Checkpoints and Continuations Instead of Nested Transactions

Eric Koskinen
Department of Computer Science
Brown University
Providence, RI 02912
ejk@cs.brown.edu

Maurice Herlihy
Department of Computer Science
Brown University
Providence, RI 02912
mph@cs.brown.edu

ABSTRACT

We present a mechanism for partially aborting transactions through the use of data structure checkpoints and control-flow continuations. In particular, we show that boosted transactions [9] already have built-in restoration points and afford a simple, efficient implementation. Our mechanism is far simpler than previous work, which relied on complex nesting schemes to establish checkpoints. We demonstrate syntactic advantages and we quantify the overhead of checkpoints and explore several examples, illustrating the utility of partially aborting transactions.

We additionally present a novel queue-based spin lock which allows threads to timeout and differ in priority. Unlike the known lock due to Craig [5], our lock is more efficient for priority schemes of few levels.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features – Frameworks; Concurrent programming structures; E.1 [**Data Structures**]: Distributed data structures

General Terms

Algorithms, Languages, Theory

Keywords

Concurrency, parallel programming, transactional memory, boosting, checkpoints, continuations

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.

Being able to partially abort a transaction is useful for both performance and semantic reasons. Aborts are necessary to resolve conflicts, but often the conflict can be resolved by only rolling back some of a transaction's operations. Thus, partial aborts may yield higher performance since non-conflicting operations need not be reverted. Partial roll-back strategies can also support semantic constructs such as conditional and or-else synchronization, where a transaction decides to roll back part of a computation to pursue another path.

We require the notion of a *checkpoint*: a program location within a transaction to which control may jump during a partial abort. For example, after a transaction removes an element from list A, it may set a checkpoint before trying to insert the element into list B. Thus, if there is a conflict adding to B then the transaction can partially abort and try to insert the element again into list B or, alternatively, try to insert the element into list C. To accommodate partial aborts, each checkpoint must save and restore its *continuation*: the control state of the program location including the program counter, stack variables, and any allocated heap objects. One candidate checkpoint is the program location prior to each write operation. In practice this is too fine-grained: the cost of storing a continuation per write operation is high, and most saved continuations would never be used.

In the literature thus far, checkpoints have been emulated as nested transactions [12, 13, 7]. The beginning of a nested transaction is a user-defined checkpoint. Further, each checkpoint already has a continuation available: the continuation which is captured as part of the transactional infrastructure. Nested transactions additionally solve a second *modularity* issue: during a transaction, a subroutine may be invoked which itself initiates a transaction. Modularity is essential, but not tied to "Nested Transactions" (though the name may lead one to believe otherwise); modularity can be solved syntactically, as we will later discuss.

Existing nesting approaches are based on transactional memory systems which use read/write sets. They therefore involve infrastructure to maintain separate levels of read-/write sets so that recovery can occur at multiple nesting depths, a complexity which we argue is unnecessary.

In this paper, we show that transactions can be partially aborted without using nested transactions. We use the far

SPAA'08, June 14–16, 2008, Munich, Germany.

simpler approach of data structure *checkpoints* and control-flow *continuations* to be able to revert transactions to intermediate program locations. This approach has syntactic advantages over nested atomic blocks. For efficiency reasons checkpoints must be used sparingly and are best suited to operationally meaningful program locations rather than after each `write` operation; in much the same way, initiating a nested transaction after each `write` operation is not advisable.

In recent work, we have proposed Transactional Boosting [9] which solves synchronization and recovery through data structure semantics rather than read/write sets. In this paper, we additionally show that boosted transactions already establish operationally meaningful program locations that are well-suited checkpoints. Each logical operation can be an implicit checkpoint. For example, the logical operation of "adding a node to a list" may involve several write operations, but only the first operation is a natural restoration point. Boosting a transaction both removes the burden on the user to define explicit checkpoints, and also facilitates an efficient runtime storage of checkpoints.

Specifically, we make the following contributions:

- We show that nesting is unneeded to partially abort a transaction. This is discussed in Section 3. Instead, transactions can be reverted to previous control-flow locations by checkpointing data structures and capturing continuations. Checkpoints also have syntactic advantages over nested atomic blocks.
- In Section 3.1, we show that a boosted transaction has natural pre-defined checkpoints: each logical operation. This both eliminates the burden of choosing checkpoints and also leads to an efficient implementation, with a sparing use of checkpoints.
- We further show that Transactional Boosting allows us to record checkpoints *operationally* rather than treating data structure manipulations as flat memory access. Consequently, the cost of storing checkpoints is reduced.
- We explore the utility of checkpoints through a number of examples of boosted transactions in Section 4, and discuss our evaluation in Section 5.
- Finally, in Section 5.3 we present a novel priority-based queue spin lock with timeouts, and contrast it to the lock due to Craig [5]. Our lock has theoretically higher performance for binary priority schemes.

We emphasize that the utility of checkpoints and continuations is not limited to transactional boosting. For example, LogTM [12] could allow users to manually denote semantically meaningful control locations whose continuation could be captured during execution. Then, rather than using nested transactions, partial aborts can be realized by reverting `write` operations to restore the heap and then invoking stored continuations.

2. BACKGROUND

2.1 Transactional Boosting

Most transactional memory approaches rely on a log of memory access to determine synchronization and recovery. As a transaction executes, each `read` and `write` operation is stored in a log along with the corresponding memory location. Two transactions *conflict* when they concurrently access the same memory location and at least one operation is a `write`. In the event of a conflict, one transaction must be aborted: all `write` operations must be reverted. The log is again used to either restore old values or discard commit-time effects.

In recent work [9] we introduced Transactional Boosting, which yields significant performance gains by shedding memory access logs and relying on data structure semantics to determine synchronization and recovery. A more thorough discussion can be found in [9], but for completeness we summarize transactional boosting here.

Transactional Boosting is built upon linearizable base objects, which have an abstract state and a corresponding concrete implementation. The semantics of object methods are known, and often specified in terms of pre- and post-conditions describing the state of the object before and after the method invocation. Two method invocations are said to *commute* if applying them in either order causes the object to transition to the same state and respond with the same return values. Additionally, an *inverse* of a method is a second method which returns the object to the previous state.

Transactional Boosting relies on commutativity to define and detect conflicts. Users define *abstract locks* associated with invocations of methods on the boosted object. These locks conflict whenever a pair of methods do not commute. Before invoking a method, transactions must acquire all abstract locks associated with the invocation. In this manner, non-commutative operations never occur concurrently since one of the two operations will be delayed.

Inverses allow recovery to be performed at the granularity of method calls. While a transaction is executing, a log of operations is maintained. If the transaction aborts, this log is played in reverse order before abstract locks are released; a committed transaction can simply discard the log.

Here is an example. A Set may have methods $\mathsf{add}(x)$, $\mathsf{remove}(y)$, and $\mathsf{contains}(z)$ with the usual semantics. Pairs of method calls such as $\mathsf{add}(x)$ and $\mathsf{add}(y)$ commute for $x \neq y$. Indeed, any pair of methods with distinct arguments commute. Moreover, inverse options are well-defined: the inverse of $\mathsf{add}(x)$ is simply $\mathsf{remove}(x)$ and vice-versa.

2.2 Nested Transactions

Several nesting approaches have been proposed [12, 13, 7] for software transactional memory. The purpose of nested transactions[1] is twofold:

Checkpoints. A nested transaction establishes a new program location where control may be returned in the event of an abort. Specifically, the beginning of each nested transaction is a checkpoint. Consider the following example:

```
global int list [10];
atomic {
  int x = list [0];
  atomic {
      list [2] = x + 1;
  }
}
```

If access to list [2] causes a conflict, the inner transaction may be aborted rather than the entire outer transaction. Thus, each nested **atomic** statement demarcates a checkpoint.

[1]By "nested transactions" we mean closed nested transactions that only commit if the top-level parent commits.

Modularity. Often code that transactionally mutates data structures may be encapsulated in libraries. Nested transactions enable the sound interaction between transactional user code and invocations of transactional library methods. For example, before a transaction removes an element from a queue, it may use a library method to check that the queue is nonempty:

```
global Queue sharedQueue;

bool EmptyQueue(Queue q) {
  atomic { ... }
}

atomic {
  if (EmptyQueue(sharedQueue))
    throw EmptyQueueException;
  return Dequeue(sharedQueue);
}
```

In the next section we describe how partial aborts can be realized through first-class checkpoints and continuations. We will then revisit nested transactions, comparing the two approaches, and showing that both of the above concerns can be addressed with checkpoints and continuations.

3. CHECKPOINTS AND CONTINUATIONS

The key idea of this paper is that partial aborts can be realized without using nested transactions. In this section, we define partial aborts, then in Section 4 we explore some applications of partial aborts, and in Section 4.1 we quantify the efficacy of partial aborts for one particularly useful application.

As a running example, consider a pool of threads transactionally accessing shared hash tables. One thread's access pattern might be as follows:

```
global HashTbl ht1, ht2;

atomic {
  Object oldVal = HashTbl_Remove(ht1,7);
  Object newVal = calculate(oldVal);
  HashTbl_Insert(ht2,7,newVal);
}
```

In this example, the transaction removes an element from the first hash table, calculates a new element, and inserts the new element into a second hash table.

Now, consider what would happen if contention at the second hash table triggered an abort of this transaction. In the simplest case, transactional implementations would undo all of the transaction's effects, returning the element to the first hash table.

Yet there may not be a conflict at ht1, so it is unfortunate that the transaction must be aborted entirely. A more satisfying solution would be to *partially* abort the transaction. If control is returned to the point before the insertion, then more progress is preserved. This is of course a trivial example with a low penalty for aborting entirely, but it is easy to imagine long-running transactions where the penalty is more severe.

The core challenge involved in partially aborting a transaction is to be able to soundly return to an earlier program location. In programming language theory, this facility is known as a *continuation* [15, 1]. There are two correctness criteria involved in implementing a continuation:

1. **Stack Restoration**: All stack variables in scope must be restored to their values at the time when control first passed through the program location. Formally, the continuation *captures the environment* of the program location.

2. **Heap Restoration**: All heap data structures in scope must be restored to their state at the time when control first passed through the program location. Formally, the continuation *captures the store* of the program location.

Purely functional languages are heap-less and so implementations must only be concerned with restoring the stack. Imperative languages add the complication of restoring the heap. Finally, concurrent shared-memory imperative programs must be able to restore the heap in such a way that consistency is maintained with respect to other concurrently executing threads.

In the remainder of this paper, we focus on checkpoints and continuations as they apply to boosted transactions. We reiterate that the principles we discuss are not limited to the realm of transactional boosting. Nonetheless, a boosted transaction has convenient properties (discussed below) which we exploit to gain an efficient implementation.

3.1 Boosted Checkpoints

Transactions manipulate a boosted object by executing a series of operations, each causing the object to transition from one abstract state to another. Continuing with the hash table example, one such operation is "inserting a key/value pair." Although the single invocation may result in many `read` and `write` operations, it implements a well-defined abstract state transition.

We define a *boosted checkpoint* at the program location that precedes each operation on a boosted object. In the hash table example, checkpoints are defined before the removal and insertion and implicitly at the beginning of the transaction. Checkpoints can be automatically inserted before each boosted method call, but for clarity they are manually defined in the following example:

```
atomic {                                  // CP0
  Object x, y;
  checkpoint;                             // CP1
  x = HashTbl_Remove(ht1,7);
  y = foo(x);
  checkpoint;                             // CP2
  HashTbl_Insert(ht2,7,y));
}
```

The first thing to note is that checkpoints are syntactically simpler then nested transactions. Unlike nested transactions, checkpoints allow a programmer to insert partial abort locations in an atomic block without rewriting the block. By contrast, the nesting equivalent of the above example would have added layers of depth.

Boosted checkpoints are not snapshots of the object, but rather semantically rich locations in the program's control-flow graph. From one checkpoint to the next, there is a well-defined transition which is given by the semantics of the interleaved operation. For example, the transition between the

checkpoints labeled CP1 and CP2 is HashTbl_Remove(ht1,7), in other words: the removal of the value at key 7 from hash table ht1. The transition from checkpoint CP2 and the end of the transaction is similarly defined.

Since boosted object state transitions are well-defined between checkpoints, heap recovery can be realized at any checkpoint without maintaining shadow copies. If, for example, this transaction is aborted after completing the insertion into ht2, it can be partially aborted to CP2 rather than to the beginning of the transaction, by simply executing the inverse operation of HashTbl_Insert(ht2,7,y). As discussed in Section 2.1, all boosted objects have known inverse operations.

Boosted checkpoints handle recovery in a way that is fundamentally different from nested transactions. Nested implementations maintain extra copies of objects. In this sense, they are agnostic of data structure semantics and treat memory as a flat array of locations and values. However, as we demonstrated in [9], this is often expensive.

Checkpoints can be used anywhere within an atomic block. For example, programmers can specify runtime decisions as to whether a checkpoint is taken. For example, a programmer may write the following:

```
atomic {
  ...
  if ( decision ())
    checkpoint;
  DataStructureOperation();
  ...
}
```

The equivalent example using nested transactions is inconvenient. Runtime decisions cannot be made as to whether or not to invoke a nested transaction without duplicating the entire body of the nested transaction.

3.2 Boosted Continuations

We now consider the issue of returning the program control to an earlier checkpoint. In the above hash table transaction, we may for example want to return control to the CP2 checkpoint if an abort occurs before the end of the transaction. Such jumps in control are a special case of the general concept of non-local control flow, which is accomplished by *continuations* from programming language theory. Informally, continuations represent the current values of stack variables in scope (the *environment*), dynamically allocated heap data (the *store*) and the program location (a point on the control-flow graph). Our non-local control flow is accomplished with two steps:

1. *Continuation storage.* Each time control flow passes a checkpoint, the current continuation is captured. Indeed, the checkpoint itself is augmented with the storage of the corresponding captured continuation.

2. *Continuation invocation.* When a transaction aborts, inverses are played backwards until the desired checkpoint is reached, discarding saved continuations along the way. Upon reaching the desired checkpoint, the corresponding continuation is invoked.

Nested Transactions rely on *implicit* continuations to return to the beginning of an inner transaction. Transactions

```
global Object data[1000];

atomic {
   key_list [] = decide();
  for(int key :: key_list []) {
    checkpoint;
    try { QLock(key); }
    catch(QLockTimeout) { throw AbortedException; }
    Modify(data,key);
    LogInverse(ModifyInv,data,key);
  }
}
```

Figure 1: A transaction accessing a boosted shared array.

by definition already have a built-in notion of returning control, so the nesting approach introduces new transactions at each program location. We suggest that conventional closed nested transaction schemes and data structures can be discarded in favor of this simple continuation scheme.

3.3 Implementation Notes

We discuss full implementation details in Section 5.1 but continuations deserve special treatment here. Unfortunately, availability of continuations differs from one language to the next. High-level languages such as Scheme and Standard ML of New Jersey support first class continuations, while most imperative languages offer dynamic variants of **goto** such as setjmp/longjmp.

Our prototype implementation is built on Transactional Locking 2 (TL2) [6] which is written in C. As such, the checkpoint continuations were captured with getcontext and invoked with setcontext. These standard C library functions allow us to capture the continuation of multiple checkpoints, whereas setjmp/longjmp capture only a single point in the control flow.

Unfortunately, these library functions do not capture the entire program environment. To compensate, our prototype allocates additional storage for stack variable values alongside the captured context. When control passes through a checkpoint, the macro SAVE_ENV(var) adds the current value of var to a data structure, which is later restored via RESTORE_ENV(var) when control returns to the context after a partial abort. Ideally, of course, this work could be done by a compiler.

4. APPLICATIONS AND EXAMPLES

In this section we explore some examples that illustrate the utility of partial aborts in Transactional Boosting.

4.1 Priority

In a priority scheme, it is desirable for some distinguished threads to have greater throughput than others. In the context of transactional memory, high priority transactions can force low priority transactions to abort. Unfortunately, aborting low priority transactions may reduce low priority throughput. Checkpoints and continuations allow us to partially abort low priority transactions, rolling them back just far enough to allow high priority transactions to commit. Low priority transactions can subsequently resume.

Consider threads accessing a boosted shared array, as is shown in pseudo-code in Figure 1. For the moment, assume QLock is a table from keys to simple spin locks. Let us also assume that Modify(data,key) is some method which modifies the index key and that ModifyInv(data,key) is the method which performs the corresponding *inverse* operation (one might increment and the other decrement, for example) which is appended to the transaction log by LogInverse().

A correct semantics for this array states that two invocations *commute* if they access distinct indices. If transactions access multiple indices, then they will acquire an abstract lock for each before committing. For example, the key_list for two transactions might include the following array indices:

$$\text{Transaction } A\text{: } 28, 56, 41, 10$$
$$\text{Transaction } B\text{: } 21, 19, 41, 67, 95$$

If transaction A acquires the abstract lock for 41 first, then transaction B will be delayed until A commits.

Now let's add priority to the scenario: perhaps B has higher priority than A. We would therefore like to allow B to proceed and commit at the cost of A's throughput. Indeed this is possible without checkpoints: we can simply *abort* A. However, we then lose the progress that A has already made. There is no need for A to undo the work which required the acquisition of locks 28 and 56 since B will never try to acquire either lock.

The main idea of this paper addresses this issue. If we capture the continuation at each checkpoint, then we can semantically undo the changes to the data structure made by A to the point before 41 was acquired by executing inverse operations.

4.1.1 CLH Queue Lock with Priority

Realizing partial aborts in the context of priority requires that our spin lock support both priority and timeouts. The queue-based spin lock due to Craig, Landin and Hagersten [4, 11] (CLH Lock) already supports timeouts, so we build on the CLH Lock and present a novel spin lock which accommodates thread priority in a distinctly different way than the priority queue lock due to Craig [5]. We qualitatively describe our lock here and a full design discussion is given in Section 5.3.

Threads trying to acquire a queue-based spin lock enqueue themselves into a list of acquirers. The front-most thread has acquired the lock and executes its critical section. When the lock is released, the front-most thread passes the lock to the next thread in line, and removes itself from the queue. In this way, cache coherency issues are minimized because threads spin on distinct memory locations.

We augment the spin lock by defining two queues: a queue of low priority threads and another of high priority threads. When a low priority thread releases the lock it passes availability either to the high priority queue (if any high priority threads exist) or to the next low priority thread. Analogously, a high priority thread either releases it to the next high priority thread or turns the lock over to the low priority queue.

4.1.2 Aborting Low Priority Threads

When a high priority transaction needs to acquire a lock which is currently held by a low priority transaction, it is desirable to abort the low priority transaction. Our transactional model has no notion of preemption, but we can signal low priority transactions to abort themselves. When a high priority transaction tries to acquire a lock held by a low priority transaction, it can send a brief message (by writing to thread-local storage) to the low priority transaction indicating which key it wishes to acquire. Low priority transactions listen for messages while they are spinning on other locks.

When a low priority transaction receives a message from a high priority transaction, it initiates its own partial abort. The message contains the key desired by the high priority transaction. The low priority transaction can therefore partially abort itself to the checkpoint which immediately precedes the contested lock. When the lock is released the high priority transaction, having earlier enqueued itself, immediately obtains it and continues.

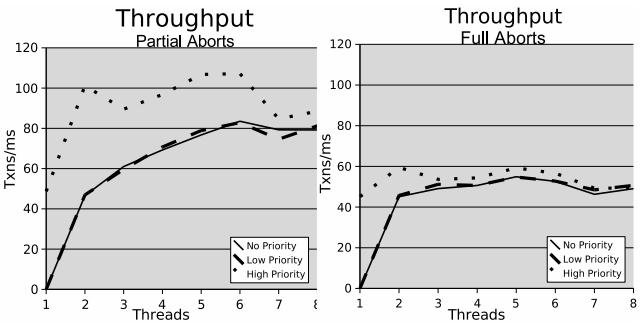

Figure 2: Throughput of high priority, low priority, and priority-free threads when they are partially aborted (to the left) or completely aborted (to the right).

4.1.3 Performance

We evaluated our priority example by confirming that high priority threads have higher throughput than (a) lower-priority threads and (b) the throughput they would have had in the absence of priority schemes. Figure 2 shows the throughput of high and low priority threads, compared with threads in a priority-free scheme. To the left, low priority transactions are partially aborted, whereas they are completely aborted on the right. The two graphs illustrate that in all cases (high priority, low priority, and priority-free) threads have greater throughput when their transactions are partially aborted rather than completely aborted.

4.2 Conditional Synchronization

Checkpoints allow us to implement two forms of transactional conditional synchronization. STMHaskell introduced two such constructs called **retry** and **orElse**. Both provide general mechanisms for incorporating conditional synchronization into transactional computation. Supporting them has a pervasive effect on any STM implementation, while a boosted implementation of transactional objects does not. In this subsection, we use the notation:

result = **with**(absLock.lock(k)) { DataStructureMethod() }

to represent the combined task of boosted access to an object: (a) acquire abstract lock absLock for key k (b) execute DataStructureMethod and if successful (c) implicitly log the corresponding inverse.

```
1  List inbound, outbound;
2  AbstractLock inL, outL;
3
4  atomic {
5    with(inL.lock(0)) { Prepare(inbound) }
6
7    checkpoint;
8    Object in = with(inL.lock(0)) { Dequeue(inbound) }
9    if (!in.isSpecial)
10     retry;
11
12   with(outL.lock(0)) { Enqueue(outbound,in) }
13 }
```

Figure 3: Example using retry in a boosted transaction.

```
1  HashTbl htA, htB, htC;
2  AbstractLock alA, alB, alC;
3
4  atomic {
5    Object r = with(alA.lock(key)) { Remove(htA,key) }
6
7    checkpoint;
8    {
9      with(alB.lock(key)) { Remove(htB, key); }
10     with(alB.lock(key)) { Add(htB, key, r); }
11   }
12   orElse
13   {
14     with(alC.lock(key)) { Remove(htC, key); }
15     with(alC.lock(key)) { Add(htC, key, r); }
16   }
17 }
```

Figure 4: Example using orElse in a boosted transaction.

Figure 3 depicts an example of the **retry** construct. First the transaction prepares the inbound queue, and then begins dequeuing elements. Each dequeued element is examined, and if the thread has not found the desired element, it uses the **retry** construct to partially abort the transaction to the previous checkpoint defined on Line 7. The element is returned to the queue, and the transaction tries to dequeue again, hoping that some other thread has enqueued an element of interest.

The example in Figure 4 illustrates the utility of the **orElse** construct. In this example, the transaction removes an element from htA and then tries to add it to another hash table. While adding it to htB, it may experience a conflict with another transaction concurrently accessing htB. In this case, we can partially abort the transaction and, following the directions of the **orElse** statement on Line12, the transaction can attempt to add the element to htC instead. This example has applications towards various forms of the work queue paradigm.

4.3 Contention Management

Contention Managers are used in transactional memory to resolve conflicts with runtime heuristics to improve throughput. Boosted transactions empower contention managers with a more fine-grained abort mechanism. With the mechanism presented in this paper, contention managers are additionally empowered with the ability to *partially* abort transactions. Moreover, by examining the operation log, contention managers can revert transactions to just before the (first) conflicting operation. See [14] for a detailed discussion of contention management.

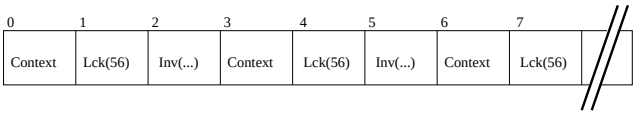

Figure 5: Diagram of the runtime *computation log*, which stores captured continuations ("Context"), abstract locks ("Lck"), and inverse methods ("Inv").

4.4 Modularity

Modularity is a desirable attribute of transactional programs. For example, during a transaction a user may want to invoke library methods which themselves initiate transactions. Consider the following code:

```
Object swap(HashTbl ht, int key, Object newVal) {
  atomic {
    Object r = HashTbl_Remove(ht,key);
    HashTbl_Insert(ht,key,newVal);
    return r;
  }
}

atomic {
  Object x = swap(htA, key, y);
  HashTbl_Insert(htB, key, x);
}
```

Here the transaction within swap() is nested (syntactically) within the outer transaction by the program's control flow.

Rather than initiating an entirely new inner transaction, a boosted implementation with checkpoints can instead simply establish a new checkpoint at the beginning of each nested transaction. Finding nested transactions is a simple runtime task, keeping a flag per thread which indicates whether a transaction has been initiated. Moreover, this task can often be further simplified by statically detecting syntactically nested transactions and replacing them with checkpoints.

5. EVALUATION

5.1 Checkpoints and Continuations

We implemented Checkpoints and Continuations on top of our Transactional Boosting implementation in TL2 [9]. Much like abstract locks and inverses, checkpoints are stored in a runtime *computation log*. A diagram of the computation log is given in Figure 5. At each checkpoint, the continuation is captured and appended to the log. Checkpoints ("Context") precede operations on shared data structures, which involves acquiring an abstract lock ("Lck"), performing the operation, and then recording the inverse operation ("Inv") in the log.

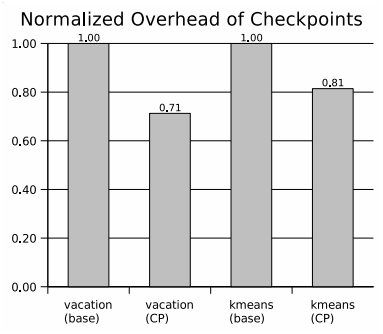

Figure 6: Normalized throughput of boosting with and without checkpoints for two *STAMP* benchmarks.

When a transaction is aborted, as long as the log is traversed in the reverse direction, invoking any context produces correct behavior. As a context is passed it is deallocated, inverses are invoked to revert data structure operations, and abstract locks are released allowing other threads to acquire them. In Section 4 we discussed application-specific strategies for deciding how far to traverse.

5.2 Overhead

There is an overhead associated with capturing contexts and storing them in the computation log. The amount of overhead depends on the workload. When a transaction performs simple operations such as incrementing a counter, then the overhead is significant. More realistic workloads (indeed the type of workloads for which it is desirable to partially abort transactions in the first place) have a more manageable overhead.

We modified two of the Stanford *STAMP* benchmarks [2] (written in C) to use boosting with and without checkpointing. The benchmarks were run on a multiprocessor with four 2.0 GHz Xeon processors, each one two-way hyper-threaded for a total of eight threads. Figure 6 we compare the throughput of transactions with and without the capture of continuations for the vacation and kmeans benchmarks[2]. These experiments show that the added expressive power of partial aborts is not without cost. In both cases, the overhead of capturing and maintaining checkpoints degraded performance.

5.3 Spin Lock Design and Implementation

The priority example in Section 4.1 requires a spin lock in the queue form which is due to Craig, Landin and Hagersten [4, 11], with the added need for timeouts and priority schemes.

As in [4, 11], threads attempting to acquire a lock append new "lock nodes" to the tail of a queue, and examine the previous tail element. The threads append themselves by swinging the tail pointer with a compare-and-swap (CAS) operation to ensure a serial ordering. Threads then spin, waiting until their predecessor lock node is marked "unlocked," at which point the thread has logically acquired the lock. After completing its critical section, the thread marks its own node as "unlocked," allowing the next thread to acquire the lock. Threads may wish to timeout, in which case they can mark a node as "abandoned." The predecessor thread spinning on the newly abandoned node deallocates the node and follows a pointer to spin on the subsequent node in the queue.

[2]We used the following recommended flags:
vacation −n4 −q90 −u80 −r65536 −t409
kmeans −m40 −n40 −t0.05 −i inputs/random1000_12

```
1  void QLockKey_Lock(QLockKey_t* ql) {
2    // Enqueue
3    if (! HighPriority || QLockKey_Empty(ql->low_queue))
4      QLockKey_Enqueue(ql->low_queue);
5    else {
6      QLockKey_Enqueue(ql->high_queue);
7      QLockKey_Enqueue(ql->low_queue);
8      if (QLockKey_Acquired(ql->low_queue))
9        QLockKey_Dequeue(ql->low_queue);
10     else
11       QLockKey_Abandon(ql->low_queue);
12   }
13   // Spin
14   while(high || pattience--) {
15     if (SpinNode->abandoned)
16       QLockKey_Advance(SpinNode);
17     else if (! SpinNode->locked)
18       // acquired!
19     else if (ReceivedMessage(OtherLock))
20       PartiallyAbortTo (OtherLock);
21     else if (NeedToPreempt())
22       SendMessage(SpinNode->owner());
23     else
24       // keep spinning!
25   }
26   // Timeout
27   QLockKey_Abandon(ql->low_queue);
28 }
```

Figure 7: Pseudo-code for QLock_Lock().

We augment the queue lock for priority by defining one queue per priority level. In the example of Section 4.1 only two queues are necessary: a high priority queue and a low priority queue. Threads trying to acquire the lock, add nodes to the queue corresponding to their priority level. When a low priority thread holding the lock completes its critical section, it passes the lock to the high priority queue if it is non-empty, or else passes it to the next node of the low priority queue. Similarly, when a high priority thread completes its critical section, it either passes the lock to the next high priority thread or else passes it back to the low priority queue. This generalizes to any number of priority levels.

We also introduce preemption to the queue lock. When a high priority thread tries to acquire a lock currently held by a low priority queue, the high priority thread can request that the low priority thread release the lock. The high priority thread writes the name of the lock into the holder's thread-local field. The holder checks the field whenever it is spinning on other locks, and if it notices that it is non-null, then the low priority thread partially aborts itself to the checkpoint which precedes the contested lock. To avoid deadlock, we also allow low priority threads to abort each other, using the transaction start time as a tie-breaker.

Pseudo-code for our implementation of QLock_Lock() is

given in Figure 7. The first enqueue case (Line 3) is an important fast path. When it is empty, high priority threads can simply add themselves to the low priority queue. The second enqueue case (Line 5) occurs when the fast path does not apply and the thread has high priority. In addition to adding itself to the high priority queue (Line 6), the thread must also enqueue in the low priority queue (Line 7) so that the lock can be passed from the low queue to the high queue. If the thread acquired the low priority lock, then by dequeuing from the low priority queue, the lock will be passed to the non-empty high priority queue. Otherwise if by enqueuing, the low priority lock was not acquired, then the thread can abandon the low priorty queue since some other thread will pass the lock to the high priority queue.

In the remainder of QLock_Lock() (Lines 14 - 27) the thread spins over a case analysis, attempting to acquire the lock. The thread may timeout (Line 14 and Line 27), it may find the next node in the queue to be abandoned (Line 15), it may acquire the lock (Line 17), it may receive a message from another transaction requesting an abort (Line 19), or it may decide to abort another transaction (Line 21). Unlocking is simply accomplished with QLockKey_Dequeue(ql).

Not shown in Figure 7 are two other implementation notes. First, in the priority example, threads may try to acquire the same lock repeatedly during a single transaction. We added a counter to track how many times the lock had already been acquired, performing the actual QLockKey_Lock(). Second, in order for high priority threads to find the low priority owner of a contested lock, threads "announce" themselves sometime after they acquire the lock, and "unannounce" before releasing the lock.

The above implementation is different from the priority queue lock due to Craig [5]. The lock of Craig stores priority as a field within the queue. This has the advantage of compactness, at the cost of a queue traversal before each acquisition. By contrast, our implementation maintains separate queues for each priority level. Thus, acquisition is done in constant time.

6. FORMAL MODEL

THEOREM 6.1. *A transactional boosting implementation that obeys the correctness rules specified in [8] and permits partial aborts as described informally above yields histories whose committed transactions are strictly serializable.*

PROOF. *Trivial. Follows from the* Main Theorem *of [8], with an added "Partial Abort" event.* □

Informally, the system is correct because partial aborts "annihilate" a subsequence of a thread's history, which is later regenerated. When the thread commits (if it does commit), commutativity ensures that the history defines the same state as if the thread's operations occurred instantaneously.

7. RELATED WORK

Our work stands in contrast to the related work on nesting approaches for transactional memory. We have shown that with a partial abort mechanism such as the one presented in this paper, nesting is unneeded.

As discussed in Section 3.1, there are several advantages to using checkpoints instead of nested transactions. First, data checkpoints and control-flow continuations alleviate the need for expensive and complex levels of activation records in a nesting implementation. Second, our approach offers syntactic advantages: checkpoints can be added to an existing atomic block, as opposed to rewriting the block in nested form. Moreover, checkpoints are syntactically convenient for making runtime decisions about whether to establish a partial abort at a particular location; the equivalent nested approach requires significant code duplication. We now summarize the recent work on nested transactions.

Closed nested transactions have been proposed by Moravan *et al.* [12] for the log-based transactional memory implementation LogTM. LogTM implements STM by tracking read/write sets in a runtime log. Nested transactions are implemented by maintaining layers of read/write, analogous to activation records. Abstract nested transactions [7] are closed nested transactions with an added optimization. In this paper we argue that the need for nested transactions is artificial and suggest that such complex nesting strategies could be discarded in favor of simple data structure checkpoints and continuations.

More recently, *open* nested transactions were proposed by Ni *et al.* [13] and expose the layers of memory access to developers through a simple API. The goal is to allow programmers to manually avoid aborts which arise from false conflicts. Additionally, examples of how to implement collection classes with open nested transactions was given by Carlstrom *et al.* [3]. We again argue that nesting is unneeded. Moreover, false conflicts can be eliminated with the transactional boosting methodology [9] rather than exposing an API that is prone to developer mistakes.

Hardware support for nested transactions has also been explored by Lev and Maessen [10], who show how nesting can be implemented in hardware by subdividing a (software) nested transaction into multiple hardware transactions.

8. CONCLUSION

We have shown that transactions on boosted objects can be partially aborted without the need for nesting. We store continuations which capture the context before each method invocation, and roll back transactions by invoking inverse object operations. We explored the utility of partial aborts through several examples.

The added expressiveness of partial aborts comes with the cost of storing continuations with each checkpoint. However, since continuations are stored sparingly (per logical operation rather than per write operation) boosted checkpoints are more feasible than conventional read/write checkpoints. Nonetheless, it may be worth exploring applying the techniques presented here toward conventional transactional memory implementations such as LogTM [12].

Acknowledgments

This work was funded by NSF grant 0410042, and grants from Sun Microsystems, Microsoft, and Intel. We would like to thank Yossi Lev and the anonymous reviewers for their feedback.

9. REFERENCES

[1] APPEL, A. W. *Modern Compiler Implementation in ML.* Cambridge University Press, New York, NY, USA, 2004.

[2] 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.

[3] Carlstrom, B. D., McDonald, A., Carbin, M., Kozyrakis, C., and Olukotun, K. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'07)* (New York, NY, USA, 2007), ACM, pp. 56–67.

[4] Craig, T. Building fifo and priority-queueing spin locks from atomic swap. Technical Report 93-02-02, Department of Computer Science & Engineering, University of Washington, 1993.

[5] Craig, T. Queuing spin lock algorithms to support timing predictability. *Proceedings of the Real-Time Systems Symposium* (1993), 148–157.

[6] Dice, D., Shalev, O., and Shavit, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC '06)* (September 2006).

[7] Harris, T., and Stipić, S. Abstract nested transactions. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Languages, compilers, and hardware support for transactional computing (TRANSACT '07)* (2007).

[8] Herlihy, M., and Koskinen, E. Transactional boosting: A methodology for highly concurrent transactional objects. Technical Report CS-07-08, Department of Computer Science, Brown University, 2007.

[9] 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).

[10] Lev, Y., and Maessen, J.-W. Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '08)* (2008).

[11] Magnusson, P. S., Landin, A., and Hagersten, E. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing* (Washington, DC, USA, 1994), IEEE Computer Society, pp. 165–171.

[12] Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)* (New York, NY, USA, 2006), ACM Press, pp. 359–370.

[13] Ni, Y., Menon, V. S., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07)* (New York, NY, USA, 2007), ACM Press, pp. 68–78.

[14] Scherer, III, W. N., and Scott, M. L. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2005), ACM, pp. 240–248.

[15] Strachey, C., and Wadsworth, C. P. Continuations: A mathematical semantics for handling fulljumps. *Higher Order Symbol. Comput. 13*, 1-2 (2000), 135–152.