

# Concurrent Non-commutative Boosted Transactions

Eric Koskinen  
Computer Laboratory  
University of Cambridge  
ejk39@cam.ac.uk

Maurice Herlihy  
Computer Science Department  
Brown University  
mph@cs.brown.edu

*This paper may be considered for the Best Student Paper award.  
The student is Eric Koskinen.*

# Concurrent Non-commutative Boosted Transactions

Eric Koskinen

Computer Laboratory  
University of Cambridge  
ejk39@cam.ac.uk

Maurice Herlihy

Computer Science Department  
Brown University  
mph@cs.brown.edu

## Abstract

Traditional software transactional memory systems implement synchronization and recovery by tracking memory access. In recent work, we introduced transactional boosting, a methodology whereby performance is improved by forgoing read/write sets and relying instead on data structure commutativity and abstract locks for synchronization.

In this paper, we describe a method for concurrent execution of non-commuting operations from distinct transactions. Abstract locks are speculatively passed from one transaction to the next, and dependencies are created, enforcing certain commit orders. We present an efficient implementation, and describe novel techniques for performing recovery lazily and detecting cyclic dependencies.

We show that the performance gain for long-lived transactions can offset the cost of tracking dependencies. Moreover we quantify the somewhat counter-intuitive discovery that dependent transactions afford scalability under increased contention.

## 1. Introduction

Recently, software transactional memory (STM) has emerged as a compelling alternative to traditional mutual exclusion techniques such as locks. In STM, programmers access shared memory inside of transactions, and the STM implementation ensures atomicity and sound recovery. Atomicity is typically achieved by detecting *conflicts* between concurrent transactions and aborting one of them. As a transaction executes, *read* and *write* locations are tracked. Two transactions conflict if the intersection of one transaction's write set with either of another transaction's sets is nonempty.

Unfortunately, conflict detection on the basis of read/write sets rejects some serial histories. Often, all of one transaction's memory accesses precede those of another even though both execute concurrently. As an example, consider the following interleaving of transactions:

$$\begin{aligned} T_1 : & \text{write}(0x42), & T_2 : & \text{read}(0x42), & T_1 : & \text{write}(0x68), \\ T_2 : & \text{read}(0x68), & T_2 : & \text{write}(0x68), & T_1 : & \text{write}(0x94) \end{aligned}$$

Traditional STM implementations would detect a conflict between  $T_1$  and  $T_2$  since they access the same memory locations, and abort one or both of them.

By contrast, a *dependence-aware* transactional implementation can track which memory locations are shared between transactions and the order in which they are shared. For any given memory location, if all of the write operations of  $T_1$  precede the read operations of  $T_2$  then  $T_2$  can commit provided that it is delayed until after  $T_1$  commits and aborted if  $T_1$  aborts.

In recent work [6] we introduced transactional boosting, a methodology for building highly-concurrent transactional objects. Boosting exploits known semantics (roughly which methods *commute* and method *inverses*) of linearizable objects. Synchronization is then performed on the basis of abstract locks rather than read/write sets, allowing transactional access to linearizable base objects with the performance on the same order as the base objects themselves.

In this paper, we show how performance of boosted transactions can be improved if more concurrency is enabled by passing abstract locks. Thus, transactions that would have previously been blocked can proceed speculatively. Dependencies among transactions are tracked to ensure that they commit in the correct order.

Our work includes the following contributions:

- We present a technique for passing an abstract lock from one transaction to another, and an infrastructure to support dependent boosted transactions in Section 3.
- Our method involves a novel approach to recovery: inverse operations are associated with abstract locks, and invoked lazily when future transactions acquire the locks. This technique is described in Section 4.
- In Section 5, we discuss a novel mechanism for accurately detecting and avoiding cyclic dependencies, similar to the DREADLOCKS algorithm [7].
- We evaluate our technique in Section 8 using our boosted version of TL2. We show that performance can be improved for long-lived transactions and, somewhat surprisingly, passing abstract locks improves scalability under workloads with high contention.

## 2. Transactional Boosting

Earlier [6] we showed how highly-concurrent linearizable base objects can be made transaction-aware without tracking read/write sets. This *transactional boosting* technique is discussed extensively elsewhere, but for completeness we summarize the basic ideas here.

Start with a thread-safe linearizable object, for example, the `ConcurrentSkipListSet` class of `java.util.concurrent`. Two method calls *commute* if applying them sequentially in either order leaves the object in the same (abstract) state, and both calls return the same values. In this example, calls to `add(x)` and `remove(y)` commute for distinct  $x$  and  $y$ . One method call is the *inverse* of another if applying the second after the first restores the object’s original state. In this example, a call to `add(x)` that returns *true* (meaning the value was not already there) has as inverse `remove(x)`.

Each method call has an associated two-phase *abstract lock*, which may depend on the method’s arguments. In this example, we associate an abstract lock with each key value: calls to `add(x)` and `remove(y)` do not conflict because, since they commute, they acquire distinct locks. By contrast, calls to `add(x)` and `remove(x)` do conflict, and, in fact, they do not commute. If a transaction aborts, it is necessary to undo its effects. For this reason, each time we call an object method, we log (but do not execute) a call to its inverse. In this example, a call to `add(x)` that returns *true* logs a call to `remove(x)`. (For this class, a call to `add(x)` that returns *false* does not modify the set, and needs no inverse.)

As shown elsewhere [6], boosting provides a substantial performance improvement through enhanced concurrency (by eliminating false conflicts), and through a smaller footprint (logging function pointers instead of read-write sets).

This paper describes a novel extension to transactional boosting, in which active transactions release abstract locks to other active transactions. In the next section we describe the core idea. We discuss recovery in Section 4 and an algorithm for detecting cyclic dependencies in Section 5.

## 3. Passing Abstract Locks

Consider the following example where two transactions acquire the same abstract lock, and perform operations on a boosted skip list.

<pre> T1: atomic {   with(AbstractLock(3)) {     if ( skipList . insert (3))       self .onAbort.add( (λ () skipList .remove(3)) )   }   BigComputation(); } </pre>	<pre> T2: atomic {   with(AbstractLock(3)) {     if ( skipList .remove(3))       self .onAbort.add( (λ () skipList . insert (3)) )   }   BigComputation(); } </pre>
---	---

T2 depends on T1 via lock 3

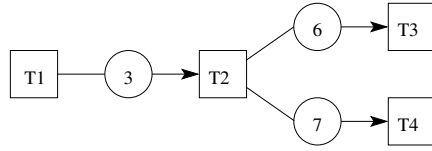


Figure 1: Passing abstract locks, creating dependencies.

The **with** block in each transaction means that an abstract lock is acquired at the start of the block, and released, not when the block is exited, but when the transaction commits. After acquiring the abstract lock, each transaction performs an operation on the shared, linearizable `skipList`, and, if the state was changed, logs the inverse. Then each transaction engages in a long computation that does not involve the skip list.

The two transactions' operations do not commute, so (following the rules for boosting) they must acquire the same abstract lock. Abstract locks are two-phase, and are held until commit time, implying that the transactions must run serially. Here, however, the two transactions could be pipelined. If one transaction speculatively hands off its skip list lock to the other, then as long as dependencies are respected, the two transactions can overlap their computations.

**Passing Abstract Locks.** The key idea of this paper is to allow non-commuting transactions to proceed in parallel by allowing one active transaction speculatively to release abstract locks to another. If transaction  $T_1$  hands off an abstract lock to  $T_2$ , then  $T_2$  must respect this dependency by following two rules:

1.  $T_2$  cannot commit until *after*  $T_1$  has committed.
2. If  $T_1$  aborts,  $T_2$  must also abort.

If  $T_1$  hands off an abstract lock to  $T_2$ , then  $T_2$  can observe  $T_1$ 's tentative effects. The first rule ensures that  $T_1$  is serialized before  $T_2$ , which is consistent with their data dependencies. The second rule ensures that if  $T_1$  aborts, then  $T_2$ , which may rely on  $T_1$ 's speculative updates, is also rolled back.

Abstract locks are passed from a transaction  $T_1$  holding a lock  $\ell$  to another transaction  $T_2$  who wishes to acquire it. Our mechanism allows  $T_2$  to effectively “steal”  $\ell$  from transaction  $T_1$ . Each lock is a tuple  $\ell = \langle T, H \rangle$  where  $T$  is the transaction which currently holds  $\ell$ , and  $H$  is a boolean flag indicating whether the lock can be handed off.

The algorithm is as follows:

1. Initially, any lock  $\ell = \langle null, false \rangle$  is free.
2.  $T_1$  acquires the lock  $\ell$ , transforming the state to  $\langle T_1, false \rangle$
3. When  $T_1$  has finished performing its operation on the base linearizable object, it may mark the lock as free, transforming it to the state  $\langle T_1, true \rangle$ . This is the default behavior of a **with**(AbstractLock) clause.
4. Another transaction  $T_2$  interested in acquiring  $\ell$  may only do so if the hand off flag is set:  $\ell = \langle T_1, true \rangle$ . Once the lock is passed to  $T_2$ , the state becomes  $\langle T_2, false \rangle$ .

A dependency of transaction  $T_1$  on  $T_2$  is created whenever a lock transitions from  $\langle T_2, true \rangle$  to  $\langle T_1, false \rangle$ . The hand off flag ensures that when a lock is acquired by a transaction, the transaction has the opportunity to perform its operations before another transaction acquires the lock.

If transaction  $T_1$  acquires lock  $\ell$ , performs an operation, and raises the hand off flag, then it must *re-acquire* the lock before executing any subsequent operations associated with  $\ell$ . It will then detect any cyclic dependencies, as discussed in Section 5.

An example of dependent transactions is given in Figure 1. Here, transaction  $T_2$  has been handed lock 6 from  $T_3$  and lock 7 from  $T_4$ . Subsequently, transaction  $T_1$  has acquired lock 3 from  $T_2$ . Note that dependencies are acyclic; our algorithm for detecting (and avoiding) cyclic dependencies is discussed in Section 5.

## 4. Lazy Recovery

As we discuss elsewhere [6], when a boosted transaction aborts it must perform recovery by applying inverse operations. We now discuss our novel approach to recovery, which is more conducive to dependent transactions.

**Per-Lock Inverse Logging.** The first insight is that inverse operations can be logged on a per-lock rather than per-transaction basis. When a transaction acquires an abstract lock and executes a method call, it then appends the corresponding inverse operation with the lock. Recovery for the above example is as follows:

```
atomic {  
  with(AbstractLock(3)) {  
    if ( skipList . insert (3))  
      AbstractLock(3).appendInv( (λ () skipList . remove(3)) )  
  }  
  BigComputation();  
}
```

Here the abstract lock is acquired and the `skipList` insertion operation is performed as usual. However, if the operation is successful, the removal inverse is appended to the `AbstractLock(3)` log rather than transaction-local storage.

Often (particularly with collections) a boosted method call involves acquiring a single abstract lock. However, in other cases one might acquire two or more abstract locks for a single operation. The decision then becomes: which abstract lock log should store the inverse? In these cases, one of two rules typically applies:

1. *Decompose the inverse.* In some situations, the logical operation spans two abstract locations, and recovery can be accomplished with multiple operations which each correspond to single abstract locations. Consider moving an element from key A to key B in a hash table using the method `HT.move(srcKey,destKey)`. Per-key inverses can be created with `HT.set(key,value)`.
2. *Compound abstract locks.* To specify some abstract locations, multiple abstract indices may be needed. For example, one might acquire two abstract locks before accessing a two-dimensional array. Instead, an abstract lock can be defined which is a tuple of indices, and then only a single lock is needed.

Per-lock logs require more care than per-transaction logs. When abstract locks are too general (as in the 2D array example) they can be combined into tuple-locks. When abstract locks are too specific (as in the hash table example) they can be broken up into multiple inverses. On the other hand per-lock logs enables lazy recovery, which is particularly useful for dependent transactions.

**Lazy Recovery.** With per-lock inverse logs, the commit and abort procedures for transactions are trivial. Transactions simply change their status from *active* to either *committed* or *aborted*. Inverse operations are postponed until another transaction tries to acquire the same abstract lock.

When arriving at an abstract lock, if a transaction finds a log of inverse operations, then it enters a delegated recovery phase. Before proceeding, the transaction scans backwards through the log of inverses. Inverses which correspond to committed transactions can simply be discarded, whereas those of aborted transactions must be invoked. Note that this recovery is not performed when a transaction is passed the lock from another transaction.

### 4.1 Passive Mode

When a chain of dependent transactions abort, it typically indicates that their operations are too entangled for dependencies to be worthwhile. Instead, it may be beneficial to resort to pessimistic abstract locks which block. Such *passive* transactions follow two rules:

1. A passive transaction must never acquire an abstract lock from a live transaction.
2. A passive transaction must never release an abstract lock until it commits or aborts.

In effect, whenever a passive transaction acquires a lock  $\ell$ , all other transactions contending for  $\ell$  do so passively. Other transactions cannot release  $\ell$  to the passive transaction until commit-time; other transactions must block

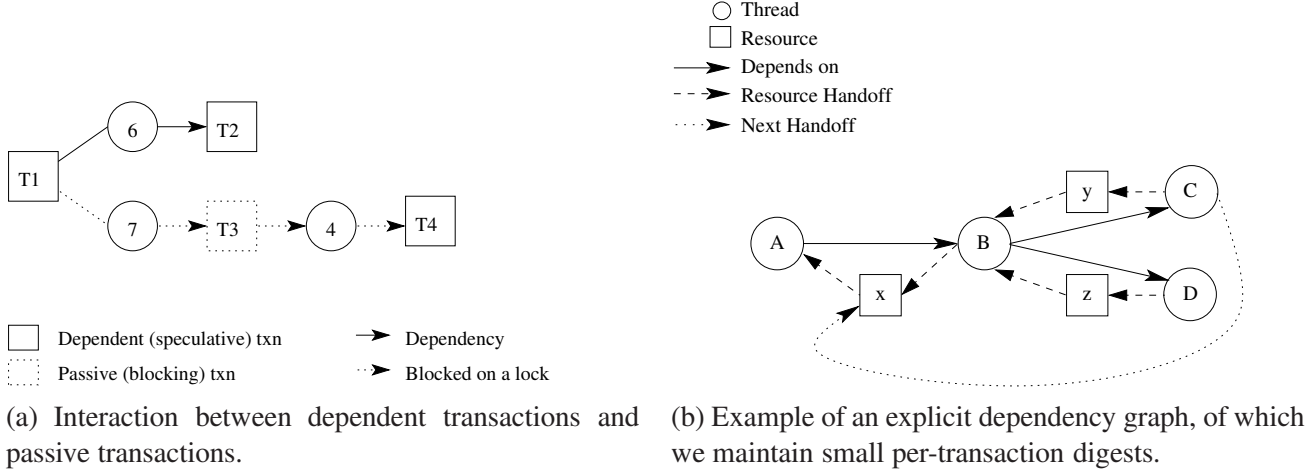


Figure 2: Diagrams.

if the passive transaction acquires  $\ell$  first. An example of interacting dependent and passive transactions is illustrated in Figure 2(a).

## 5. Cyclic Dependencies

If active transactions pass abstract locks to one other, then a cyclic dependency exists. Transactions linked by cyclic dependencies cannot commit, since there is no serial order in which they can commit. We now discuss how to detect and avoid such dependencies.

To detect cyclic dependencies, we adapt the DREADLOCKS algorithm [7] for detecting deadlock among transactions that acquire two-phase locks. Nevertheless, detecting cyclic dependencies is more difficult than detecting deadlocks. Deadlock occurs when threads are actively acquiring locks, and can be detected *before* it occurs. By contrast, cyclic dependencies can be detected only after a lock has been handed off. This distinction means that the DREADLOCKS algorithm for detecting deadlocks requires non-trivial modifications to detect cyclic dependencies.

**Digests.** To detect cyclic dependencies, we track the transitive closure of which transactions have handed off abstract locks to other transactions. We track dependencies with transaction-local transitive closures called *digests*.

Let  $T \in \mathbb{N}$  represent a transaction and  $R \in \mathbb{N}$  represent a resource (abstract lock). Define  $owner : \mathbb{N} \rightarrow \mathbb{N}$  to map resources to the transactions which currently hold them.

**Definition 5.1.** Transaction  $T$ 's digest, denoted  $\mathcal{D}_T$ , is the set of other transactions upon which  $T$  depends.

1. If transaction  $T$  has no dependencies,  $\mathcal{D}_T = \{T\}$
2. If  $T$  is dependent on the set of other transactions  $\mathbb{T}$ ,  

$$\mathcal{D}_T = \{T\} \cup \bigcup_{T_i \in \mathbb{T}} (\{T_i\} \cup \mathcal{D}_{T_i}).$$

Informally, each transaction's digests initially contains only that transaction. When transaction  $A$  acquires an abstract lock from active  $B$ , then  $A$ 's digest must include  $B$ . Moreover,  $A$  is now (indirectly) dependent upon each of  $B$ 's dependencies. It follows that  $A$ 's digest also includes  $\mathcal{D}_B$ .

Cyclic dependency detection is accomplished when a transaction tries to acquire a resource as follows:

**Definition 5.2.** Transaction  $T$  detects a cyclic dependency when acquiring resource  $R$  if  $T \in \mathcal{D}_{owner(R)}$ .

Consider the cyclic dependency graph given in Figure 2(b), ignoring the dotted line for the moment. Transaction  $A$  has acquired the lock for  $x$  from transaction  $B$ , and thus  $A$  depends on  $B$  (as shown by a solid

line). Similarly,  $B$  has acquired lock  $y$  from  $C$  and  $z$  from  $D$ , and thus depends on both  $C$  and  $D$ . Following the above rules, here are their digests:

$$\mathcal{D}_D = \{D\} \quad \mathcal{D}_C = \{C\} \quad \mathcal{D}_B = \{B, C, D\} \quad \mathcal{D}_A = \{A, B, C, D\}$$

Now suppose  $C$  tries to acquire lock  $x$  held by  $B$ .  $C$  first examines  $\mathcal{D}_B$ . Since  $C \in \mathcal{D}_B$ ,  $C$  detects the cyclic dependency.

**Recovery.** Cyclic dependencies can be resolved by simply aborting both transactions. Unfortunately, this is a potential source of degraded performance. Not surprisingly, workloads in which cyclic dependencies are frequent are not good candidates for building dependent transactional boosting.

**Digest Propagation.** Here is why detecting cyclic dependencies is harder than detecting deadlocks. Suppose transaction  $B$  acquires a lock held by  $A$ , thus becoming dependent on  $A$ . Following the above rules,  $\mathcal{D}_A$  is passed to  $B$ . Then suppose that  $A$  acquires a lock held by  $C$ . Now, transaction  $B$  may be busy executing arbitrary transactional code, and unable to propagate the newly updated  $\mathcal{D}_A$  (which includes  $C$ ) to its own digest (and so on down the chain). Thus, if  $C$  acquires a lock held by  $B$  (or any other transaction which transitively depends on  $B$ ) the cyclic dependency will not be detected.

To avoid the need for complex propagation algorithms, we impose some simple restrictions: once transaction  $A$  is depended upon by another transaction,  $A$  no longer acquires abstract locks from other transactions. Thus, dependencies grow only in one direction.

**Correctness.** The digest  $\mathcal{D}_A$  for a transaction  $A$  is a mutable set. If  $B$  is in  $\mathcal{D}_A$ , then a call to  $\mathcal{D}_A.contains(B)$  returns *true*. If  $B$  is not in  $\mathcal{D}_A$ , the call can return either *true* or *false*. A digest is allowed *false positives*, indicating that a transaction is present when it is not. *False negatives* are prohibited (unlike in DREADLOCKS [7]). False positives may cause unnecessary aborts which is acceptable if rare, but false negatives lead immediately to mutually dependent transactions.

Digests have two mutator methods:  $\mathcal{D}_T.setSingle(A)$  sets the digest to  $\{A\}$ , and  $\mathcal{D}_T.setUnion(A, \mathcal{D}_S)$  sets the digest to  $\{A\} \cup \mathcal{D}_S$ , where  $A$  is a transaction and  $\mathcal{D}_S$  another digest.

**Theorem 5.1.** *Any cyclic dependency will be detected by at least one transaction.*

*Proof.* Consider a potential cycle of transactions, where if  $A$  hands off a lock to  $B$ , the cycle is complete. First,  $B$  examines  $\mathcal{D}_A$ , which must already contain  $A$  and all the transactions it depends on. So if  $A$  transitively depends on  $B$ , then  $B$  is in  $\mathcal{D}_A$ , so  $B$  will detect a cyclic dependency, aborting all the transactions in the cycle.  $\square$

**Remark: Representation.** In prior work [7] we analyze how dependency digests can be represented using bit-vectors or Bloom filters [2]. Here, we use 64-bit Bloom filters to represent cyclic dependency digests.

**Remark: Deadlock.** Note that detecting cyclic dependencies does not supplant deadlock detection: unfortunately deadlock is not simply a special case of cyclic dependencies. As we discuss in Section 7, we also require DREADLOCKS [7] to detect deadlock.

## 6. Discussion

There are advantages to passing abstract locks between boosted transactions. The principle *performance* advantage is that certain transactions can be pipelined. By handing off locks, conflicting transactions can overlap execution. This is particularly effective for “phased” transactions which operate a sequence of disjoint objects.

A second, more surprising advantage is one of *scalability*. As we show in Section 8.1, workloads of increased contention degrade slower when abstract locks are passed between transactions. Under high contention, deadlock is common in transactional boosting and leads to many aborts. However, by passing an abstract lock, some of these aborts can be avoided. The result is that passing abstract locks affords the benefit of transactional boosting without the cost of poor scalability under contention.

Of course, there are transactions for which building dependencies does not make sense. A transaction that uses  $x$ , runs a long computation, and uses  $x$  again should not hand off the lock for  $x$  after the first phase,

because it will create a cyclic dependency when it tries to re-acquire the lock for  $x$ . A sensible way to approach this problem is to keep track of dependency-related aborts, and to restart a transaction aborted in this way in passive mode. It is also possible for users to insert pragmas suggesting which locks may profitably be handed off and which may not.

A *zombie* transaction is an active transaction that is certain to abort. One disadvantage of dependent boosted transactions is that zombie transactions are not guaranteed to see consistent states. Suppose there is an invariant relating object  $x$  and  $y$ . Normally T1 locks  $x$ , modifies it, temporarily violating the invariant, locks  $y$ , and modifies it, restoring the invariant. If T1 hands off the lock for  $x$  to T2 before it acquires the lock for  $y$ , then T2 may see the invariant violated. Here, too, user-provided pragmas can restrict possible inconsistencies.

Boosted objects can be *privatized*. Operations on a privatized object commute with no other transaction's operations. Privatization can thus be achieved, for example, by acquiring a per-object universal lock. The recovery method described in this paper adds a single caveat: before an object can be privatized, all outstanding lazy inverse operations must be executed.

## 7. Implementation

We implemented dependent boosted transactions as an extension of our transactional boosting implementation, which is based on TL2 [5]. Also essential to our implementation is deadlock detection for which we use our detection scheme, DREADLOCKS [7]. Our implementation consists of a few key components:

### 7.1 Lock Data Structure

Each abstract lock is implemented as a structure of two fields, corresponding to the lock tuple  $\ell = \langle T, H \rangle$  defined in Section 3. An owner field ( $T$ ) that is atomically updated with a pointer to the current transaction when the lock is acquired. We reserve the least significant bit as a *hand off flag* ( $H$ ) to indicate whether the lock can be passed to another transaction. This flag prevents a lock from being handed off immediately after it is acquired. Transactions instead acquire the lock, then perform their data structure operation before raising the flag to allow other transactions to acquire the lock.

Though we expect that transactions will typically raise the hand off flag immediately after they complete the data structure operation, they may further delay raising the flag or decline entirely to raise the flag. We have found that avoiding flag raising is beneficial when the transaction is expected to commit shortly, or the lock is unlikely to be of interest to other transactions.

Also associated with each lock is a list of inverse operations. When transactions acquire a lock and perform an operation, they log the inverse operation with the corresponding lock (as described in Section 4). We discuss how these inverses are invoked in Section 7.4.

### 7.2 TTAS Lock

For dependent boosted transactions, abstract locks are acquired via a *test-and-test-and-set* (TTAS) spin lock. When a boosted transaction attempts to acquire an abstract TTAS lock, it executes the algorithm in Figure 3(a).

Recall from Section 3 that an abstract lock is a tuple  $\ell = \langle T, H \rangle$  where  $T$  is the transaction which currently holds  $\ell$ , and  $H$  is a boolean flag indicating whether the lock can be handed off. On most architectures transaction state is allocated on word boundaries, so this tuple can be implemented in a single word containing a pointer to  $T$  and using the least significant bit for  $H$  (Lines 4-6). The remainder of the TTAS algorithm is a large case analysis.

First, the transaction checks to see if the lock can be trivially acquired since it is not held by any other transaction (Line 7). In the event that the lock is already held by the current transaction, the transaction simply returns successfully (Line 8). If the lock is held by an aborted or committed transaction, then it can be acquired (Line 9). The lock may also be held by a transaction which is willing to hand it off (Line 10), in which case if the transaction is not in passive mode, then a check for cyclic dependencies is made (Line 11) and the thread's digest of dependencies is updated. If none of the above cases apply, then the transaction is truly spinning on the lock. The transaction executes the deadlock detection algorithm (Line 19) called DREADLOCKS [7]. It is

```

1 void TxLock(Thread Self, XLock lock) {
2   while(true) {
3     while(true) {
4       Txn owner = lock.owner;
5       bool HandOffFlag = (owner & 0x1);
6       owner &= 0xFFFFFFFF;
7       if (!owner) break;
8       if (owner == Self.CurrTxn) return;
9       if (owner.Aborted || owner.Committed) break;
10      if (HandOffFlag && Self.AllowDepend) {
11        if (owner.CyclicDigest & Self.CyclicDigestID) {
12          owner.Aborted = true;
13          Self.abort();
14        }
15        Self.Dependencies.Append(owner);
16        Self.CyclicDigest |= owner.Thread.CyclicDigest;
17        break;
18      }
19      if (owner.Thread.DreadDigest & Self.DreadID) {
20        Self.DreadDigest.Clear();
21        Self.Abort();
22      }
23      Self.DreadDigest |=
24        Self.DreadID | owner.Thread.DreadDigest;
25
26      if (Self.Aborted)
27        Self.Abort();
28    }
29    if (CAS(lock.owner,owner,Self.CurrTxn)) {
30      Self.DreadDigest = Self.DreadID;
31      break;
32    }
33  }
34  lock.Flush(Self);
35  return;
36 }

```

(a) A TTAS Abstract Lock with Lazy Release.

```

1 void TxLockCanSteal(Thread Self, XBlock lock) {
2   while(1) {
3     Txn selfSteal = Self.CurrTxn | 0x1;
4     if (CAS(lock.owner,Self.CurrTxn, selfSteal ))
5       break;
6   }
7 }

```

(b) Using a CAS to raise the hand off flag

```

1 void TxCommit (Thread Self) {
2   if (Self->AbortRequested)
3     TxAbort(Self);
4
5   foreach (Txn : Self->Dependencies) {
6     Self.CurrTxn.DreadDigest = Self.DreadID;
7     while(Txn.Active) {
8       if (Txn.DreadDigest.Contains(Self.DreadID))
9         Self.Abort();
10      Self.CurrTxn.DreadDigest |= Txn.DreadDigest;
11    }
12    if (Txn.Aborted)
13      Self.Abort();
14  }
15  Self.Dependencies.Clear();
16
17  if (Self.CurrTxn.Aborted)
18    Self.Abort();
19 }

```

(c) Committing an dependent transaction.

Figure 3: Implementation

also possible that another transaction has signaled an abort (as we will discuss in Section 7.4) in which case the transaction aborts itself (Line 26). If the transaction breaks out of the inner loop, it will attempt to atomically update the lock's owner field to obtain ownership using a *compare-and-swap* operation (Line 29). If successful, the transaction must flush the list of inverses in the lock's inverse log. We discuss this process in Section 7.4.

After acquiring the lock using the TTAS lock described above, the hand off flag is raised using a compare-and-swap operation, shown in Figure 3(b). Again, in our implementation we simply set the least significant bit.

### 7.3 Commit

To ensure serializability, when a transaction tries to commit it must ensure that all other transactions from which it has been passed a lock have committed. In Figure 3(c) we give pseudo-code to show the commit procedures. First the transaction iterates over all of its dependencies (Line 5). If a dependency is active (Line 7) then the transaction must spin, waiting for it to commit. However, this is a potential source of deadlock if the dependency tries to acquire a lock held by the current transaction. Thus the transaction performs the standard Deadlocks

deadlock detection algorithm (Lines 8 – 10) checking to see if an abort is necessary. If the dependency has aborted (Line 12) then so must the current transaction. Finally, when all dependencies have been traversed, it is possible that another transaction has signaled an abort (as we will discuss in Section 7.4) in which case the transaction aborts itself (Line 17).

Aborting a transaction is as simple as marking `Self.CurrTxn` as aborted. This signals all other transactions which have been passed a lock to abort themselves. Moreover, inverses can be invoked as described in the next section.

## 7.4 Lazy Inverses

When a lock is acquired by a transaction, the transaction must flush the lock’s log of any inverses corresponding to transactions which previously held the lock. For brevity we omit pseudo-code, but the the algorithm proceeds as follows. First, the log of inverses is scanned to find the oldest entry corresponding to an aborted transaction. All newer entries correspond to transactions which are doomed, so each of them can be signaled to abort (they will receive the signal when spinning on other locks). Finally, the log is scanned backwards waiting for each entry’s transaction to either commit (in which case inverses are discarded) or abort (in which case inverses are invoked).

## 7.5 Implementation Notes

**Deadlock.** There are numerous opportunities for deadlock, so it is imperative that whenever a transaction is spinning it executes a deadlock detection algorithm (our implementation uses `DREADLOCKS` [7]). Deadlock is detected in the following places:

1. While acquiring the TTAS lock.
2. At commit time, while a transaction is waiting for others it depends on to commit.
3. When a transaction flushes a lock log and finds an active transaction.

**Digests.** We use digests per-thread to detect two pathologies. We use a 64-bit bit-vector representation of digests to detect deadlock between threads. We also use digests to detect cyclic dependencies between transactions. Since there are many transactions per thread, we need a digest which can represent a larger domain of unique identifiers. For this, we use a 64-bit Bloom filter. A full discussion of using digests can be found elsewhere [7].

## 8. Evaluation

We now turn to our evaluation. The experiments in this section were conducted on a machine running the GNU/Linux 2.6.26 kernel. The machine had 16 Intel Xeon 2.93GHz processors, 8 GB of RAM, and a 60 GB hard drive. The software stack consisted of TL2/x86 version 0.9.6 and STAMP version 0.9.10, compiled with `gcc` version 4.3.2. We extended our existing implementation of transactional boosting [6] to support abstract lock hand-off, lazy recovery, and cyclic dependency detection.

### 8.1 Synthetic Benchmarks

We first evaluated our technique with a simple shared array benchmark. Threads repeatedly execute transactions which access a shared array. In the case of traditional STM (RWSTM) synchronization and recovery is achieved in the usual manner, where as the boosted (XB) and dependent boosted (XB/D) cases synchronize via abstract locks and log inverses for recovery as discussed previously [6]. In each benchmark we used eight threads, each executing 100 transactions. The experiments reflect averages of 20 iterations. Note that we use `DREADLOCKS` in the case of XB to abort deadlocked transactions.

**Delays.** The benchmark in Figure 4a shows throughput as the length of transactions is increased with an increasingly computationally intensive workload (matrix multiplication) between each operation. The vertical axis is plotted in logarithmic scale. In the limit, performance is dominated by the matrix multiplication. Across the board, as previously shown [6], transactional boosting out-performs traditional STM. More to the point is

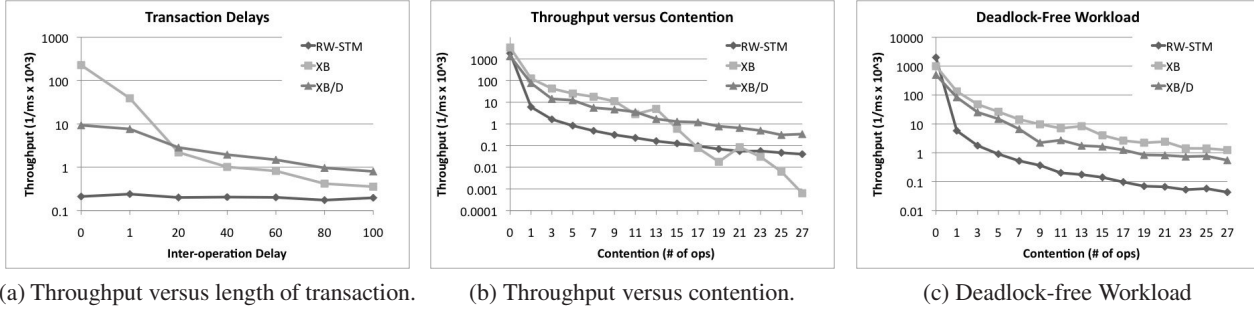


Figure 4: Results of experiments measuring the performance of dependent boosted transactions (XB/D), as compared with non-dependent boosted transactions (XB) and traditional read/write-based STM (RWSTM).

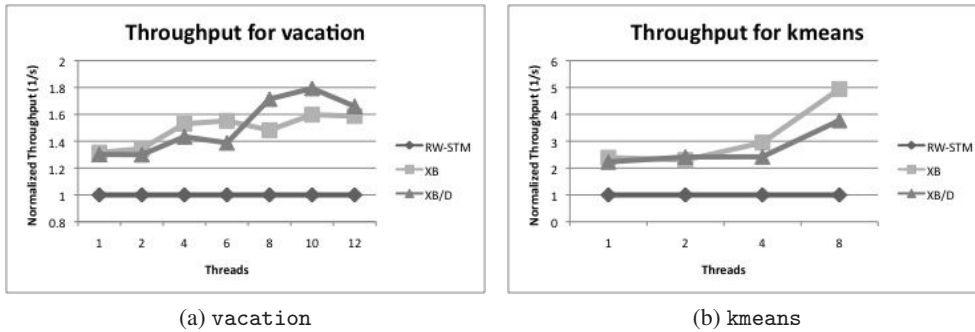


Figure 5: Throughput for dependent and normal transactional boosting for two of the STAMP benchmarks. Figures are normalized against the throughput of conventional TL2, which maintains shadow copies.

the relation between XB and XB/D. When transactions are short-lived, the overhead of tracking dependences is not offset by a performance gain and consequently XB performs better. However, as transactions are delayed, passing abstract locks enable transactions to “pipeline”: a transaction acquiring a held abstract lock can begin some computation rather than waiting idle.

**Contention.** We then compared how scenarios compared while varying the amount of contention. Figure 4b illustrates this relationship. When there is low contention, the overhead of passing abstract locks and tracking dependencies is not offset by a performance gain. But as contention is increased, the value of dependent transactions becomes apparent.

We found this benchmark to reveal a surprising result: handing off abstract locks affords better scalability. Under high contention normal boosting performance declines, as a result of deadlock-induced aborts. By contrast, when locks are passed, performance declines slower because more contention is tolerated.

**Absence of Deadlock.** Finally, we confirmed this revelation with the benchmark shown in Figure 4c. This benchmark illustrates performance when the workload contains no deadlock: each transaction acquires locks in a canonical order. In such workloads there is no advantage to passing abstract locks because there is no possibility of deadlock from which to escape.

## 8.2 STAMP Benchmarks

We now continue our evaluation by considering the STAMP benchmarks [4]. Most of these benchmarks involve short transactions so the overhead of handing off abstract locks is not offset by the afforded earlier progress. This is a general experimental limitation: we lack availability of transactional applications. We assume, nonetheless, that in many domains long-running transactions will be common. Note that all throughput is normalized against the throughput of RWSTM.

**Vacation.** The vacation benchmark is a simulation of a travel reservation system. A number of clients perform a series of random operations booking, canceling, and revising travel itineraries. The core data structure is a red/black tree, which we describe how to boost elsewhere [6]. Briefly, an abstract lock is associated with each key to achieve transaction-level synchronization, while coarse-grained locking is used to achieve thread-level synchronization. Dependent boosted transactions require no change to the boosted user code.

In Figure 5a we show the throughput for three scenarios. Both boosted implementations surpass the performance of RWSTM. When there is only one thread, the XB/D performance shows that the overhead of tracking dependencies is negligible. As the number of threads increases, the overhead of tracking dependencies becomes slightly noticeable until a critical point (around 8 threads) where dependencies can be leveraged to pipeline more transactions. Our experiment shows a degradation in performance around 12 threads, but we believe that to be an artifact of this particular benchmark.

**KMeans.** The kmeans benchmark is a clustering algorithm, in which objects are assigned to clusters on the basis of some similarity function. Again, no modifications were necessary to user code in our (passively) boosted version of kmeans. The workload in kmeans is quite different from vacation. Rather than performing operations on a shared red/black tree, the shared object is a multi-dimensional array.

Figure 5b shows the throughput of XB/D and XB normalized against RWSTM. As the number of threads increases, false conflicts drive the performance of RWSTM downward. We believe that this benchmark is an example of a workload in which there are many cyclic dependencies – such workloads may not be appropriate for dependent transactions.

## 9. Related Work

This work builds on *transactional boosting* [6]. We have presented a new form of boosting in which transactions which would normally block on an abstract lock held by another transaction, can instead have the abstract lock passed to them, allowing them to proceed in parallel. A dependency is then created to ensure serializability.

We also present a novel approach to inverse logging and a method for detecting cyclic dependencies, similar to DREADLOCKS [7]. Our methodology is able to efficiently detect cycles by using thread-local digests, which can either be implemented as bit-vectors or Bloom filters [2]. A thorough survey of related work on Bloom filters is also available [3].

The notion of dependent transactions has been well-studied in database literature. The canonical work by Badrinath and Ramamritham [1] introduced *recoverability*: the core property which dependent transactions are based on. A boosted transaction, by definition, consists of recoverable operations. In this paper we use the property of recoverability to apply operations, despite the fact that they depend on the success of another transaction. To ensure serializability we enforce a commit dependency, as per the authors' methodology [1]. The novelty of our work is applying recoverability to multi-processor programming, and combining it with lazy abstract locks and deadlock detection. The ARIES system [8] pioneered the idea of logging abstract operations and their inverses, as well as key-based locking.

Ramadan *et al.* recently explored how dependence-aware transactions can be implemented in hardware [9]. Unlike ours, their work applies to traditional Transactional Memory which is based on read/write sets. Our novel approaches to lazy recovery and efficient detection of cyclic dependencies are also distinctions of our work.

## 10. Conclusion

We have presented a new form of boosted transactions in which transactions hand off abstract locks, creating commit-time dependencies among transactions. We include a novel lazy approach to performing recovery and detecting cyclic dependencies. We have also shown that this technique improves performance for certain long-lived transactions with structured access to shared data objects, and achieves scalability under increased contention.

**Acknowledgments** We thank the anonymous reviewers for their valuable feedback. We would also like to thank Andrew Bartholomew for his assistance on this project.

## References

- [1] BADRINATH, B. R., AND RAMAMRITHAM, K. Semantics-based concurrency control: beyond commutativity. *ACM Transactions on Database Systems* 17, 1 (1992), 163–199.
- [2] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.
- [3] BRODER, A., AND MITZENMACHER, M. Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1, 4 (2004), 485–509.
- [4] CAO MINH, C., TRAUTMANN, M., CHUNG, J., McDONALD, A., BRONSON, N., CASPER, J., KOZYRAKIS, C., AND OLUKOTUN, K. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA 07)*. Jun 2007.
- [5] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC '06)* (September 2006).
- [6] HERLIHY, M., AND KOSKINEN, E. Transactional boosting: A methodology for highly concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '08)* (2008).
- [7] KOSKINEN, E., AND HERLIHY, M. DREADLOCKS: Efficient deadlock detection. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)* (June 2008).
- [8] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.* 17, 1 (1992).
- [9] RAMADAN, H., ROY, I., HERLIHY, M., AND WITCHEL, E. Committing conflicting transactions in an STM. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)* (Feb. 2009).