word. Note that this will naturally change when the STM word is stored directly in the multi-use word. If the multi-use word has been inflated then `CloseSTMWord` creates a new inflated structure and copies the contents of the previous structure to it.

The idea of inflating a header word has been widely used to associate locks or hash values with objects [4, 1, 8]. The key novelty of our work is to extend the design to include an STM word while avoiding the need to interrogate the inflated structure in `DTMOpenForRead`.

3.1.2 Transaction log structure

Each thread has a separate transaction manager with three logs. The *read-object log* and *updated-object log* track objects that the transaction has open for read or for update. The *undo log* tracks updates that must be undone on abort. All logs are written sequentially and never searched. We use separate logs because the entries in them have different formats and because, during commit, we need to iterate over entries of different kinds in turn. Each log is organized into a list of arrays of entries, so they can grow without copying.

We illustrate the structure of the logs using the running list example. Figure 4(a) shows the initial state of a list holding a single node with value `10`. We assume that the multi-use header words of

the objects are both being used to hold STM words – in this case the objects are at versions 90 and 100.

The first operation from Figure 1(d) opens `this` for update, using `OpenSTMWord` to atomically replace the version number with a pointer to a new entry in the updated-object log. Figure 5(a) defines this in pseudo-code and Figure 4(b) shows the result[2].

The list-summing example proceeds to open each list node for read. DTM makes this straightforward: for each object we log the object reference and its current STM snapshot. Figure 5(b) shows this in pseudo-code and Figure 4(c) shows the log entry it creates.

We do *not* attempt to detect conflicts when opening an object for reading. This follows the design assumption that contention is rare, so the benefits of discovering it early are outweighed by the cost of checking. One could also imagine attempting to avoid writing duplicate log entries at this point, either by searching the log (as we did in previous work [12]), or by updating the object to record that we have already opened it for reading (as in several object-based transactional memories that use *visible reads* [16, 23]). *We do neither of these*. Searching the log is practical only for short transactions, and updating the object prevents it being cached at multiple processors.

After reading the list nodes, the final step is to update the `Sum` field. `DTMLogFieldStore` records the overwritten value with an entry in the undo log as shown in Figure 4(d). We omit the pseudo-code for this — the particular record we use is influenced by the GC support in Bartok, and other designs will be appropriate in other systems. The `undo` log entry records the address of the overwritten value as an (*object*, *offset*) pair. This avoids using interior pointers, which are expensive to process in some garbage collectors. The entry also distinguishes between scalar or reference-typed stores. This type information is also useful to the GC. Finally, it records the overwritten value. In principle, a shorter two-word log entry could be used that holds just an address and the overwritten word, at the cost of more work during garbage collection.

3.1.3 Commit

There are two phases to DTMCommit: the first checks for conflicting updates to the objects opened for reading and the second *closes* the objects that were opened for update. There is no need to explicitly close objects opened for reading because that fact is recorded only in thread-private transaction logs.

Figure 5(c) shows the structure of `ValidateReadObject`. There are a large number of cases in the code, but the overall design is clearer if you consider them in terms of the operations on the DTM interface:

V1 The object was not open for update at any point in the transaction's duration.

V2 The object was open for update by us for the whole duration.

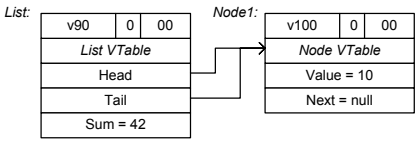
V3 The object was originally not open for update, and we were the next transaction to open it for update.

V4 The object was open for update by another transaction for the whole duration.

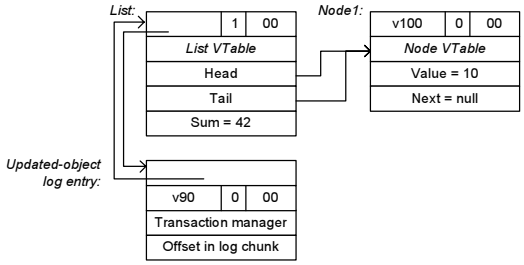
V5 The object was originally not open for update, and another transaction was the next to open it for update.

These cases are marked in the pseudo-code. Some occur multiple times because we must distinguish between occasions where the test made on the STM snapshot fails because of an actual conflict, and where it fails without conflict (e.g. because the STM snapshot changed when the object's multi-use-word became inflated).

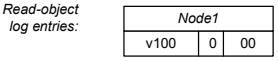
[2] The '*offset in log chunk*' field is used during GC as a fast way to map an interior pointer into the log (such as that from the *List* node in the figure) to a reference to the array of log entries holding it.



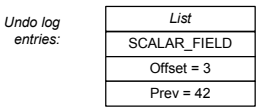
(a) Before the transaction begins. The *List* object has version number v90 and the single node in the list (*Node1*) has version v100.



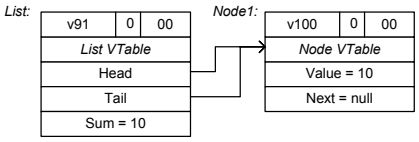
(b) The *List* object is opened for update. A new entry is added to the updated-object log and the object's STM word is set to point to that entry. The entry includes the object's previous STM word (version v90), and a pointer to the transaction manager to identify the thread involved.



(c) The *List* and *Node1* objects are opened for reading, adding two entries to the transaction manager's read-object log. Each entry refers to the object that's been opened and contains a copy of the object's STM snapshot.



(d) Before the *List.Sum* field is updated, the old value is written to the undo log in case of roll-back. The log entry identifies the object holding the field, whether the field is a reference or a scalar, the offset of the field, and the value overwritten in it.



(e) After committing the transaction the list is as in (a), but with the version number incremented in the updated objects.

Figure 4. Transaction log structure for our running example.

Figure 5(d) shows the `CloseUpdatedObject` operation used to close an object that was open for update. Figure 4(e) shows the resulting update to the list structure, with the new version number 91 placed in the list object's header.

In the pseudo-code we have not considered the fact that version numbers may overflow and that, with 29 bits available, we are limited to around 500M distinct versions. Notice that, in our design, it is safe for version numbers to overflow: what is problematic is not the actual overflow, but rather the *re-use* of a version number in the same object while a running transaction has the object open for read. This A-B-A problem [20] can allow a reading transaction to commit successfully without detecting there may have been some 500M updates to the number.

For correctness, we prevent this by (a) performing a GC at least once every 500M transactions, and (b) validating running

```
void DTMOpenForUpdate(tm_mgr tx, object obj) {
  word stm_word = GetSTMWord(obj);

  if (!IsOwnedSTMWord(stm_word)) {
    entry -> obj = obj;
    entry -> stm_word = stm_word;
    entry -> tx = tx;

    word new_stm_word = MakeOwnedSTMWord(entry);
    if (OpenSTMWord(obj, stm_word, new_stm_word)) {
      // Open succeeded: advance our log pointer
      entry ++;
    } else {
      // Open failed: make the transaction invalid
      // (and/or invoke contention manager)
      BecomeInvalid(tx);
    }
  } else if (GetOwnerFromSTMWord(stm_word) == tx) {
    // The object is already open for update by the
    // current transaction: nothing more to do
  } else {
    // The object is already open for update by another
    // transaction: abort our transaction (and/or invoke
    // contention manager)
    BecomeInvalid(tx);
  }
}
```

(a) Pseudo-code to open objects for update.

```
void DTMOpenForRead(tm_mgr tx, object obj) {
  snapshot stm_snapshot = GetSTMSnapshot(obj);
  entry -> obj = obj;
  entry -> stm_snapshot = stm_snapshot;
  entry ++;
}
```

(b) Pseudo-code to open objects for read.

```
void CloseUpdatedObject(tm_mgr tx,
                        object obj,
                        update_entry *entry) {
  word old_stm_word = entry -> stm_word;
  word new_stm_word = GetNextVersion(old_stm_word);
  CloseSTMWord(obj, new_word);
}
```

(d) Pseudo-code to close objects opened for update.

```
void ValidateReadObject(tm_mgr tx, object obj, read_entry *entry) {
  snapshot old_snapshot = entry -> stm_snapshot;
  snapshot cur_snapshot = GetSTMSnapshot(obj);
  word cur_stm_word = SnapshotToWord(cur_snapshot);

  if (old_snapshot == cur_snapshot) {
    // Snapshot match: no-one has closed the object

    if (!IsOwnedSTMWord(cur_stm_word)) {
      // V1: Snapshot unchanged, no conflict
    } else if (GetOwnerFromSTMWord(cur_stm_word) == tx) {
      // V2: Opened by us for update before read
    } else {
      // V4: Opened for update by another tx
      BecomeInvalid(tx);
    }
  } else {
    // Snapshots mismatch: slow-path test on STM word

    word old_stm_word = SnapshotToWord(old_snapshot);
    if (!IsOwnedSTMWord(old_stm_word)) {
      if (old_stm_word == cur_stm_word) {
        // V1: OK: STM word inflated during the transaction
      } else if (!IsOwnedSTMWord(cur_stm_word)) {
        // V5: Conflicting update by another tx
        BecomeInvalid(tx);
      } else if (GetOwnerFromSTMWord(cur_stm_word) == tx) {
        // We opened the object for update...
        update_entry *update_entry = GetEntryFromSTMWord(cur_stm_word);
        if (update_entry -> stm_word != SnapshotToWord(old_snapshot)) {
          // V5: ...but another tx opened and closed the
          // object for update before we opened it
          BecomeInvalid(tx);
        } else {
          // V3: No intervening access by another tx
        }
      } else {
        // V5: The object was opened by another transaction
        BecomeInvalid(tx);
      }
    } else if (GetOwnerFromSTMWord(cur_stm_was) == tx) {
      // V2: Opened by us for update before read
    } else {
      // V4: STM word unchanged, but previously open for
      // update by another transaction
      BecomeInvalid(tx);
    }
  }
}
```

(c) Pseudo-code to validate objects opened for read.

Figure 5. Pseudo-code for opening objects, for validating objects during commit, and for closing objects at the end of commit. For brevity, the `DTMOpen*` operations assume `entry` refers to the next log entry.

transactions at every GC. An entry in the read-object log is only valid if the logged version number matches the current one: the result is that each GC 'resets the clock' of 500M transactions without needing to visit each object.

4. Compiler optimizations

This section describes the static analyses that we have developed to try to improve the placement of `DTM*` operations.

4.1 Extending existing code-motion optimizations

We have extended existing compiler optimizations to the new operations on the decomposed STM interface. The `DTMGetTMMgr` operation is constant and can be subject to common subexpression elimination (CSE) or code motion. The `DTMOpenForRead`, `DTMOpenForUpdate`, and `DTMLog*` operations are idempotent within a transaction. They are also eligible for CSE or code motion, with their availability being killed at transaction boundaries. We extend CSE so that an available `DTMOpenForUpdate` operation can replace a corresponding `DTMOpenForRead` because update access subsumes read access.

The sequencing rules from Section 3 between `DTMOpen*`, `DTMLog*` and subsequent data accesses are expressed as data dependences. We introduce extra output values for earlier operations that are used as extra input values by later operations.

The `DTMGetTMMgr` operation is implemented by fetching the current transaction manager for a thread from a per-thread `Thread` object (and creating the transaction manager if necessary). The Bartok compiler also treats `GetCurrentThread` as a constant operation subject to code motion.

4.2 Avoiding log operations on newly allocated objects

We use a simple flow-sensitive interprocedural analysis to identify variables that are always bound to objects that were allocated since the start of a transaction. We remove the `DTMLog*` operations for assignments to fields and array elements through those variables.

The analysis works as follows. For each basic block, there is a map from object-typed variables to lattice elements. The map represents the kinds of values that may be assigned to a variable at any point in the block. The lattice has three elements in it: `Top` (may not be newly allocated), `New` (must be newly allocated), and

`Bottom` (absence of information). Allocation operations generate `New` for the variables to which they are assigned. Assignments and casts propagate their abstract value. Calls propagate abstract values to the call formals and from the return value. All other operations generate `Top` for the variable to which they are assigned. Transaction start operations generate `Top` values for all variables.

The analysis initializes all maps to `Bottom`. It then propagates information forward and iterates until a fixed point is reached. At a block that is a join point, the maps from predecessor blocks are point-wise combined with the existing map for the block. The beginning of a function is considered a join point from all of its call sites.

4.3 Treating DTM operations as calls

There are several points at which we can replace the abstract DTM operations with calls to the methods that implement them. We explore how two options affect performance. By default, we introduce calls when lowering Bartok's medium-level intermediate representation (IR) to target machine instructions. Alternatively, we can introduce calls earlier, while still working in a higher-level IR. This exposes the calls to inlining.

4.4 Mapping statics to surrogates at compile time

When accessing static fields, we can perform `DTMAddrToSurrogate` at compile time. This avoids a runtime address-to-surrogate mapping and exposes further CSE opportunities for operations on different addresses but the same surrogate.

4.5 Moving logging to callers

We observe that many methods begin by performing `DTMOpen*` operations on parameters that may have already been opened by the caller. To reduce this kind of redundant logging we (a) identify any `DTMOpen*` operations on parameters that postdominate the method's entry point on non-exception paths, (b) create a cloned version of the method without these operations, (c) replace any non-virtual (or devirtualized) calls to the original method with the removed `DTMOpen*` operations and a call to the replacement.

To some extent this is reminiscent of work such as Diniz and Rinard's on computation lock coarsening [9]: unsurprisingly, a `DTMOpenForUpdate` operation can be moved from a method out to its caller in much the same way as a lock acquire-release pair. However, there are a number of differences. First, the STM infrastructure provides for *deadlock recovery* rather than restricting us to transformations that avoid deadlock. Second, our `DTM*` operations are idempotent, letting us remove many operations if they are guaranteed to be preceded by an equivalent operation, rather than requiring us to re-arrange earlier operations to ensure a lock remains held.

4.6 Avoiding read-to-update upgrades

A remaining case where unnecessary logging occurs is when a `DTMOpenForRead` operation is followed by a `DTMOpenForUpdate` operation. This arises from fragments like `this.count++` which first open `this` for reading and then open it for updating.

We handle the specific case of read-to-update upgrades within a basic block by a straightforward dataflow analysis, upgrading `DTMOpenForRead` operations if followed by a `DTMOpenForUpdate`.

We handle the general case by inserting `DTMOpenForUpdate` operations at the beginning of all basic blocks from which all non-exception paths perform the same `DTMOpenForUpdate` (without intervening stores to the variables involved). CSE then attempts to eliminate the extra `DTMOpenForUpdate` operations as well as any subsequent `DTMOpenForRead` operations on the same object. If an exception occurs at runtime, then more objects may be opened for update than otherwise – but this will not affect correctness.

4.7 Decomposing log management

We can reduce the cost of log management further by decomposing log operations. This allows the amortization of the cost of log-management work across multiple operations. In particular, `DTMOpen*` and `DTMLog*` operations start with a check that there is space in the current array. For `DTMOpenForRead`, this is the only check that must be performed in the fast-path version of the code.

To amortize the cost of these checks, we introduce a new operation, `EnsureLogMemory`, taking an integer that indicates how many slots to reserve in a given log[3]. Specialized versions of the `DTMOpen*` and `DTMLog*` operations can assume that space exists.

To reduce runtime bookkeeping, `EnsureLogMemory` operations are not additive: two successive operations reserve the maximum requested, not the total. For simplicity, we do not place the specialized operations where we would require reserved space after a call or back edge. In one version of the optimization, we simply combine reservations for all operations between calls within each basic block. In another version we use a backwards analysis to eagerly reserve space as early as possible, being forced to stop at all calls and loop headers. This has the advantage of combining more reservations but may introduce reservation operations on paths that do not require them.

5. Runtime log filtering

In this section we describe runtime techniques to filter duplicates. There are three techniques: in Section 5.1 we describe how we track objects allocated in the current transaction. Updates to such objects do not need to be logged because, since the object is guaranteed to be dead on abort, there is no need to roll back updates to it. In Section 5.2 we describe a probabilistic hashing scheme to filter duplicates from the read-object log and the undo log. Finally, in Section 5.3, we describe a bitmap-based scheme to deterministically filter duplicates from the undo log.

5.1 Track transaction-local objects

If we can dynamically identify objects allocated by the current transaction, then we can filter out any undo-log entries for them that the static analysis in Section 4.2 is unable to avoid. This is safe because the objects will be dead if the current transaction aborts.

We do this by (a) adding a version of `DTMOpenForUpdate` that is specialised to work on newly allocated objects, (b) having this operation write a designated STM word value to mark the object as transactionally allocated.

Figure 6 depicts the structures used at runtime. In the example, objects *List* and *Node1* have been allocated in the current transaction. Each has an updated-object log entry as usual: these are needed so that the objects can be closed when the transaction commits. However, the STM words refer to a single *transaction-local log entry* (TLLE) instead of entries in the updated-object log.

From the point of view of the current transaction, this lets `DTMLogFieldStore` perform a cheap test of whether or not a prospective store is to a transactionally allocated object: a single comparison is needed against the current transaction's TLLE. From the point of view of other transactions, the *List* and *Node1* objects look like ordinary objects that are currently locked for update.

5.2 Hashing

The hashing scheme probabilistically detects duplicate logging requests to the read-object log and the undo-log. We use per-thread tables that map a hash of an address to details of the most recent

[3] As our results show, reservation sizes are vastly smaller than the arrays from which the log is built: reservations that leave unused space at the end of an array do not yield noticeable fragmentation.

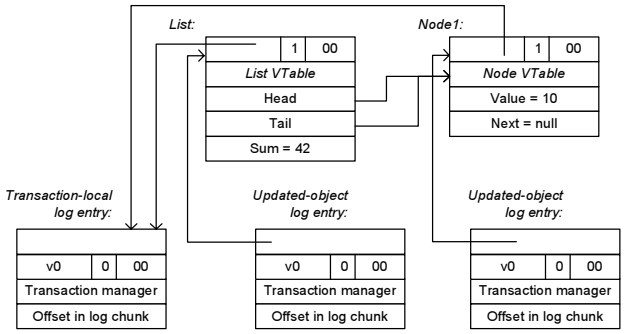


Figure 6. Tracking transaction-local objects: until the transaction that allocated them commits, their STM words refer to a special per-transaction log entry. The example here supposes that the *list* and *node* objects were allocated earlier in the current transaction.

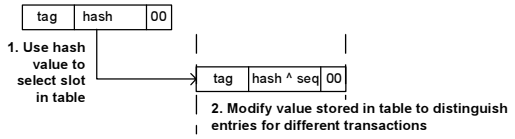


Figure 7. Hashing scheme to detect duplicate logging.

logging operation relating to addresses with that hash. Stores to the read-object log use the address of the object's header word; stores to the undo log use the address of the word being logged. These sets of addresses are disjoint so a single table serves both purposes.

Hashing schemes have been used before to remove duplicates in an undo log; not least in our earlier work [29, 12]. However, previous work requires the table to be cleared *once per transaction*. This is inefficient when the table is large enough to be an effective filter.

Figure 7 shows our new design. The word-aligned address on which duplicate-detection is performed is split into a hash index and a tag. Instead of storing the full address in the table, we combine a portion of a thread-local transaction sequence number with the hash index. The hash index is identical for all values stored in the table entry. Thus, an entry from another transaction will not be confused with an entry from the current transaction. We only need to clear the table when the bits for the sequence number overflow. Because the count of sequence numbers between overflows is the same as the number of table entries, on average we clear only one table entry per transaction.

We use exclusive-or to form the modified value that is stored in the hash table. This is faster than replacing the bits occupied by `hash`.

5.3 Updated-word bitmaps

The final runtime scheme we use is to add per-object bitmaps to detect all duplicate stores to the undo log. This is a deterministic scheme that can be compared against hashing. Similar schemes have been used in DBMSs to remove all but the most recently logged write from a log of updates [21]. Conceptually our scheme can be seen as an analogue that removes all but the first write from an undo log. Of course, the implementation of this idea is much different because it has to be performed online by concurrent threads and the bitmaps must be managed without introducing conflicts between otherwise-unrelated transactions.

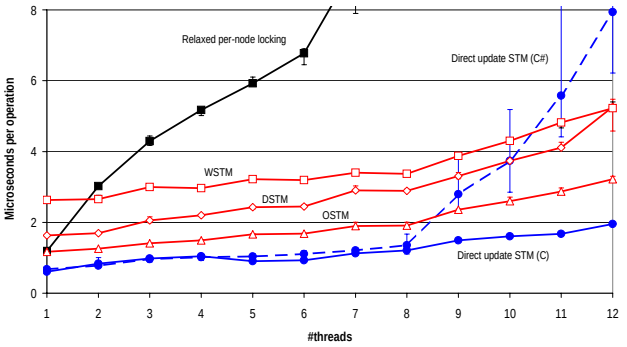


Figure 8. Red-black tree performance.

To support bitmaps we extend entries in the updated-object log by four words when compared with the structure in Figure 4(b). The first three words provide flags that indicate if the transaction has written an undo log entry for each of the first $3*32=96$ words within the object. The fourth word is a pointer to an external bitmap with one bit for each word in the full object.

Why do we use these particular sizes? The three bits reserved at the low end of STM words (Figure 5(a)) mean that entries in the updated-object log are 8-byte aligned, and so extra words must be added to log entries in pairs. We chose to add four words based on the sizes of objects in our benchmarks in Section 7. Of course, alternative implementations are possible.

6. GC-time log compaction

In this section, we turn to the final technique we use: GC-time compaction of the logs. Previous STMs have had limited integration with the GC: either all transactions are aborted when the GC is invoked [12], or the GC considers object references in the transaction logs as roots [13, 16]. The first option cannot support long-running transactions. The second option can retain objects unnecessarily.

We avoid these problems by making the GC aware of the structure of the transaction logs and which references from them need to be treated as strong and which as weak. Furthermore, we use GC time as an opportunity to compact the logs, removing duplicate entries, or entries that are now superfluous. This work scales with the number of entries in the transaction logs (as it must if the GC is to visit all the log entries). We do not perform operations which must visit objects not currently involved in transactions – doing so is incompatible with generational collection.

We remove entries in the read-object log in three cases:

1. If the object was allocated after the transaction began and is now unreachable. It will be lost whether or not the transaction commits. We detect this by simply treating the reference as weak and visiting the log after visiting the undo log, because the undo log entries ensure all objects predating the transaction remain strongly reachable.
2. If the object is open for update by the same transaction and there were no intervening conflicting updates. We use a specialized version of `ValidateReadObject` from Section 3.1.2 in which cases V2 and V3 cause the log entry to be dropped.
3. If there are duplicate entries. We set a bit in the object's header the first time we find an entry for a given object. We then remove log entries relating to objects with this bit set. We use a second pass over the read-object log to clear the bits.

Test		Read-object	Updated-object	Undo	Description
`tree`	original	235 601 949	5 778 625	5 465 513	5 000 000 red-black tree operations from SXM [15]. 6:1:1 mix of lookup:insert:delete uniformly distributed on a 0..65535 key space.
	static	88 573 474	3 456 692	3 592 242	
	dynamic	85 458 480	2 380 391	3 154 563	
`skip`	original	181 062 984	17 785 320	32 218 292	1 000 000 skip-list operations from SXM [15]. Maximum node height 32. Same operation workload as trees.
	static	114 925 456	1 724 370	883 056	
	dynamic	37 143 730	1 503 507	566 808	
`go`	original	39 697 435	5 790 969	5 688 272	Playing on a 10x10 board, computing each move in a separate atomic block. 69 blocks executed.
	static	12 016 600	3 410 033	5 502 127	
	dynamic	37 996	112 669	244 300	
`sort`	original	357 596 090	107 483 100	107 483 040	Merge sort an array of $256 * 2^{10}$ integers 0..65535, ten repetitions, each in a separate atomic block.
	static	114 481 920	54 362 230	107 483 010	
	dynamic	50	70	262 175	
`xlisp`	original	96 347 281	31 890 055	30 516 996	Lisp interpreter running `au` and `ctak` lisp benchmarks, each in a separate atomic block.
	static	47 616 579	19 581 233	29 233 487	
	dynamic	3 506 713	1 378 265	65 732	

Figure 9. Log entries written without any optimization ('original'), with static optimization but no dynamic filtering ('static') and with dynamic filtering to remove duplicates ('dynamic'). Workloads were sized so that they could be completed without optimizations (hence the small board size in `go` and the lisp scripts chosen).

The log entries written by `DTMOpenForUpdate` are straightforward: only the first case applies because the function itself avoids duplication via the atomic compare-and-swap on the STM word.

Similarly, for the undo log, we can remove updates logged for objects which are dead whether or not the transaction commits.

7. Results

We use three sets of benchmarks. First, we look at concurrent data structures which provide a comparison with results from other STMs. Second, we use longer running tests derived from C# implementations of the `099.go`, `129.compress` and `130.li` programs from the SPEC CINT95 suite and an in-memory sequential merge sort. These, of course, are not concurrent algorithms, but they serve to assess the effectiveness of the compile-time optimizations on longer code sequences which run without contention (as we assume is true of many sections of a well-designed concurrent workload). Of course, as we discussed in Section 1.2, atomic blocks are useful in single-threaded applications for exception recovery. Finally, we look at the use of atomic blocks within the ASP.NET Cassini Web Server running on the Singularity Research OS [19].

We split our evaluation into two section: Section 7.1 looks at the multi-processor performance of the underlying STM, while Section 7.2 looks at the effect of the compiler optimizations and runtime integration.

7.1 STM performance

It is difficult to directly compare the performance of our new decomposed direct-access STM with that of previous designs: the interface is different, and only our new interface is integrated in the Bartok compiler. This means that we cannot simply 'plug in' the new STM in an existing test infrastructure, nor can we readily include an alternative STM in the Bartok compiler's runtime.

In order to get a fair assessment of the performance of the new STM we based this part of our evaluation on a re-implementation of the new STM in C. We used the STM implementations and red-black tree test harness from Fraser's thesis work which provides relaxed per-node locking based on Hanke's design [14], WSTM [12], OSTM [10], and DSTM[4] [16].

[4] DSTM is configured to use a 'polite' contention manager that uses exponential backoff. However, little contention is seen in the test workload.

Specialized versions of the test harness are used to optimize the performance of each STM: the WSTM variant uses local variables to avoid repeated reads from the heap, while the OSTM and DSTM variants attempt to avoid opening objects more than once. Following this approach, the harness for decomposed direct-access STM mirrors the optimizations made by Bartok. Consequently, the results indicate the possible performance that could be achieved by a compiler with knowledge of the semantics of each particular STM.

The red-black tree workload performs a 6:1:1 ratio of lookup : insert : delete operations uniformly on a 0..65535 key space. We use a 4*2-core Opteron machine and record the CPU-time required per operation. Figure 8 shows the results: a scalable implementation is indicated by a flat line, while a fast implementation is indicated by a line close to the x-axis.

The performance of relaxed locking, OSTM and DSTM agrees with Fraser's results [10]. WSTM's scalability is similar under low contention but, as one would expect, it is not as fast: it incurs per-word costs that are comparable with the per-object costs of OSTM, and most operations on red-black trees access at least two words in each object. In comparison, the direct-access STM scales well but incurs much lower per-object costs when compared with OSTM.

The graph also shows the performance of a similar red-black tree harness implemented in C# and compiled using Bartok with all our optimizations enabled. As expected the performance tracks that of the C version in which we have manually incorporated the results of the optimizations. The C# implementation does not currently scale beyond the 8 hardware threads available because of a spin-lock used elsewhere in the runtime system.

7.2 Optimizations and runtime integration

Results in this section use a 3GHz Pentium 4 CPU with sufficient physical memory to avoid any disk activity. Results are normalized against the single-threaded run time of the benchmarks without any concurrency control in the benchmark itself, so a result of 2.0 means that a run took twice as long. Results are the best of five runs to reduce perturbations from background processes.

The GC is a two-generation copying collector with a 4MB nursery. A full collection occurs every 8 nursery collections. When using STM, transaction logs are allocated in the same heap as the application state and each chunk in the log holds 1024 entries. When using hashing (Section 5.2), the table holds 2048 entries.

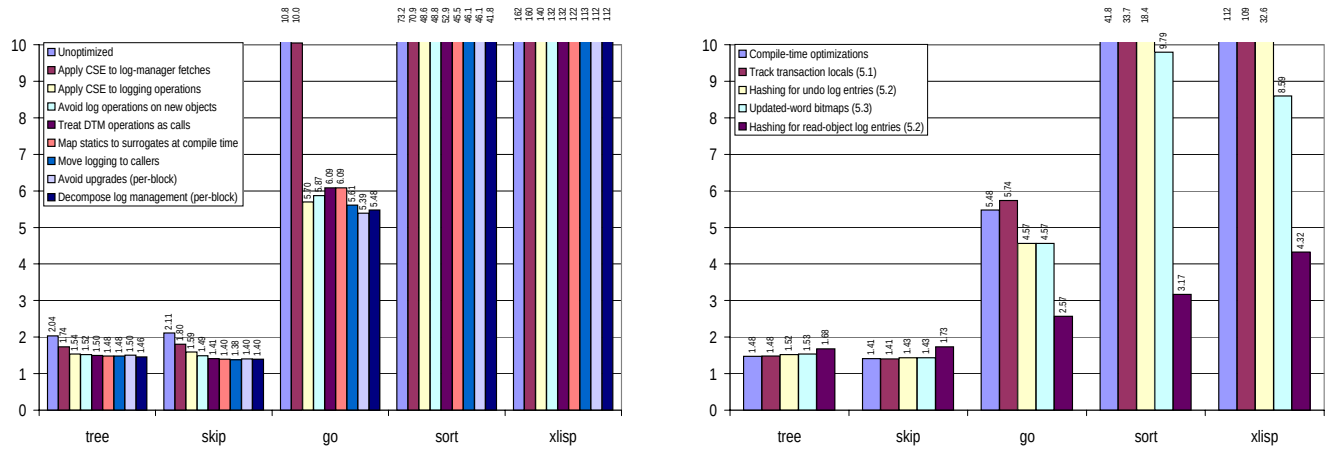


Figure 11. Wall-clock runtimes, normalized so 1.0 is the original performance with no concurrency control in the benchmark.

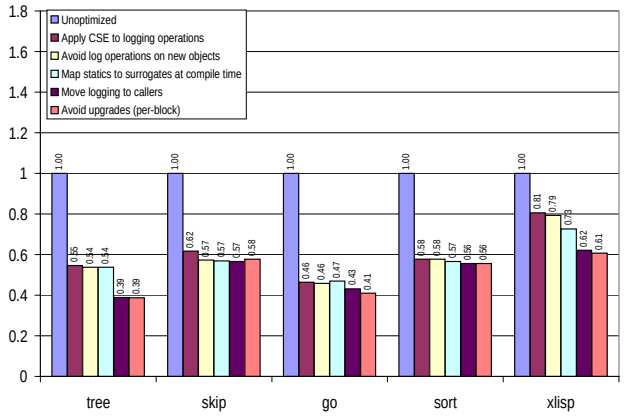


Figure 10. The effect of optimizations on the number of log entries written. Results are normalized against the unoptimized case.

We tested the sensitivity of results to these settings: Our short-running tests generate little garbage and perform essentially unchanged with nursery sizes from 128MB down to 1MB, the size of the L2 data cache. Little log space is used, so performance is flat with log chunk sizes beyond 256 entries. For long-running tests, settings below a 4MB nursery, 1024-entry log chunk and 1024-entry hashtable give poor performance because of memory pressure from log chunks, frequent chunk overflows and hash collisions.

All results here have GC-time log compaction enabled. The cost of this scales with the volume of the transaction logs at the points where GC occurs; it never adds more than 12% to the total GC time (on the longest benchmarks with no optimizations or filtering to reduce the volume of the log).

Figure 9 shows the number of logging operations produced by our test programs: in some cases millions of operations per block. The 'original' lines indicate the baseline performance without any attempt to remove logging work. The 'static' lines show the impact of the optimisations in Section 4. The 'dynamic' lines also perform runtime filtering to remove duplicates.

We first investigate the effect of each of our optimizations in contributing to the reduction from 'original' to 'static'. Figure 10 shows the cumulative reduction in the log entries written as optimizations are enabled. Each cluster of bars deals with a single benchmark and successive optimizations are enabled left-to-right.

The most significant reductions in logging work occur in `tree`, `go`, `sort` and `xlisp` – primarily by exposing decomposed STM operations to CSE. In `tree` the code for traversing down the tree is identical to careful manual placement of logging operations; runtime filtering culls further duplicate log entries written during rotations.

Although we statically eliminate around half of `sort`'s logging, almost all of the remaining entries are duplicates: our current intra-procedural CSE-based techniques are ineffective for merge-sort's recursive structure. Looking at the three logs individually shows that the analysis to identify newly allocated objects is very effective in the `skip` test where temporary objects are used as a mechanism for returning multiple values.

Figure 11(a) shows how this translates to wall-clock execution times when the optimizations are used *without* runtime filtering. `sort` and `xlisp` perform extremely poorly, even with all of the optimizations enabled. This is no surprise: without filtering there is a vast number of duplicate entries in the read-object log, triggering frequent GCs. Although duplicates can be removed at GC time, the volume of logging means that over 77% of `sort`'s execution time and 90% of `xlisp`'s is spent in GC.

We were surprised that further decomposition of log operations did not give noticeable performance improvements (Section 4.7). The analysis identifies several opportunities for decomposition: in `tree` almost all `DTMOpenForRead` operations are decomposed, with one `EnsureLogMemory` per 5.8 logging operations. The same is true in `go` and `xlisp` for stores to the undo log (one `EnsureLogMemory` per 1.9 operations and 2.3 operations respectively). We suspect that performance is limited by the memory traffic of the stores to the log and that, on the superscalar Pentium 4, no benefit is achieved by avoiding the comparisons and predictable branch involved in checking for log space.

Figure 11(b) shows the impact of runtime filtering alongside compile-time optimization. The leftmost bar in each cluster shows the performance with the full set of compile-time optimizations enabled for that benchmark. Note that hashing alone is ineffective for `sort` and `xlisp` (this remains the case if we vary the table size: an impractically large table is needed to avoid collisions). Although the hashing and bitmap schemes slightly degrade the performance of the shortest benchmarks, they are necessary for practical perfor-

mance from the longer running benchmarks. Runtime adaptation may give some benefit here.

As well as the techniques from Section 5, we also studied the impact of *visibly* opening objects for read. This *guarantees* that there are no duplicate entries in the read-object log and reflects another popular design choice in STMs [16]. However, because it requires two atomic compare-and-swap operations per object logged, it vastly reduces performance on the short benchmarks with little change to the performance of the longer ones when compared with our hash-based filtering.

With runtime filtering enabled, GC occurs only during `xlisp`. Log compaction eliminates almost all of the entries in the read-object log: 7.34M related to objects that were subsequently opened for update, and 19K were duplicate entries that, due to hash collisions, had not been removed by filtering. Similarly, almost all of the entries in the updated-object log (1.6M of 1.8M) were removed because they were related to dead objects; in this case temporaries created during the evaluation of expressions in the LISP programs. We saw similar trends when forcing GCs in other benchmarks.

Given the benefits seen from runtime filtering, we investigated if the compile-time optimizations were necessary at all. They are: starting from the best performing combination of filtering techniques in Figure 11(b), running *without* optimization degrades performance by 17%, 16%, 50%, 72% and 25% respectively on the six benchmarks. This is roughly due in equal portions to optimizations that can make individual STM operations faster and optimizations that reduce the number of STM operations executed. We investigated where time is spent within the longer benchmarks. `go` shows a 15%:85% split between time in 'real' STM work (writing to the log and performing `DTMCommit`) and time spent filtering. `sort` shows a 27%:73% split, and `xlisp` a 56%:44% split.

Finally, we performed a whole-system test using atomic blocks in the Cassini web server running on the Singularity Research OS. We modified Cassini with an atomic block around its request parsing code: around 150 lines of C# spread over four methods making heavy use of object-oriented string parsing routines. If a request's headers are malformed then the atomic block performs automatic roll-back before returning an error to the client. We load the web server using a SpecWeb99-derived test harness configured to use up to 10 parallel connections, and measure the time taken to execute the request parsing code for both the original server and our modified version. After warm-up, the baseline parses requests in $53 \pm 1\mu s$, and the new version in $154 \pm 3\mu s$ when using all our optimizations. Of course, since the server is ultimately I/O bound, this extra work does not effect the overall throughput.

8. Related work

A number of early languages included support for features like `atomic` blocks without building them on transactional memory. These are either safe only for uniprocessors or they are extremely pessimistic and serialize non-conflicting `atomic` blocks. distributed applications based on strict two-phase locking of atomic objects [22].

Early work on STM has focused on libraries, such as Herlihy *et al.*'s [16] and Fraser's [10]. Aside from our own work on language integration, Welc *et al.* [30] showed how STM-like techniques can increase the concurrency available in systems based on Java's `synchronized` blocks and Ringenburg and Grossman showed how atomic blocks could be added to OCaml [27].

Although this paper has focused on word-based transactional memory, many of the techniques would apply to object-based designs [16, 10, 23]. There is one notable change in the operations exposed in the compiler's intermediate representation: object-based designs return *handles* when an object is opened and updates are made relative to these handles rather than to the original object reference. This allows writers to be provided with thread-local copies of the object and so it is not necessary to record a separate undo-log (and therefore it is not necessary to filter duplicates from it). Of course, the runtime structures used to represent an object-based STM would differ substantially from those in Section 3.1: existing object-based STMs add at least one level of indirection between an object reference and the object's current contents.

The evolving designs for the Fortress [2], Chapel [7] and X10 [6] languages for high-performance computing all specify forms of atomic block. The optimizations and runtime techniques we have developed will be applicable to these new languages.

The `System.Transactions` namespace in the .NET Framework 2.0 provides resource managers for transacted access to databases, file systems and the configuration registry. Unlike `atomic` blocks, memory accesses within transactions are performed directly. A combination of this work with `atomic` blocks would address many of the questions about how I/O operations should be integrated with memory transactions [11].

Hardware transactional memory was originally proposed by Herlihy and Moss [17]. Early designs buffered a processor's transactional accesses in a local cache and used slight extensions to MESI cache management protocols to detect conflict between transactions. This approach inevitably exposes hardware limits: transactions must be aborted on context switches, and all of the transactional accesses must fit within the capacity and associativity limits of the cache in which they are buffered.

Researchers have only recently turned to the question of how to allow transactions of unbounded size while still being able to enjoy hardware support. Hardware-based speculative lock elision uses a TM-like mechanism to speculatively run lock-based code without actually taking a lock [24, 25]. If hardware limits are reached then execution can fall back to ordinary locking.

Rajwar *et al.*'s *virtualizing transactional memory* splits transaction state between buffers in fast per-processor memory and overflow buffers held in an application's virtual address space [26]. Common-case operations (short transactions that commit successfully) run without using the overflow buffers.

Ananian and Rinard showed how hardware and software transactional memory could be combined by using special 'flag' values to identify where transactions may be operating [3].

9. Conclusion

This paper has taken a four-pronged approach to speeding up word-based transactional memory: direct-access memory to avoid searching logs, compile-time decomposition and optimization to reduce the use of logging operations, fast runtime filtering of duplicate logging requests, and GC-time compaction of logs to deterministically remove dead objects and any duplicates that were missed.

The overall results vary between programs: in micro-benchmarks, where the optimizations approach the quality of hand-placed calls to an STM, execution takes around 1.5x that of the same code without any concurrency control. At the other extreme, on long-running transactions with millions of transactional accesses, execution takes around 2.5-4.5x that of single-threaded versions.

For short blocks, these results are promising and suggest that software-only approaches may be sufficient for some applications. For longer blocks, acceptable speed may require additional hardware support to complement the techniques we have developed, although there may be applications where the software engineering benefits of `atomic` blocks or their parallelism-preserving performance make current performance acceptable.

We remarked in Section 4 that our techniques build on earlier work for optimising the placement of lock/unlock operations. It would be interesting to explore this relationship further: can we

perform the analogue of *data lock coarsening* [9] to vary the granularity with which STM meta-data is associated with objects? One can imagine cases where a single STM word could be used to manage an aggregate object or, conversely, where separate STM words might be used on independent fields of a single object.

Our final conclusion is about how hardware can support atomic blocks. Previous research has suggested 'fall back to software' models in which short blocks execute entirely in hardware and longer ones are implemented using STM. Our results suggest that hardware support for short running blocks needs to be considered in the context of an optimized software implementation. Furthermore, it may be worthwhile to investigate hardware support for long running blocks. It is in these cases that duplicate log removal at runtime and GC-time is necessary, and so an effective implementation of long-running transactions could benefit from hardware support for log filtering as well as simply multi-word concurrent updates.

Acknowledgments

Thanks to Keir Fraser for providing access to his STM implementations and test harness, to the Bigtop group for access to the SMP test machine we used, and to Austin Donnelly for running the Cassini experiments. Thanks to Paul Barham, Dave Detlefs, Eric Jul, Simon Marlow, Simon Peyton Jones, David Richter, Ben Zorn, and of course the anonymous reviewers for their careful reading of earlier drafts of this paper.

References

[1] AGESEN, O., DETLEFS, D., GARTHWAITE, A., KNIPPEL, R., RAMAKRISHNA, Y. S., AND WHITE, D. An efficient meta-lock for implementing ubiquitous synchronization. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)* (Nov. 1999), vol. 34(10) of *ACM SIGPLAN Notices*, pp. 207–222.

[2] ALLAN, E., CHASE, D., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE JR, G. L., AND TOBIN-HOCHSTADT, S. The Fortress language specification v0.618, Apr. 2005.

[3] ANANIAN, C. S., AND RINARD, M. Efficient software transactions for object-oriented languages. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Lanaguages (SCOOL)* (Oct. 2005). Also available in the University of Rochester digital archive.

[4] BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. Thin locks: Featherweight synchronization for Java. In *Programming Language Design and Implementation (PLDI)* (Jun. 1998), vol. 33(5) of *ACM SIGPLAN Notices*, pp. 258–268.

[5] CARLSTROM, B. D., CHUNG, J., CHAFI, H., MCDONALD, A., MINH, C. C., HAMMOND, L., KOZYRAKIS, C., AND OLUKOTUN, K. Transactional execution of Java programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Lanaguages (SCOOL)* (Oct. 2005). Also available in the University of Rochester digital archive

[6] CHARLES, P., DONAWA, C., EBCIOGLU, K., GROTHOFF, C., KIELSTRA, A., SARKAR, V., AND PRAUN, C. V. X10: An object-oriented approach to non-uniform cluster computing. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)* (Oct. 2005), pp. 519–538.

[7] CRAY INC. The Chapel language specification v0.4, Feb. 2005.

[8] DICE, D. Implementing fast Java monitors with relaxed-locks. In *Proceedings of USENIX JVM 2001* (2001), pp. 79–90.

[9] DINIZ, P. C., AND RINARD, M. C. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing 49*, 2 (Mar. 1998), 218–244.

[10] FRASER, K. *Practical lock freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2003.

[11] HARRIS, T. Exceptions and side-effects in atomic blocks. In *PODC 2004 Workshop on Concurrency and Synchronization in Java programs (CSJP)* (Jul. 2004), pp. 46–53. Proceedings published as Memorial University of Newfoundland CS Technical Report 2004-01.

[12] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA)* (Oct. 2003), pp. 388–402.

[13] HARRIS, T., HERLIHY, M., MARLOW, S., AND PEYTON-JONES, S. Composable memory transactions. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Jun. 2005), pp. 48–60.

[14] HANKE, S., OTTMANN, T., AND SOISALON-SOININEN, E. Relaxed Balanced Red-Black Trees. In *Italian Conference on Algorithms and Complexity* (1997), vol. 1203 of *Springer-Verlag LNCS*, pp. 193–204.

[15] HERLIHY, M. SXM1.1: Software transactional memory package for c#. Tech. rep., Brown University & Microsoft Research, May 2005.

[16] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Principles of Distributed Computing (PODC)* (Jul. 2003), pp. 92–101.

[17] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer Architecture (ISCA)* (May 1993), pp. 289–300.

[18] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (Jul. 1990), 463–492.

[19] HUNT, G.C. *et al.* An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[20] INTERNATIONAL BUSINESS MACHINES CORP. *System/370 Principles of Operation*, 1983.

[21] KAUNITZ, J., AND VAN EKERT, L. Audit trail compaction for database recovery. *Commun. ACM 27*, 7 (Jul. 1984), 678–683.

[22] LISKOV, B. Distributed programming in argus. *Commun. ACM 31*, 3 (Mar. 1988), 300–312.

[23] MARATHE, V. J., SCHERER III, W. N., AND SCOTT, M. L. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems* (Oct. 2004).

[24] RAJWAR, R., AND GOODMAN, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th Annual International Symposium on Microarchitecture* (Dec. 2001), pp. 294–305.

[25] RAJWAR, R., AND GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 37(10) of *ACM SIGPLAN Notices*, pp. 5–17.

[26] RAJWAR, R., HERLIHY, M., AND LAI, K. Virtualizing transactional memory. In *International Symposium on Computer Architecture (ISCA)* (Jun. 2005), pp.494–505.

[27] RINGENBURG, M. F., AND GROSSMAN, D. AtomCaml: First-class atomicity via rollback. In *International Conference on Functional Programming (ICFP)* (Sept. 2005), pp. 92–104.

[28] SCHERER III, W. N., AND SCOTT, M. L. Contention management in dynamic software transactional memory. In *PODC 2004 Workshop on Concurrency and Synchronization in Java Programs (CSJP)* (Jul. 2004). Proceedings published as Memorial University of Newfoundland CS Technical Report 2004-01 .

[29] SHINNAR, A., TARDITI, D., PLESKO, M., AND STEENSGAARD, B. Integrating support for undo with exception handling. Tech. Rep. MSR-TR-2004-140, Microsoft Research, Dec. 2004.

[30] WELC, A., JAGANNATHAN, S., AND HOSKING, A. Transactional monitors for concurrent objects. In *European Conference on Object-Oriented Programming (ECOOP)* (Jun. 2004), pp.519–542.