Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects

Maurice Herlihy and Eric Koskinen

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-07-08
July 2007

We describe a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. As long as the linearizable implementation satisfies certain regularity properties (informally, that every method has an inverse), we define a simple wrapper for the linearizable implementation that guarantees that concurrent transactions without inherent conflicts can synchronize at the same granularity as the original linearizable implementation.

Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects

Maurice Herlihy Eric Koskinen

Computer Science Department, Brown University

{mph,ejk}@cs.brown.edu

Abstract

We describe a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. As long as the linearizable implementation satisfies certain regularity properties (informally, that every method has an inverse), we define a simple wrapper for the linearizable implementation that guarantees that concurrent transactions without inherent conflicts can synchronize at the same granularity as the original linearizable implementation.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms concurrency, parallel programming

Keywords software transactional memory boosting

1. Introduction

Software Transactional Memory (STM) has emerged as an alternative to traditional mutual exclusion primitives such as monitors and locks, which scale poorly and do not compose cleanly. In an STM system, programmers organize activities as *transactions*, which are executed atomically: steps of two different transactions do not appear to be interleaved. A transaction may *commit*, making its effects appear to take place atomically, or it may *abort*, making its effects appear not to have taken place at all.

To our knowledge, all transactional memory systems, both hardware and software, synchronize on the basis of *read/write conflicts*. As a transaction executes, it records the locations (or objects) it read in a *read set*, and the memory locations (or objects) it wrote in a *write set*. Two transactions *conflict* if one transaction's read or write set intersects the other's write set. Conflicting transactions cannot both commit. Conflict *detection* can be eager (detected before it occurs) or lazy (detected afterwards). Conflict *resolution* (deciding which transactions to abort) can be implemented in a variety of ways.

Synchronizing via read/write conflicts has one substantial advantage: it can be done automatically without programmer participation. It also has a substantial disadvantage: it can severely and unnecessarily restrict concurrency for certain shared objects. If these objects are subject to high levels of contention (that is, they are "hot-spots"), then the performance of the system as a whole may suffer.

Here is a simple example. Consider a mutable set of integers that provides $\mathsf{add}(x)$, $\mathsf{remove}(x)$ and $\mathsf{contains}(x)$ methods with the obvious meanings. Suppose we implement the set as a sorted linked list in the usual way. Each list node has two fields, an integer value and a node reference next. List nodes are sorted by value, and values are not duplicated. Integer x is in the set if and only if a list node has value field x. The $\mathsf{add}(x)$ method reads along the list until it encounters the largest value less than x. Assuming x is absent, it creates a node to hold x, and links that node into the list.

Consider a set whose state is $\{1, 3, 5\}$. Transaction A is about to add 2 to the set and transaction B is about to add 4. Since neither transaction's pending method call depends on the other's, there is no inherent reason they cannot run concurrently. Nevertheless, calls to $\mathsf{add}(2)$ and $\mathsf{add}(4)$ do conflict in the list implementation, because no matter how A and B's steps are interleaved, one must write to a node read by the other. Bear in mind that transactional delays can be quite long. Unlike conflicts between short-term locks, where the delay is typically bounded by a statically-defined critical section, if transaction A is blocked by B, then A is blocked while B completes an arbitrarily long sequence of steps.

By contrast, a lock-based list implementation easily supports a high level of concurrency. As a thread traverses the list, it successively locks each node a, then locks its successor b = a.next, and then unlocks a. (It is not safe to unlock a before locking b because another thread could remove b from the list in the interval between unlocking a and locking b.) All critical sections are short-lived, and multiple threads can traverse the list concurrently. This technique

[Copyright notice will appear here once 'preprint' option is removed.]

is called *lock coupling* [2]. Moreover, there also exist well-known *lock-free* list implementations [22] that provide even more fine-grained concurrency, relying only on individual compareAndSet() calls for synchronization.

Several "escape" mechanisms have been proposed to address the limitations of STM concurrency control based on read/write conflicts. For example, *open nested transactions* [25] (discussed in more detail later) permit a transaction to commit the effects of certain nested transactions while the parent transaction is still running. Unfortunately, lock-coupling's critical sections do not correspond naturally to properly-nested sub-transactions. Lock coupling can be emulated using an *early release* mechanism that allows a transaction to drop designated locations from its read set [13], but it is difficult to specify precisely when early release can be used safely, and the technique seems to have limited applicability. We are not aware of any prior escape mechanism that approaches the level of concurrency provided by common lock-free data structures.

We are left in the uncomfortable position that well-known and efficient data structures can easily be made concurrent in standard non-transactional models, but appear to be inherently sequential in all known STM models. If transactional synchronization is to gain wide acceptance, however, it must support roughly the the same level of concurrency as as state-of-the-art lock-based and lock-free algorithms, among transactions without real data dependencies. This challenge has two levels: transaction-level and thread-level. At the coarse-grained, transactional level, a transaction adding 2 to the set should not have to wait until a transaction adding 4 to the same set completes. Equally important, at the fine-grained thread level, the concurrent calls should be able to execute at the same degree of interleaving as the best existing lock-based or lock-free algorithms.

This paper introduces *transactional boosting*, a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. We describe how to transform a highly-concurrent linearizable *base object*, implemented without any notion of transactions, into an equally concurrent transactional object.

Transactional boosting treats each base object as a *black box*. It requires only that the object provide a *specification* characterizing its *abstract state* (for example, it is a set of integers), and how its methods affect the state (for example, $\mathsf{add}(x)$ ensures that x is in the set). Transactional boosting also requires certain regularity conditions (basically, that methods have inverses) which we will discuss later.

Transactional boosting complements, but does not completely replace conventional read/write synchronization and recovery. We envision using boosting to implement libraries of highly-concurrent transactional objects that might be synchronization hot-spots, while ad-hoc user code can be protected by conventional means.

This paper makes the following contributions:

- To the best of our knowledge, transactional boosting is the first STM technique that relies on object *semantics* to determine conflict and recovery.
- Because linearizable base objects are treated as black boxes, transactional boosting allows STMs to exploit the considerable work and ingenuity that has gone into libraries such as java.util.concurrent.
- Because we provide a precise characterization of how to use the technique correctly, transactional boosting avoids the deadlock and information leakage pitfalls that arise in open nested transactions [25].
- We identify and formally characterize an important class of *disposable* method calls whose properties can be exploited to provide novel transactional approaches to semaphores, reference counts, free-storage management, explicit privatization, and related problems.
- Preliminary experimental evidence suggests that transactional boosting performs well on simple benchmarks, primarily because it performs both conflict detection and logging at the granularity of entire method calls, not individual memory accesses. Moreover, the number of aborted transactions (and wasted work) is substantially lower.

It must be emphasized that all of the mechanisms we deploy originate, in one form or another, in the database literature from the 70s and 80s. Our contribution is to adapt these techniques to software transactional memory, providing more effective solutions to important STM problems than prior proposals.

2. Software Transactional Memory

We assume an STM where transactions can be serialized in the order they commit, a property called *dynamic atomicity* [36]. For brevity, we assume for now that transactions are not nested. We require the ability to register user-defined handlers called when transactions commit or abort (as provided by DSTM2 [12] and SXM [32]). We now describe our methodology in more detail, postponing formal definitions to Section 5.

Any transactional object must solve two tasks: *synchronization* and *recovery*. Synchronization requires detecting when transactions conflict, and recovery requires discarding speculative changes when a transaction aborts.

The specification for a linearizable base object defines an *abstract state* (such as a set of integers), and a *concrete state* (such as a linked list). Each method is usually specified by a *precondition* (describing the object's abstract state before invoking the method) and a *postcondition*, describing the object's abstract state afterwards, as well as the method's return value.

Informally, two method invocations *commute* if applying them in either order leaves the object in the same state and

returns the same response. In a Set, for example, add(x) commutes with add(y) if x and y are distinct. This commutativity property is the basis of how transactional boosting performs conflict detection.

We define an *abstract lock* [25] associated with each invocation of a boosted object. Two abstract locks *conflict* if their invocations do not commute. Abstract locks ensure that non-commuting method calls never occur concurrently. Before a transaction calls a method, it must acquire that method's abstract lock. The caller is delayed while any other transaction holds a conflicting lock (timeouts avoid deadlock). Once it acquires the lock, the transaction makes the call, relying on the base linearizable object implementation to take care of thread-level synchronization. In the integer set example, the abstract locks for add(x) and add(y) do not conflict when x and y are distinct, so these calls can proceed in parallel.

A method call m has *inverse* m' if applying m' immediately after m undoes the effects of m. For example, a method call that adds x to a set not containing x has as inverse the method call that removes x from the set. A method call that adds x to a set already containing x has a a trivial inverse, since the set's state is unchanged.

When inverses are known, recovery can be done at the granularity of method calls. As a transaction executes, it logs an inverse for each method call in a thread-local log. If the transaction commits, the log is discarded, and the transaction's locks are released. However, if the transaction aborts, the transaction revisits the log entries in reverse order executing each inverse. (A transaction that added x to the set would call remove(x).) When every inverse has been executed, the transaction releases its locks.

Sometimes it is convenient to delay certain method calls until after a transaction commits or aborts. For example, consider an object that generates unique IDs for transactions. The object's abstract state is the pool of unused IDs. It provides an assignID() method that returns an ID not currently in use, removing it from the pool, and a releaseID(x) method that returns ID x to the pool. Any two assignID() calls that return distinct IDs commute, and a releaseID(x) call commutes with every call except an assignID() call that returns x. As a result, if a transaction that obtains x aborts, we can postpone returning x to the pool for arbitrarily long, perhaps forever. For example, if the ID generator is implemented as a counter, then it is sensible never to return x to the pool. We call these *disposable* method calls.

There are other examples of disposable methods. One can implement a *transactional semaphore* that decrements a counter immediately, blocking while the counter value is zero, but postpones incrementing the counter until the calling transaction commits. *Reference counts* would follow a dual strategy: the reference count is incremented immediately, but decremented lazily after the transaction commits. (When an object's reference count is zero, its space can be freed.) Reference counter decrements can also be postponed, allowing deallocation to be done in batches. Similar disposability tradeoffs apply to transactional malloc() and free(), and counters used to manage "privatization" of objects shared by transactional and non-transactional threads.

Transactional boosting is not a panacea. It is limited to objects (1) whose abstract semantics are known, (2) where commutative method calls can be identified, and (3) for which reasonably efficient inverses either exist or can be composed from existing methods. This methodology seems particularly well suited to collection classes, because it is usually easy to identify inverses (for example, the inverse of removing x is to put it back), and many method calls commute (for example, adding or removing x commutes with adding or removing y, for $x \neq y$).

Further, transactional boosting supports a clean separation between low-level thread synchronization, which is the responsibility of the underlying linearizable object implementation, and high-level transactional synchronization, which is handled by the abstract locks and undo log. Non-conflicting concurrent transactions synchronize at the level of the linearizable base object, implying for example, that if the base object is non-blocking for concurrent threads, then it is non-blocking for concurrent non-conflicting transactions. No prior STM technique can achieve this kind of fine-grained thread-level parallelism.

3. Examples

We now consider some examples illustrating how highly-concurrent linearizable data structures can be adapted to provide the same fine-grained thread-level concurrency in transactional systems. Our presentation is informal, postponing more precise definitions to Section 5.

For each example we provide a specification, such as that of the Set in Figure 1. We use the notation method(v)/r to indicate the invocation of method with argument v and response r. In some cases the response is inconsequential to commutativity and is denoted _ and void method calls are simply denoted method(v). Finally, we use the infix symbol $\Leftrightarrow$ to mean that its two arguments commute, and $\not\Leftrightarrow$ when they do not.

3.1 Sets

A Set is a collection of *items* without duplicates. Like the integer set described above, a Set provides add(x), remove(x), and contains(x) methods. A call to add() or remove() returns a Boolean indicating whether the set was modified. Each of these calls has an inverse. A remove(x) call that returns *true* has inverse add(x), and vice-versa. Note that a call's inverse often depends on its result: the inverse to a remove(x) call that returns *false* is vacuous.

Many calls commute: either order yields the same results and produces the same final state. For example, consider a set in state $\{5, 16, 29\}$. If we call add(3) and remove(29) in either order, both return *true*, and the set ends up in

Set Specification

Method	Inverse
add(x)/*false*	noop()
add(x)/*true*	remove(x)/*true*
remove(x)/*false*	noop()
remove(x)/*true*	add(x)/*true*
contains(x)/_	noop()

Commutativity

insert (x)/_ $\Leftrightarrow$ insert (y)/_, $x \neq y$
remove(x)/_ $\Leftrightarrow$ remove(y)/_, $x \neq y$
insert (x)/_ $\Leftrightarrow$ remove(y)/_, $x \neq y$
add(x)/*false* $\Leftrightarrow$ remove(x)/*false* $\Leftrightarrow$ contains(x)/_

Figure 1. Transactional Boosting Specification of a Set

state $\{3, 5, 16\}$. Transactional boosting allows transactions to call commuting methods without blocking, independently of how the set is actually implemented. If either transaction aborts, the inverse method calls will be applied, undoing the aborted transaction's changes to the Set's abstract state.

Figure 1 summarizes Set methods' inverses and commutativity. Indeed *any* call to add(x), remove(y), or contains(z) commutes with the others so long as they have distinct arguments.

3.1.1 Skip List Implementation

A *skip list* [26] is linked list in which each node has a set of short-cut references to later nodes in the list. A skip list is an attractive way to implement a Set because it provides logarithmic-time add(), remove(), and contains() methods.

To illustrate our claim that we can treat base linearizable objects as black boxes, we describe how to boost the ConcurrentSkipListSet class from the java.util.concurrent library. This class is a very efficient, but complicated, lock-free skip list. We will show how to transform this highly-concurrent linearizable library class into an equally concurrent transactional library class without the need to understand how the linearizable object is implemented.

Figure 2 shows part of the SkipListKey class, a transactional Set implementation that is constructed by boosting ConcurrentSkipListSet using a LockKey for synchronization. For brevity, we focus on implementing a set of integers, called *keys*. Before we describe the implementation of the boosted ConcurrentSkipListSet class, we consider some utility classes.

The Log class (Line 9) is a thread-local stack of Entry objects, where Entry is an abstract class that provides an abstract undo() method. This class registers an *abort handler* that pops each Entry from the stack and calls its undo() method.

The LockKey class, as shown in Figure 3.1.1, associates an abstract lock with each key. Key-based locking may block commuting calls (for example, two calls to add(x) when x is in the set), but it provides enough concurrency for practi-

```
1  public class SkipListKey {
2    ConcurrentSkipListSet<Integer> list;
3    LockKey lock;
4     ...
5    public boolean add(final int v) {
6      lock.lock(v);
7      boolean result = list.add(v);
8      if (result) {
9        Log.Entry e = new Log.Entry() {
10         public void undo() {
11           list.remove(v);
12         }};
13         Log.getLocal().add(e);
14     }
15     return result;
16   }
17    ...
18 }
```

Figure 2. The SkipListKey class

```
19 public class LockKey {
20   ConcurrentHashMap<Integer,Lock> map;
21   public LockKey() {
22     map = new ConcurrentHashMap<Integer,Lock>();
23   }
24   public void lock(int key) {
25     Lock lock = map.get(key);
26     if (lock == null) {
27       Lock newLock = new ReentrantLock();
28       Lock oldLock = map.putIfAbsent(key, newLock);
29       lock = (oldLock == null) ? newLock : oldLock;
30     }
31     if (LockSet.add(lock)) {
32       if (!lock.tryLock(LOCK_TIMEOUT,
33        TimeUnit.MILLISECONDS);) {
34         lockSet.remove(lock);
35         Thread.getTransaction().abort();
36         throw new AbortedException();
37       }
38     }
39   }
40    ...
41 }
```

Figure 3. The LockKey class

cal purposes. (Naturally, transactional boosting does not require the programmer to exploit all commutative methods.) This class's commit and abort handlers release the locks (on abort, after replaying the log). The lock's map field is a ConcurrentHashMap (Line 20) that maps integers to locks.[1] The lock(k) method first checks whether there exists a lock for this key, and if not, creates one (Lines 25-30).

Each transaction has a thread-local LockSet tracking the locks it has acquired that must be released when the transaction commits or aborts. The transaction must register commit and abort handlers instructing the STM to release all locks (after replaying the log, if necessary). The transaction tests whether it already has that lock (Line 31). If so, nothing more needs to be done. Otherwise, it tries to acquire the lock (Line 33). If the lock attempt times out, it aborts the transaction (Lines 33-37).

In the boosted skip list (Figure 2), an add(v) call first acquires the lock for v (Line 6), and then calls the linearizable base object's add(v) method (Line 7). If the return value indicates that the base object's state has changed (Line 8), then the caller creates a new log entry indicating that v should be removed on abort (Lines 9-12). It pushes this entry onto the log (Line 13) and returns the base method's result.

3.2 Priority Queues

A *priority queue* (PQueue) is a collection of keys, where the domain of keys has a natural total order. Unlike Sets, PQueues may include duplicate keys. A priority queue provides an add(x) method that adds x to the collection, a removeMin() method that returns and removes the *least* key in the collection, and a min() method that returns but does not remove the least key.

Priority queue methods and their inverses are listed in Figure 4. The inverse for a removeMin() call that returns x is just add(x). In most linearizable heap implementations, removing x and adding it again may cause the internal structure of the heap to be restructured, but such changes do not cause synchronization conflicts because the PQueue's abstract set is unchanged. The min() method does not change the queue's state, and needs no inverse.

Most priority queue classes do not provide an inverse to add(x). Nevertheless, it is relatively easy to synthesize one. We create a simple Holder class containing the key and a Boolean deleted field, initially *false*. Holders are ordered by their key values. Instead of adding the key to the PQueue, we add its holder. To undo the effects of an add() call, the transaction sets that Holder's deleted field to *true*, leaving the Holder in the queue. We change the transactional removeMin() method to discard any deleted records returned by the linearizable base object's removeMin(). (We will show an example later.)

[1] We use maps and locks from the Java concurrency packages.

Priority Queue Specification

Method	Inverse
removeMin()/x	add(x)/_
min()/x	noop()
add(x)/_	addInverse(x)/_

Commutativity

add(x)/_ $\Leftrightarrow$ add(y)/_

removeMin()/x $\Leftrightarrow$ add(y)/_, $x \leq y$

min()/x $\Leftrightarrow$ min()/x

Figure 4. Transactional Boosting Specification of a Priority Queue

```
42 public class HeapRW {
43   ConcurrentHeap<Holder> heap;
44   LockRW lock;
45   public void add(int item) {
46     lock.readLock();
47     heap.add(new Holder(item));
48     Log.Entry e = new Log.Entry() {
49       public void undo() {
50         holder.deleted = true;
51       }};
52     Log.getLocal().add(e);
53   }
54   public int removeMin() {
55     Holder holder;
56     lock.writeLock();
57     do {
58       holder = heap.removeMin();
59     } while (holder.deleted);
60     if (holder != null) {
61       Log.Entry e = new Log.Entry() {
62         public void undo() {
63           heap.add(holder);
64         }
65       };
66       Log.getLocal().add(e);
67     }
68     return holder.value;
69   }
```

Figure 5. The HeapRW class

All add() calls commute. Additionally, removeMin()/x commutes with add(y) if $x \leq y$. Here too, commutativity depends on both method call arguments and results.

3.2.1 Heap Implementation

Priority queues are often implemented as *heaps*, which are binary trees where each item in the tree is less than its descendants. We implemented the linearizable concurrent heap implementation due to Hunt *et al.* [17]. The removeMin()

Pipeline Specification

Method	Inverse
offer (x)/_	takeLast()/x
take()/x	offerFirst (x)

Commutativity

offer (x)/_ $\Leftrightarrow$ take()/_, pipeline non-empty
offer (x)/_ $\not\Leftrightarrow$ take()/y, otherwise

Figure 6. Transactional Boosting Specification of a Pipeline

method removes the root and re-balances the tree, while add(x) places the new value at a leaf, and then "percolates" the value up the tree. This implementation uses fine-grained locks. (Because locks are not nested, this algorithm is not a good candidate for open nested transactions.)

Figure 5 shows part of the boosted heap implementation. The heap field (Line 43) is the base linearizable heap, and the lock field (Line 44) is a two-phase readers-writers lock. The readLock() method acquires the lock in shared mode, and writeLock() in exclusive mode. All such locks are released when the transaction commits or aborts. Each add() call acquires a shared-mode lock (Line 46), relying on the base object's thread-level synchronization to coordinate concurrent add() calls. As described earlier, the add() method does not add the key directly to the base heap, but instead creates a Holder containing the key and a Boolean deleted flag (Line 47). For recovery, it logs a call to mark that key's Holder as deleted (Lines 48-52).

3.3 Pipelining

Pipelining is a well-established way to achieve concurrency in certain applications. Algorithms employing pipelining are common in areas such as networking or graphics. In typical pipelined applications, one thread (or transaction) is in charge of each stage of the pipeline. Because different pipeline stages may need varying amounts of time, threads communicate by bounded queues called *buffers*. If a thread tries to enqueue an item to a full buffer, it blocks, as does a thread that tries to dequeue from an empty buffer. The larger the buffer, the greater the tolerance for asynchrony.

In a transactional mode, matters become even more complicated. Pipelining introduces the need for *conditional synchronization*: a transaction that encounters an empty (or full) buffer should block until that buffer becomes non-empty (or non-full). The need for conditional synchronization implies that not all existing STMs support pipelining.

Figure 6 shows the BlockingQueue methods and their inverses. Following Java conventions, the BlockingQueue class provides two methods: offer () enqueues a work item on a queue, while take() dequeues a work item. If the queue is full, however, offer () blocks until there is room, and if the queue is empty, take() blocks until a work item is available.

```
70  public class BlockingQueue<T> {
71    BlockingDeque<T> queue;
72    TSemaphore full;    // block if full
73    TSemaphore empty; // block if empty
74    public BlockingQueue(int capacity) {
75      queue = new LinkedBlockingDeque<T>(capacity);
76       full  = new TSemaphore(capacity);
77      empty = new TSemaphore(0);
78    }
79    public void offer(final T value) {
80       full.acquire();
81      queue.offerLast(value);
82      empty.release();
83      Log.Entry e = new Log.Entry() {
84        public void undo() {queue.takeLast();}
85      };
86      Log.getLocal().add(e);
87    }
88    public T take() {
89      empty.acquire();
90      T result = queue.takeFirst();
91       full.release();
92      Log.Entry e = new Log.Entry() {
93        public void undo() {
94          queue.offerFirst(result);
95        }
96      };
97      Log.getLocal().add(e);
98      return result;
99    }
100 }
```

Figure 7. The BlockingQueue class

Because BlockingQueue does not provide inverses, we take as linearizable base class a blocking *double-ended queue* (BlockingDEQueue) taken from java.util.concurrent. This class provides offerFirst (x), offerLast (x), takeFirst (), and takeLast() methods. A transactional offer (x) call results in a linearizable offerFirst (x) call, with inverse takeFirst ().

Because BlockingQueue objects are shared by pairs of transactions, of which one repeatedly calls offer (), and the other repeatedly calls take(), we care only about commutativity between these two methods. Here, commutativity depends on the queue's abstract state: offer () commutes with take() if and only if the buffer is non-empty.

3.3.1 Pipeline Implementation

To detect when BlockingQueue methods within a pipeline can proceed in parallel, we introduce a *transactional semaphore* class (TSemaphore) to mirror the queue's committed state. Figure 7 shows the BlockingQueue implementation. It uses two transactional semaphores: the full semaphore blocks

Unique ID Generator Specification

Method	Inverse	Post-Abort
assignID()/x	noop()	releaseID(x/_)

Commutativity

assignID()/x ⇔ assignID()/y $x \neq y$

assignID()/x ⇔ assignID()/x

Figure 8. Transactional Boosting Specification of a Unique ID Generator

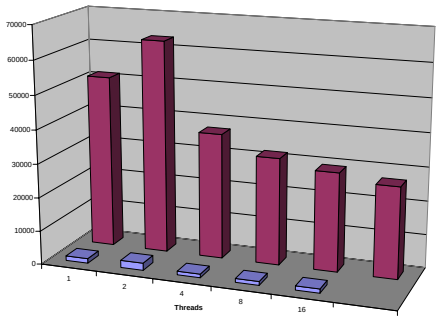

Figure 9. Throughput for red-black tree implementations using shadow copies (front) and boosting (back)

the caller when the queue is full by counting the number of empty slots. It is initially set to the queue capacity (Line 76). The empty semaphore blocks the caller while the queue is empty by counting the number of items in the queue. It is initially set to zero (Line 77). As noted above, the acquire() method, which decrements the semaphore, takes effect immediately, blocking the caller while the semaphore's committed state is zero. The release() method is disposable: it takes effect only when the transaction commits. Note that transactional semaphores cannot be implemented by conventional read/write synchronization: they require boosting to avoid deadlock.

The offer() method decrements the full semaphore before calling the base queue's offerLast() method (Line 80). When the decrement returns, there is room. After placing the item in the base queue, offer() increments the empty semaphore (Line 82), ensuring that the item will become available after the transaction commits. The take() method increments and decrements the semaphores in the opposite order.

3.4 Unique Identifiers

Generating unique IDs is a well-known problem for STMs based on read/write conflicts. The obvious approach, incrementing a shared counter, introduces false read/write conflicts. Under transactional boosting, we would define an ID generator class that provides an assignID() method that returns an ID distinct from any other ID currently in use. Note that assignID()/x commutes with assignID()/y for $x \neq y$.

If a thread aborts after obtaining ID x from assignID() then, strictly speaking, we should put x back by calling releaseID(x), which returns x to the pool of unused IDs. Nevertheless, release is disposable: we can postpone putting x back (perhaps forever). As long as x is assigned, no transaction can observe (by calling assignID()) whether x is in use. Figure 8 shows the commutativity specification for a unique ID generator. Transactional boosting not only permits a transactional unique ID generator to be implemented as a getAndAdd() counter, it provides a precise explanation as to why this implementation is correct.

4. Evaluation

We now describe some experiments testing the performance of transactional boosting. In each experiment, each thread repeatedly starts a transaction, calls a method, and then sleeps for 100 milliseconds (simulating work on other objects), and then tries to commit the transaction. We measure the number of transactions completed in a fixed duration. Tests were implemented in Java using the DSTM2 [12] software transactional memory system and the java.util.concurrent libraries. These experiments were run on a Sun Microsystems T2000 system with 32 cores.

4.1 Red-Black Trees

We first consider a Set implemented as a *red-black tree*, an approximately balanced search tree [5]. The following experiment compares the throughput of a transactionally-boosted *red-black tree* Set implementation versus a transactional red-black tree based on read/write sets.

We started with a sequential red-black tree implementation. For the boosted class, we made all the sequential methods **synchronized**, yielding a linearizable base type with no thread-level concurrency, and we protected the transactional class with a single two-phase lock, yielding no transactional concurrency. For the *shadow* transactional class, we started with the same sequential code, and applied DSTM2's *shadow factory* to ensure that the first time a transaction modifies a tree node, the factory creates a "shadow copy" for possible later recovery.

Figure 9 shows a substantial difference in throughput. There are several reasons for this difference. One is simply that the shadow copy implementation has a substantially higher rate of aborts because read/write conflict detection introduces false conflicts. This difference is also attributable to differences in granularity of synchronization and recovery. In DSTM2, each access to each field of each tree node requires synchronization overhead, and each first write ac-

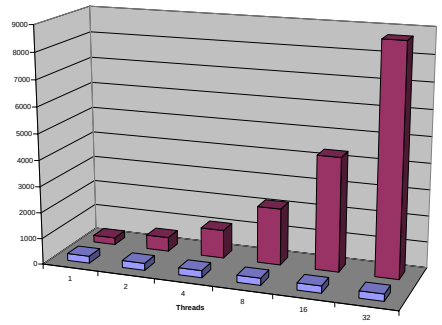

Figure 10. Throughput for transactionally-boosted skip lists using simple (front) and key-based (back) two-phase locks

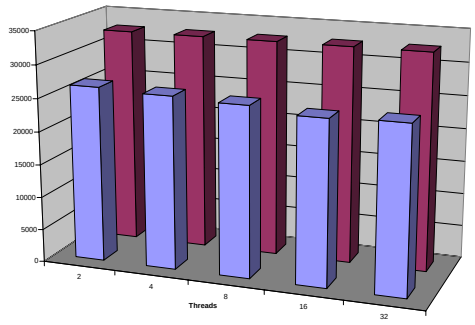

Figure 11. Throughput for boosted heap implementations using a single lock (front) and a read/write-lock (back)

cess copies the node. In the boosted implementation, however, the entire traversal is protected by a transaction-level lock and a method-level Java monitor lock, each of which is acquired only once per transaction. The boosted implementation does not copy any fields, it simply logs references to inverse operations.

4.2 A Lock-Free Skip List

To test the effect of transactional lock granularity, we tested two transactionally-boosted implementations of the lock-free ConcurrentSkipListSet base class. The first uses a single transactional lock for all method calls, while the second uses a lock per key. Because they use the same base object, difference in throughput can be attributed entirely to differences in parallelism. Figure 10 shows that fine-grained transactional synchronization can have a dramatic effect on throughput, especially when combined with fine-grained thread-level synchronization.

4.3 A Concurrent Heap

Figure 11 shows the relative throughput of two heap implementations executing half add() calls and half removeMin() calls. As noted above, the base object is protected by a readers/writers lock, where add() calls acquire the shared reader's lock, and removeMin() calls the exclusive writer's lock. This experiment suggests that using a read/write lock to discriminate between add() and removeMin() calls is worthwhile.

5. Formal Model

Here are the formal definitions underlying transactional boosting. This model is adapted from Weihl [36] and from Herlihy and Wing [16].

5.1 Histories

A computation is modeled as a *history*, that is, a sequence of instantaneous *events*. Here are the events associated with a transaction T:

- $\langle T\ \mathsf{init}\ \rangle$: T starts
- $\langle T\ \mathsf{commit}\rangle$: T commits
- $\langle T\ \mathsf{abort}\rangle$: T aborts, and starts rolling back its effects
- $\langle T\ \mathsf{aborted}\rangle$: T finishes rolling back its effects
- $\langle T, \mathsf{x.m}(v)\ \rangle$: Transaction T invokes method m of object x with argument v.
- $\langle T, v\ \rangle$: A method invoked by T returns v.

For example, here is a history in which transaction A adds 3 to a skip list:

$$\langle A\ \mathsf{init}\ \rangle \cdot \langle A,\ \mathsf{list.add}(3)\ \rangle \cdot \langle A, true\ \rangle \cdot \langle A\ \mathsf{commit}\rangle$$

A single transaction run in isolation defines a *sequential history*. A *sequential specification* for an object defines a set of *legal* sequential histories for that object. For example, the Skip List sequential specification would define the following history to be illegal:

$$\cdots \langle A,\ \mathsf{list.contains}(3)\ \rangle \cdot \langle A,\ true\ \rangle \cdot \langle A,\ \mathsf{list.contains}(3)\ \rangle \cdot \langle A,\ false\ \rangle \cdots$$

A Skip List cannot "change its mind" about whether it contains a given item unless that item is removed or inserted.

A *concurrent history* is one in which events of different transactions are interleaved. For example:

$$\langle A\ \mathsf{init}\ \rangle \cdot \langle B\ \mathsf{init}\ \rangle \cdot \langle B,\ \mathsf{list.remove}(3)\ \rangle \cdot \langle B,\ false\ \rangle \cdot \langle A\ \mathsf{abort}\rangle \cdot \langle B\ \mathsf{commit}\rangle$$

A *subhistory* of a history h is a subsequence of the events of h. For a history h, $h|T$ is the subsequence of events labeled with transaction T, and $h|x$ is the subsequence labeled with

object x.

$$h|T = \langle B \ \mathsf{init}\rangle \cdot \langle B, \ \mathsf{list.remove}(3)\rangle \cdot \langle B, \ false\rangle \langle B \ \mathsf{commit}\rangle$$
$$h|list = \langle B, \ \mathsf{list.remove}(3)\rangle \cdot \langle B, \ false\rangle$$

We sometimes abuse notation, referring to a *set* of transactions or events as in $h|\{T_1, T_2\}$.

Two histories h and h' are *equivalent* if for every transaction A, $h|A = h'|A$. If h is a history, $committed(h)$ is the subsequence of h consisting of all events of committed transactions.

Definition 5.1. *A history h is* strictly serializable *if $committed(h)$ is equivalent to a legal history in which these transactions execute sequentially in the order they commit.*

For example the history

$$\langle A \ \mathsf{init}\rangle \cdot \langle B \ \mathsf{init}\rangle \cdot \langle A, \ \mathsf{list.insert}(3)\rangle \cdot \langle A, \ true\rangle$$
$$\cdot \langle B, \ \mathsf{list.contains}(3)\rangle \cdot \langle B, \ true\rangle \cdot \langle B \ \mathsf{commit}\rangle \cdot \langle A \ \mathsf{commit}\rangle$$

is strictly serializable as follows:

$$\langle A \ \mathsf{init}\rangle \cdot \langle A, \ \mathsf{list.insert}(3)\rangle \cdot \langle A, \ true\rangle \cdot \langle A \ \mathsf{commit}\rangle$$
$$\cdot \langle B \ \mathsf{init}\rangle \cdot \langle B, \ \mathsf{list.contains}(3)\rangle \cdot \langle B, \ true\rangle \cdot \langle B \ \mathsf{commit}\rangle$$

By contrast, the following history is not serializable

$$\langle A \ \mathsf{init}\rangle \cdot \langle B \ \mathsf{init}\rangle \cdot \langle A, \ \mathsf{list.insert}(3)\rangle \cdot \langle A, \ true\rangle$$
$$\cdot \langle B, \ \mathsf{list.contains}(3)\rangle \cdot \langle B, \ true\rangle \cdot$$
$$\langle A, \ \mathsf{list.remove}(3)\rangle \cdot \langle A, \ true\rangle \cdot \langle B \ \mathsf{commit}\rangle \cdot \langle A \ \mathsf{commit}\rangle$$

even though it is a legal history. The commit order places B before A, but B observes a value added by A.

5.2 Method Call Properties

Definition 5.2. *Histories h and h'* define the same state *if, for every history g, $h \cdot g$ is legal if and only if $h' \cdot g$ is.*

5.2.1 Method Call Inverse

In the Skip List example, consider some initial history h. Now, let us define h' as h followed by additional events:

$$h' = h \cdot \langle A, \ \mathsf{list.insert}(3)\rangle \cdot \langle A, \ true\rangle$$
$$\cdot \langle A, \ \mathsf{list.remove}(3)\rangle \cdot \langle A, \ true\rangle$$

It is easy to see that h and h' define the same state. Informally, if an element is added to a list and then removed, the list is returned to its initial state. For this example, remove() is the inverse of insert() (eliding item values for the moment).

Definition 5.3. *For a history h and any given invocation I and response R, let I^{-1} and R^{-1} be the* inverse *invocation and response. That is, the invocation and response such that the state reached after the history $h \cdot I \cdot R \cdot I^{-1} \cdot R^{-1}$ is the same as the state reached after history h.*

If I does not modify the data structure, its inverse I^{-1} is trivial; we denote it noop(). Note that inverses are defined in terms of method calls (that is, invocation/response pairs), not invocations alone. For example, one cannot define an inverse for the removeMin() method call of a heap without knowing which value it removed.

5.2.2 Commutativity

In the Skip List example, the following two histories define the same state:

$$\langle A, \ \mathsf{list.insert}(3)\rangle \cdot \langle A, \ true\rangle \cdot$$
$$\langle A, \ \mathsf{list.insert}(5)\rangle \cdot \langle A, \ true\rangle$$

$$\langle A, \ \mathsf{list.insert}(5)\rangle \cdot \langle A, \ true\rangle \cdot$$
$$\langle A, \ \mathsf{list.insert}(3)\rangle \cdot \langle A, \ true\rangle$$

Here we take advantage of the commutativity of the insert() method for distinct values. No matter how these method calls are ordered, they leave the object in the same final state.

Definition 5.4. *Two method calls I, R and I', R'* commute *if: for all histories h, if $h \cdot I \cdot R$ and $h \cdot I' \cdot R'$ are both legal, then $h \cdot I \cdot R \cdot I' \cdot R'$ and $h \cdot I' \cdot R' \cdot I \cdot R$ are both legal and define the same state.*

Commutativity identifies method calls that are in some sense orthogonal and have no dependencies on each other. In the example above, the method calls

$$\langle A, \ \mathsf{list.insert}(3)\rangle, \langle A, \ true\rangle$$
$$\text{and} \quad \langle A, \ \mathsf{list.insert}(5)\rangle, \langle A, \ false\rangle$$

commute because starting at any state they are both legal, appending them in either order yields the same state.

5.2.3 Disposable Method Calls

Returning to the Unique ID Generator example, consider the following history:

$$\langle A \ \mathsf{init}\rangle \cdot \langle B \ \mathsf{init}\rangle \cdot \langle A, \ \mathsf{gen.assignID}()\rangle \cdot \langle A, 3\rangle \cdot \langle A \ \mathsf{abort}\rangle$$
$$\cdot \langle A \ \mathsf{aborted}\rangle \cdot \langle B, \ \mathsf{gen.assignID}()\rangle \cdot \langle B, 4\rangle \cdot \langle B \ \mathsf{commit}\rangle$$
$$\cdot \langle A, \ \mathsf{gen.releaseID}(3)\rangle \cdot \langle A, true\rangle$$

In this history, after A aborts it delays before releasing the ID 3. The delay is long enough such that B obtains the ID 4 rather than 3 (we assume the generator assigns IDs sequentially). This example illustrates that the releaseID() call can be arbitrarily postponed.

For a history h, let G be the set of histories g such that $h \cdot g$ is legal.

Definition 5.5. *A method call denoted $I \cdot R$ is* disposable *if, $\forall g \in \mathcal{G}$, if $h \cdot I \cdot R$ and $h \cdot g \cdot I \cdot R$ are legal, then $h \cdot I \cdot R \cdot g$ and $h \cdot g \cdot I \cdot R$ are legal and both define the same state.*

In other words, the method call $I \cdot R$ can be postponed arbitrarily without anyone being able to tell that $I \cdot R$ did not occur. When $I \cdot R$ does occur it may alter future computation, but postponing it arbitrarily long always results in a legal history. In the above definition we quantify over all possible histories that proceed h and end with $I \cdot R$.

A notable subset of $\mathcal{G}$ are those that consist of single method calls. For this subset, disposability collapses to commutativity in effect as per the following lemma.

Lemma 5.1. *A disposable method call $I_d \cdot R_d$ commutes with all method calls in G.*

Proof. Trivial. Follows from the definition of disposable methods, letting $g \equiv I \cdot R$. □

5.3 Model of Active Computation

In this subsection, we present a model of active computation. We first define the model in terms of the framework presented thus far. Then we introduce rules for any system that follows our model. Finally, we present our main correctness result for such systems.

Rule 1: Linearizability. For any history h and object x, two concurrent invocations I and I' on x are equivalent to either the history $h \cdot I \cdot R \cdot I' \cdot R'$ or the history $h \cdot I' \cdot R' \cdot I \cdot R$

Linearizability [16] requires that each method call appears to take effect instantaneously some time between its invocation and its response.

The premise of transactional boosting is the ability to execute commuting method calls concurrently. However, we must be able to accommodate all method calls, even those that do not commute. We accomplish both goals by only allowing a transaction to execute if all of its method calls commute with the method calls of all other concurrently executing transactions. We present this formally shortly, but for now focus on non-commutative method calls. To this end, we define the following function:

Definition 5.6. *Let $precedes(h)$ define a partial ordering of transactions. Specifically, if $(T_1, T_2) \in precedes(h)$ then T_1 must commit or abort before any additional method calls in T_2 are invoked.*

In the Skip List example, $\langle A,$ list.insert$(3)\ \rangle$ does not commute with $\langle B,$ list.remove$(3)\ \rangle$. So we must ensure that either $(A, B) \in precedes(h)$ or $(B, A) \in precedes(h)$. With this definition in hand, we present the second rule:

Rule 2: Commutativity Isolation. For any non-commutative method calls $I_1, R_1 \in T_1$ and $I_2, R_2 \in T_2$, it must be the case that either $(T_1, T_2) \in precedes(h)$ or $(T_2, T_1) \in precedes(h)$.

Intuitively, this rule prevents non-commutative operations from being active at the same time. Returning to our example, since $\langle A,$ list.insert$(3)\ \rangle$ does not commute with $\langle B,$ list.remove$(3)\ \rangle$, we must assert that either $(A, B) \in precedes(h)$ or $(B, A) \in precedes(h)$.

Note that this rule does not specify the locking discipline, but rather specifies a property of histories resulting from all possible (correct) disciplines. In practice, choosing a locking discipline is an engineering decision. A discipline that is optimal in the sense that no two commutative operations are serialized may suffer performance overhead from the computation involved in implementing the locking discipline. By contrast, an overly conservative approximation may inhibit all concurrency. In Section 4 we quantified this trade-off with some examples.

The next rule concerns the behavior of an aborting transaction. At the point when a transaction decides to abort, it must subsequently invoke the inverse method calls of all method calls completed thus far. Formally:

Rule 3: Compensating Actions. For any history h and transaction T, if $\langle T$ aborted$\rangle \in h$, then it must be the case that $h|T = \langle T$ init $\rangle \cdot I_0 \cdot R_0 \cdots I_i \cdot R_i \cdot \langle T$ aborted$\rangle \cdot I_i^{-1} \cdot R_i^{-1} \cdots I_0^{-1} \cdot R_0^{-1} \cdot \langle T$ aborted$\rangle$ where i indexes the last successfully completed method call.

In the absence of any other concurrent transactions, it follows easily from the definition of inverse that Rule 3 ensures the system will be restored to its state just before the $\langle T$ init $\rangle$ event. For example, consider the transaction over a heap that calls add(63) and then removeMin$()$. If the transaction aborts after calling add(63), then it will execute addInverse(63) instead of removeMin$()$, and 63 will be removed from the heap.

When a transaction aborts, it does not need to acquire locks to undo its effects. This claim follows from Lemma 5.2. This property is important because for alternative techniques, such as nested open transactions, it can be challenging to ensure that compensating actions (the analog of inverse methods) do not cause deadlocks.

Lemma 5.2. *(Inverse Commutativity): If $I \cdot R \Leftrightarrow I' \cdot R'$ then $I \cdot R \Leftrightarrow I' \cdot R' \cdot I'^{-1} \cdot R'^{-1}$.*

Proof. We must show that for any history h, the three histories

$$\begin{aligned} h_1 &= h \cdot I \cdot R \cdot I' \cdot R' \cdot I'^{-1} \cdot R'^{-1} \\ h_2 &= h \cdot I' \cdot R' \cdot I'^{-1} \cdot R'^{-1} \cdot I \cdot R \\ h_3 &= h \cdot I' \cdot R' \cdot I \cdot R \cdot I'^{-1} \cdot R'^{-1} \end{aligned}$$

define the same state. It follows immediately from the definition of inverse that the states defined by h_1 and h_2 are

equivalent to the state defined by $h \cdot I \cdot R$. Finally, by the commutativity of I, R and I', R',

$$h_3 = h \cdot I \cdot R \cdot I' \cdot R' \cdot I'^{-1} \cdot R'^{-1} = h_1$$

□

In the Skip List example, consider a transaction T_1 consisting of $\mathsf{add}(3)$ and T_2 consisting of $\mathsf{remove}(5)$. The inverse method for T_2 is $\mathsf{add}(5)$. Clearly $\mathsf{add}(3)$ commutes with $\mathsf{remove}(5)$ followed by $\mathsf{add}(5)$, because whether 5 is in the list is unrelated to whether 3 is in the list.

The final rule concerns method calls that occur *after* a transaction commits or aborts:

Rule 4: Disposable Methods. For any history h and transaction T, any method call $\langle T,\ \mathsf{xm}(v)\ \rangle \cdot\ \langle T, r\rangle$ that occurs after $\langle T\ \mathsf{commit}\rangle$ or after $\langle T\ \mathsf{abort}\rangle$ must be disposable.

As a result of this rule, if T generates a method call after it commits, regardless of how far into the future the method call occurs, the history is legal. For example in the unique ID generator, the method call $\mathsf{releaseID}\,(x)$ can be arbitrarily postponed after the transaction aborts. The timing of these delayed disposable methods is entirely an engineering decision, as discussion in Section 3.

5.4 Correctness

We now define notation to represent the instantaneous computation. First, we define functions to categorize transactions as follows, all in terms of the state of the world just after a history h:

- $active(h)$ is the set of all transactions which have been initiated, but not yet committed nor are aborting.
- $committed(h)$ is the set of all committed transactions.
- $aborting(h)$ is the set of all transactions currently aborting.
- $aborted(h)$ is the set of all transactions that have finished aborting.

We similarly define functions to categorize method calls on objects:

- $perm(h)$ is the set of method call events which are associated with committed transactions.
- $live(h)$ is the set of method call events corresponding to uncommitted, unaborting, unaborted transactions.
- $stale(h)$ is the set of method call events corresponding to aborting transactions.
- $reverting(h)$ is the set of *inverse* method call events corresponding to aborting transactions.
- $invisible(h)$ is the set of method call events corresponding to aborted transactions.
- $disposable(h)$ is the set of *disposable* method call events for transactions that have either committed or aborted.

For example in the history

$$h = \langle A\ \ \mathsf{init}\ \rangle \cdot\ \langle B\ \ \mathsf{init}\ \rangle \cdot\ \langle A,\ \ \mathsf{list.insert}\ (3)\ \rangle \cdot\ \langle B\ \mathsf{abort}\rangle$$

Here, $active(h) = \{A\}$, $aborting(h) = \{B\}$, $live(h) = \langle A,\ \ \mathsf{list.insert}\ (3)\ \rangle$ and all other functions produce empty sets.

We now present our main result:

Theorem 5.3. *(Main Theorem) For any STM system that obeys the correctness rules, the history of committed transactions is strictly serializable.*

Proof. For brevity, we say a history is *proper* if it follows the correctness rules. We will inductively prove the following two properties, which imply the main theorem.

Property 5.1. *For any proper history h, $committed(h)$ is strictly serializable.*

Property 5.2. *For any proper history h, m and n commute for all $m, n \in live(h) \cup stale(h)$.*

An immediate consequence of Property 5.2 and Lemma 5.2 is that any $m \in live(h)$ commutes with n, n^{-1} where $n \in stale(h)$, $n^{-1} \in reverting(h)$ are corresponding methods and inverse methods from the same transaction.

Let h be a proper history. If h is empty, both the properties hold vacuously. We now show that for any subsequent event, both properties remain invariant. Let h_p be a history and h be the history defined by h_p followed by the next event as follows:

- $\langle T\ \ \mathsf{init}\ \rangle$: There are no new committed transactions nor changes to $live(h)$ or $stale(h)$.
- $\langle T\ \mathsf{commit}\rangle$: Using Property 5.2, it is easy to check that the state defined by h is the same as the state defined by
  $h_{alt} \equiv h|(committed(h) \setminus \{T\})$
  $\cdot\ h|T \cdot\ h|(active(h) \cup aborting(h))$
  It is also easy to check that we need not consider $h|aborted(h)$ in this history by the inverse property. Thus, Property 5.1 is invariant because there exists a legal order h_{alt}, which is consistent with the commit order of h and is strictly serializable with respect to the newly committed transaction.
  Additionally, since $live(h) \subseteq live(h_p)$ and $stale(h) = stale(h_p)$ Property 5.2 is also invariant.
- $\langle T\ \mathsf{abort}\rangle$: There are no new committed transactions. $stale(h) = stale(h_p) \cup live(h_p|T)$, $live(h) \subseteq live(h_p)$, $reverting(h) = reverting(h_p)$.
- $\langle T\ \mathsf{aborted}\rangle$: There are no new committed transactions. $stale(h) \subseteq stale(h_p)$, $live(h) = live(h_p)$, $reverting(h) \subseteq reverting(h_p)$.
- $\langle T,\ \mathsf{xm}(v)\ \rangle$: No new committed transactions. By Rule 2, we are assured that there does not exist $T' \in active(h)$ such that $(T', T) \in precedes(h)$. Therefore there does not exist $n \in live(h) \cup stale(h)$ such that m and n do not commute. Thus Property 5.2 is invariant.

- $\langle T,\ \mathsf{xm}^{-1}(v)\ \rangle$: No new committed transactions. By Rule 3, we know that $\exists m \in live(h_p)$ such that m is the inverse of m^{-1}. By the lemma, $\forall n \in live(h_p)$, m commutes with n.
- $\langle T, \mathsf{xm}(v)\ \rangle$ (where $T \in committed(h_p) \cup aborted(h_p)$): No new committed transactions. By Rule 4 such invocations must be disposable and by Lemma 5.1 must commute with any method $m \in live(h) \cup stale(h)$. Thus Property 5.2 is invariant.
- $\langle T, v \rangle$: No new committed transactions nor changes to $live(h)$ or $stale(h)$.

□

Theorem 5.4. *(Aborted Transactions) For any STM system that obeys all three correctness rules, any history defines the same state as a history with aborted transactions removed.*

Proof. Our proof follows the same format as that of Theorem 5.3, but only the $\langle T\ \mathsf{aborted}\rangle$ event is non-trivial. Let h be some history for which the theorem is invariant, and now consider $h \cdot \langle T\ \mathsf{aborted}\rangle$.

It is easy to check that in history h there exists a pair of method call $m \in stale(h)$ and inverse $m^{-1} \in reverting(h)$ such that no other inverse $n^{-1} \in reverting(h)$ precedes the occurrence of m^{-1}. Using commutativity, we can find an alternate history consistent with the commit order, which defines the same state as h such that the following events are adjacent:

$$\cdots \langle T, \mathsf{xm}(v)\ \rangle \cdot \langle T, r\rangle \cdot \langle T\ \mathsf{abort}\rangle \langle T, \mathsf{xm}^{-1}(v')\ \rangle \cdot \langle T, r'\rangle \cdots$$

By definition of inverse, this history defines the same state as:

$$\cdots \langle T\ \mathsf{abort}\rangle \cdots$$

Following an inductive argument, we can repeat this process until all inverse operations have been removed. By rule 3, T performs an inverse for each of its method calls. Thus, there exists a history that is consistent with commit order, which defines the same state as $h|\{\forall T'.T' \neq T\}$. □

6. Related Work

Transactional memory [14] has gained momentum as an alternative to locks in concurrent programming. This approach has been investigated in hardware [1, 8, 14, 24, 23], in software [9, 10, 13, 21, 31], and in schemes that mix hardware and software [6, 28].

All existing STM techniques determine conflicts on the basis of read/write conflicts: if one transaction writes an object (or memory location) read by a concurrent transaction, then those transactions conflict. We argue that read and write operations are too crude as they elide the inherent *semantics* of the object being mutated, and thus are prone to, in the words of Harris [11], *benign* conflicts.

Open nested transactions [25] (ONT) have been proposed as a way to implement highly-concurrent transactional objects. In ONT, a nested transaction can be designated as *open*. If an open transaction commits, its effects immediately become visible to all other transactions. The programmer can register handlers to be executed when transactions enclosing the open transaction commit or abort.

Although transactional boosting and ONT both use abstract method-based locks, the two approaches are starkly divergent. Open nested transactions are a *mechanism*, not a methodology. By themselves, they provide no guidance as to how they can be used correctly. As noted by Ni *et al.* [25], care must be taken to avoid deadlocks in abort handlers, and to avoid unexpected behavior that may occur if an open nested transaction's read set intersects a parent's write set. In transactional boosting, however, inverse methods called by an aborting transaction cannot deadlock with other ongoing transactions because the aborting transaction acquires no additional locks. Moreover, because the base object's methods are not called in a nested transaction, read/write conflicts between parents and open children do not arise.

When comparing boosting to ONT, it is important to distinguish between different kinds of potential deadlocks. Like McRT [28], transactional boosting uses two-phase locking, which is vulnerable to deadlock, and can be avoided by timeouts. Deadlocks on lock acquisition are qualitatively different from the deadlocks that arise in ONT because it is possible to recover from lock acquisition deadlock by aborting and retrying a transaction that times out. By contrast, in ONT, there is no way to recover if an aborting transaction deadlocks while executing its abort handler.

Open nested transactions also have certain limitations in expressive power. It is unclear how to map open nested transactions onto algorithms that use techniques such as lock coupling, where synchronization regions are not properly nested. Moreover, because transactions enforce isolation, there is no possibility of thread-level concurrency between open nested transactions, and therefore no way to exploit existing thread-level synchronization libraries. Finally, using open nested transactions to construct a highly-concurrent transactional hash table requires reimplementing the hash table itself, while transactional boosting would treat the hash table as a black box.

Harris and Stipić [11] recently proposed "abstract nested transactions" (ANTs). Although ANTs also aim to reduce benign conflicts, the approach is substantially different from ours. Unlike our methodology, memory access during an abstract nested transaction is logged, and this log is used to detect conflicts. As a commit-time optimization, ANT re-evaluates expressions (closures) which are part of conflicting data access to ensure the expression value has not changed since it was first computed. While this approach is well-suited to mostly-functional languages, it is unclear how well it could be used in a language where re-evaluating closures

may have side effects. Transactional boosting does not need to track memory access as executing transaction will not conflict with any other.

Our formal model of event histories is similar models used by Weihl [36] for atomicity in distributed systems, and by Herlihy and Wing's for linearizability [16], although a number of changes were needed to model explicit compensating actions.

Many of the mechanisms used by transactional boosting are well-known from other contexts. Bernstein [4] first used commutativity in the parallel programming community to determine whether two successive portions of a program can be performed concurrently. Commutativity was also used by Steele [7] for optimistic concurrency and by Diniz and Rinard [27] for static analysis of method commutativity. Weihl [34, 15] showed that commutativity can be used for concurrency control in distributed databases. Finally, work has also been done on commutativity-based locking protocols [3, 18, 33], including protocols that use method arguments [29] and return values [35]. This work has shown that concurrently executing only operations that commute is known to yield histories that are strictly serializable. The second principle is *operation logs*. Unlike traditional STM systems which log individual memory accesses, our methodology logs operations, and uses inverse operations to recover from failures. Operation logs have been studied extensively in the database community [20, 30]. Our contribution is to extend these well-understood techniques to software transactional memory.

Kulkarni *et al.* [19] describe Euclid, a system that exploits commutativity and inverses for efficient thread-level speculation.

Moravan *et al.* [24] and Zilles and Baugh [37] observe that constructs similar to open nested transactions can be used to allow transactions to execute non-transactional code, such as system calls.

The pipelining example raises the important question of *conditional transactional synchronization*. The transactional semaphore was enough to satisfy our needs. Nevertheless, the `retry()` construct from STMHaskell [10] provides an elegant and general mechanism for incorporating conditional synchronization into transactional computation. Supporting `retry()` has a pervasive effect on any STM implementation, while a boosted implementation of transactional semaphores does not.

7. Conclusion

We have presented a methodology for translating a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. Given only a well-defined specification of a black-box object, transactional boosting allows concurrent threads to interact with the object within a transaction, and no log of memory access is needed neither for conflict detection nor for recovery from aborted transactions. We have shown that for many workloads, the additional run-time burden of transactional boosting is far offset by the performance gain of eliminating memory access logging. Finally, our approach guarantees that the history of computation remains strictly serializable in the presence of arbitrarily many concurrent transactions and abortions.

There are a number of ways in which transactional boosting can be extended. It could encompass STMs based on nested transactions using techniques similar to those employed by LogTM [24]. Transactions could be extended to encompass multiple threads, using abstract locks for transactional synchronization, and relying on the base object for thread-level synchronization. In a hybrid system that combines small hardware transactions with STM, one could implement base method calls as hardware transactions, using boosting to managed long-lived software transactions.

References

[1] Scott Ananian, Krste Asanovic, Bradley Kuszmaul, Charles Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, February 2005.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

[3] C. Beeri, P. A. Bernstein, N. Goodman, M. Y. Lai, and D. E. Shasha. A concurrency control theory for nested transactions (preliminary report). In *Proceedings of the 2nd annual ACM symposium on Principles of distributed computing (PODC '83)*, pages 45–62, New York, NY, USA, 1983. ACM Press.

[4] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, 1966.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter Chapter 13: Red-Black Trees, pages 273–301. MIT Press and McGraw-Hill, 2001.

[6] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 336–346, New York, NY, USA, 2006. ACM Press.

[7] Jr. Guy L. Steele. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '90)*, pages 218–231, New York, NY, USA, 1990. ACM Press.

[8] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *SIGARCH Computer Architecture News*, 32(2):102, 2004.

[9] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN con-*

ference on Object-oriented programing, systems, languages, and applications (OOPSLA '03), pages 388–402. ACM Press, 2003.

[10] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '05)*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[11] Tim Harris and Srdjan Stipić. Abstract nested transactions. http://research.microsoft.com/ tharris/papers/2007-ant.pdf, 2007.

[12] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing (PODC '03)*, pages 92–101. ACM Press, 2003.

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300. ACM Press, 1993.

[15] Maurice P. Herlihy and William E. Weihl. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS '88)*, pages 201–210, New York, NY, USA, 1988. ACM Press.

[16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[17] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, 1996.

[18] Henry F. Korth. Locking primitives in a database system. *J. ACM*, 30(1):55–79, 1983.

[19] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and Paul Chew. Optimistic parallelism requires abstractions. In *PLDI 2007*, 2007.

[20] David B. Lomet. Mlr: a recovery method for multi-level systems. *SIGMOD Rec.*, 21(2):185–194, 1992.

[21] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. Technical Report TR 868, Computer Science Department, University of Rochester, May 2005.

[22] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM Press, 2002.

[23] Kevin E. Moore, Mark D. Hill, and David A. Wood. Thread-level transactional memory. Technical Report CS-TR-2005-1524, Deptartment of Computer Sciences, University of Wisconsin, March 2005.

[24] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 359–370, New York, NY, USA, 2006. ACM Press.

[25] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07)*, pages 68–78, New York, NY, USA, 2007. ACM Press.

[26] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[27] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, November 1997.

[28] Bratin Saha, Ali-Reza Adl-Tabatabai, Rick Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm. In *PPoPP*, 2006.

[29] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.*, 2(3):223–250, 1984.

[30] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[31] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing (PODC '95)*, pages 204–213. ACM Press, 1995.

[32] http://research.microsoft.com/research/downloads/Details/6cfc842d-1c16-4739-afaf-edb35f544384/Details.aspx.

[33] W. E. Weihl. Specification and implementation of atomic data types. Technical report, M.I.T. Laboratory for Computer Science, Cambridge, MA, USA, 1984.

[34] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[35] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, 1985.

[36] William E. Weihl. Data-dependent concurrency control and recovery (extended abstract). In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83)*, pages 63–75, New York, NY, USA, 1983. ACM Press.

[37] Craig Zilles and Lee Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.