Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory

Woongki Baek, Nathan Bronson, Christos Kozyrakis, Kunle Olukotun
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{wkbaek,nbronson,kozyraki,kunle}@stanford.edu

ABSTRACT

Transactional Memory (TM) is a promising technique that simplifies parallel programming for shared-memory applications. To date, most TM systems have been designed to efficiently support single-level parallelism. To achieve widespread use and maximize performance gains, TM must support nested parallelism available in many applications and supported by several programming models.

We present NesTM, a software TM (STM) system that supports closed-nested parallel transactions. NesTM is based on a high-performance, blocking STM that uses eager version management and word-granularity conflict detection. Its algorithm targets the state and runtime overheads of nested parallel transactions. We also describe several subtle correctness issues in supporting nested parallel transactions in NesTM and discuss their performance impact.

Through our evaluation, we quantitatively analyze the performance of NesTM using STAMP applications and microbenchmarks based on concurrent data structures. First, we show that the performance overhead of NesTM is reasonable when single-level parallelism is used. Second, we quantify the incremental overhead of NesTM when the parallelism is exploited in deeper nesting levels and draw conclusions that can be useful in designing a nesting-aware TM runtime environment. Finally, we demonstrate a use-case where nested parallelism improves the performance of a transactional microbenchmark.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – parallel programming

General Terms

Algorithms, Design, Performance

Keywords

Transactional Memory, Nested Parallelism, Parallel Programming

SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.

1. INTRODUCTION

Transactional Memory (TM) [13] has surfaced as a promising technique to simplify parallel programming. TM addresses the difficulty of lock-based synchronization by allowing programmers to simply declare certain code segments as *transactions* that execute in an *atomic* and *isolated* way with respect to other code. TM takes responsibility for all concurrency control. The potential of TM has motivated extensive research on hardware, software, and hybrid implementations. We focus on software TM (STM) [8, 11, 19], because it is the only approach compatible with existing and upcoming multicore chips.

Most TM systems, thus far, have assumed that the code within a transaction executes sequentially. However, real world applications often include the potential for *nested parallelism* in various forms such as nested parallel loops, recursive function calls, and calls to parallel libraries [21]. As the number of cores scales, it is important to fully exploit the parallelism available at all levels to achieve the best possible performance. In this spirit, several parallel programming models that support nested parallelism have been proposed [1, 20]. Hence, to maximize performance gain and integrate well with popular programming models, TM must support nested parallelism.

However, efficiently exploiting nested parallelism in TM is not trivial. The general challenge of nested parallelism is amortizing the overhead for initiating, synchronizing, and balancing inner-level, fine-grained parallelism [5]. Nested parallelism within transactions exacerbates this challenge due to the extra overheads for initiating, versioning, and committing nested transactions. The design of a TM system that supports nested parallel transactions is also challenging. First, the conflict detection scheme must be able to correctly track dependencies in a hierarchical manner instead of a flat way. Nested parallel transactions may conflict and restart without necessarily aborting their parent transaction. Second, apart from the runtime overhead, we must ensure that the memory overhead necessary for tracking the state of nested transactions is small. Third, since some applications may not use nested parallelism, we must ensure that its overhead is reasonable when only a single level of parallelism is used.

A few recent works on nested parallelism in STM have discussed the semantics of nested parallel transactions and provided prototype implementations [2, 4, 18, 22]. However, the following questions still require further investigations. First, what is a cost-effective algorithm for nested parallelism in high-performance STMs? Second, using a detailed performance analysis, what are the practical tradeoffs and issues when using nested parallelism in STM? Answering these questions is also important to guide future work on nesting-aware TM runtime environments.

This paper presents *NesTM*, an STM that supports closed-nested parallel transactions. NesTM is based on a high-performance, blocking STM that uses eager versioning and word-granularity conflict detection. NesTM extends the baseline STM to support nested parallel transactions in a manner that keeps state and runtime overheads small.

The specific contributions of this work are:

- We propose an STM system that supports nested parallelism with transactions and parallel regions nested in arbitrary manners.
- We present several complications of concurrent nesting, describe solutions for correct execution, and discuss their impact on performance.
- We provide a quantitative performance analysis of NesTM across multiple use scenarios. First, we show that the performance overhead of NesTM is reasonable when using only a single level of parallelism. Second, we quantify the overhead of NesTM when we exploit the parallelism in deeper nesting levels. Finally, we demonstrate that NesTM improves the performance of a transactional microbenchmark that uses nested parallelism.

The rest of the paper is organized as follows. Section 2 reviews the baseline STM and the semantics of nested parallel transactions. Section 3 describes NesTM and Section 4 discusses subtle correctness issues. Section 5 presents the quantitative evaluation. Section 6 reviews related work. Finally, Section 7 concludes the paper.

2. BACKGROUND

2.1 Baseline STM

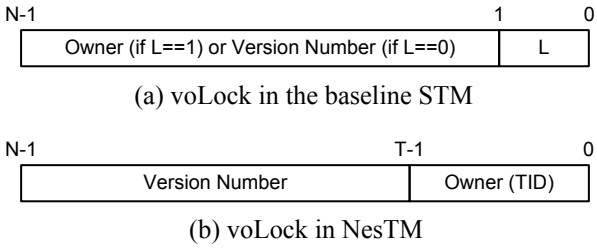
Our starting point is a blocking STM algorithm that uses eager versioning [11, 19]. This approach has been shown to have performance advantages over non-blocking or lazy versioning STMs and is used by the Intel STM compiler [19] and the Microsoft Bartok environment [11]. While we focus on an STM with word-granularity conflict detection, our findings can apply to STMs that perform object-granularity conflict detection.

The exact code we start with is an eager variant of TL2 STM [6, 9]. It maintains an undo log for data written within a transaction. The STM uses a global version clock to establish serializability. Using a hashing function, each memory word is associated with a variable (voLock) that either acts as a lock or stores a version number (i.e., the clock value when the word was written by a committing transaction). When a transaction reads data, it inserts them in its read-set. When a transaction writes data, it acquires the associated locks. The code for the read and write barriers is carefully optimized to keep the overhead per call (some parts are in assembly) small. Conflicts are detected by checking the associated voLocks when read, write, and commit barriers are executed. A randomized exponential backoff scheme is used for contention management.

2.2 Semantics of Concurrent Nesting

We describe a few concepts for nested parallel transactions. Additional discussion is available in [2, 16].

Definitions and concepts: At runtime, each transaction is assigned with a *transaction ID (TID)*, a unique positive integer. *Root* transaction (TID 0) is reserved to represent the globally committed state of the system. Every non-root transaction has a unique *parent* transaction. *Top-level* transactions are the ones whose parent is the root transaction. Following the assumption in [16], a transaction is only allowed to execute when it does not have any active children.



(a) voLock in the baseline STM

(b) voLock in NesTM

Figure 1: Comparison of voLocks.

Transactional semantics: We describe the definition of conflict discussed in [2] for TM systems with closed nesting. For a memory object l, let readers(l) be a set of active transactions that have l in their read-sets. writers(l) is defined similarly. When a transaction T accesses l, the following two cases are conflicts:

- T reads from l: if there exists a transaction T' such that $T' \in$ writers(l), $T' \neq T$ and $T' \notin$ ancestors(T).
- T writes to l: if there exists a transaction T' such that $T' \in$ readers(l) $\cup$ writers(l), $T' \neq T$ and $T' \notin$ ancestors(T).

As for the commit semantics, if T is not a top-level transaction, its read- and write-sets are merely merged into its parent's read- and write-sets. Otherwise, all the values written by T become visible to other transactions and its read- and write-sets are reset. If T aborts, all the changes made by T are discarded and previous states are restored [16].

3. DESIGN AND IMPLEMENTATION OF NESTM

This section describes the NesTM algorithm, an execution example, and the main issues related to performance.

3.1 NesTM Algorithm

The key design goal of NesTM is to keep state and runtime overheads small in supporting nested parallel transactions. For instance, we do not want to significantly increase memory footprint by using multiple sets of locks and global version clocks to support multiple nested parallel regions. The blocking, eager versioning STM used as our baseline has a useful property that helps us meet our goal: once a transaction writes (i.e., acquires a lock) to a memory object, it is guaranteed to have an exclusive ownership for the object until it commits or aborts.

Before discussing the NesTM algorithm, we describe the changes in the version-owner locks (voLock) compared to the baseline STM. As shown in Figure 1, voLock in the baseline STM is a word-sized data structure (i.e., N=32 and 64 on 32-bit and 64-bit machines) that encodes the version or owner information on the associated memory object. If L=1 (locked), the remaining N-1 bits store the owner information. If L=0 (unlocked), the remaining N-1 bits store the version number. This encoding is sufficient to support only top-level transactions because once a transaction locks a memory object, no other transactions are allowed to access the object until the transaction commits or aborts. In NesTM, however, other transactions can correctly access the locked object as long as they are descendants of the owner. To allow this, the ownership information should always be available in voLock to consult the ancestor relationship at any time. Similarly, the version number in voLock should also be always available to serialize the conflicting transactions.

```
1: procedure IsInReadSet(Self, addr)
2:  acquireLock(Self.commitLock)
3:  result ← addr ∈ Self.RS
4:  releaseLock(Self.commitLock)
5:  return result
6: procedure DoomHighestConflictTx(Self, Owner)
7:  ptr ← Self
8:  while ptr.Parent ∉ {Root,Owner,Ances(Owner)} do
9:   ptr.doomed ← true
10:  ptr ← ptr.Parent
11: procedure ValidateReaders(Self, Owner, addr)
12:  ptr ← Self
13:  hcr ← NIL
14:  while ptr ∉ {Root,Owner,Ances(Owner)} do
15:   if getTS(addr) > ptr.rv and isInReadSet(ptr, addr) then
16:    ptr.doomed ← true
17:    hcr ← ptr
18:   ptr ← ptr.Parent
19:  return hcr
20: procedure TxStart(Self)
21:  Self.aborts ← 0
22:  checkpoint()
23:  if isAnyDoomedAnces(Self) then
24:   return fail
25:  Self.doomed ← false
26:  Self.rv ← GlobalClock
27:  return success
28: procedure TxLoad(Self, addr)
29:  if Self.doomed = true or isAnyDoomedAnces(Self) then
30:   TxAbort(Self)
31:  retry_load:
32:  rb ← RollbackCounter
33:  cv ← getVoVal(addr)
34:  Owner ← extractOwn(cv)
35:  value ← Memory[addr]
36:  if Owner = Self then
37:   Self.RS.insert(addr)
38:   return value
39:  else if Owner∈Ances(Self) and cv = getVoVal(addr) then
40:   if rb ≠ RollbackCounter then
41:    goto retry_load
42:   if extractTS(cv) > Self.rv then
43:    TxAbort(Self)
44:   else
45:    Self.RS.insert(addr)
46:    return value
47:  else
48:   if Owner∉Ances(Self) and Self.aborts%p = p − 1 then
49:    DoomHighestConflictTx(Self, Owner)
50:   TxAbort(Self)
51: procedure TxAbort(Self)
52:  Self.doomed ← false
53:  Self.aborts ← Self.aborts + 1
54:  Self.RS.reset()
55:  atomicIncrementRollbackCounter()
56:  for all e in Self.WS do    ▷ Traversing direction: backward
57:   Memory[e.addr] ← Self.WS.lookup(e.addr)
58:  for all e in Self.WS do    ▷ Traversing direction: forward
59:   Owner ← getOwner(e.addr)
60:   if Owner = Self then
61:    setVoVal(e.addr, e.prevVoLock)
62:  Self.WS.reset()
63:  doContentionManagement()
64:  restoreCheckpoint()
```

Algorithm 1: Pseudocode for the basic functions in NesTM.

```
1: procedure TxStore(Self, addr, data)
2:  if Self.doomed = true or isAnyDoomedAnces(Self) then
3:   TxAbort(Self)
4:  Owner ← getOwner(addr)
5:  if Owner = Self then
6:   cv ← getVoVal(addr)
7:   Self.WS.insert(addr, Memory[addr], cv)
8:   Memory[addr] ← data
9:  else
10:  cnt ← 1
11:  repeat
12:   cv ← getVoVal(addr)
13:   ov ← cv
14:   nv ← extractTS(cv) | Self.TID
15:   if extractOwner(cv) ∈ Ances(Self) then
16:    ov ← atomicCAS(getVoAddr(addr), cv, nv)
17:    if ov = cv then
18:     hcr←ValidateReaders(Self,extractOwner(cv),addr)
19:     if hcr ≠ NIL then
20:      setVoVal(addr, cv)
21:      TxAbort(Self)
22:     Self.WS.insert(addr, Memory[addr], cv)
23:     Memory[addr] ← data
24:     return
25:   cnt ← cnt + 1
26:  until cnt = C
27:  if Self.aborts % p = p − 1 then
28:   DoomHighestConflictTx(Self, extractOwner(ov))
29:  TxAbort(Self)

30: procedure TxCommit(Self)
31:  if Self.doomed = true or isAnyDoomedAnces(Self) then
32:   TxAbort(Self)
33:  wv ← Fetch&Increment(GlobalClock)
34:  acquireLock(Self.Parent.commitLock)
35:  for all e in Self.RS do
36:   cv ← getVoVal(e.addr)
37:   Owner ← extractOwner(cv)
38:   if Owner = Self then
39:    continue
40:   else if Owner ∈ Ances(Self) then
41:    if extractTS(cv) > Self.rv then
42:     releaseLock(Self.Parent.commitLock)
43:     TxAbort(Self)
44:   else
45:    releaseLock(Self.Parent.commitLock)
46:    if Self.aborts % p = p − 1 then
47:     DoomHighestConflictTx(Self, Owner)
48:    TxAbort(Self)
49:  mergeRWSetsToParent(Self)
50:  releaseLock(Self.Parent.commitLock)
51:  for all e in Self.WS do
52:   Owner ← getOwner(e.addr)
53:   if Owner = Self then
54:    nv ← wv | Self.Parent.TID
55:    setVoVal(e.addr, nv)
56:  Self.RS.reset()
57:  Self.WS.reset()
```

Algorithm 2: Pseudocode for the basic functions in NesTM.

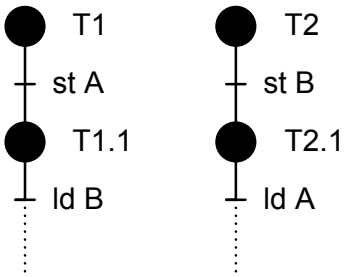


Figure 2: A livelock scenario avoided by eventual rollback of the outer transaction.

To enable this, we modify the voLock as shown in Figure 1. T least significant bits (LSBs) are used to encode the owner of the associated object. Since TID 0 is reserved for the root transaction, NesTM can support up to $2^T - 1$ concurrent transactions. While we use T=10 (i.e., 1023 transactions) in this paper, it is tunable. The remaining N-T bits store the version number. Since the global version clock increases by 2^T at the commit of each transaction, it can saturate faster than the baseline STM. Recent work discusses how to handle the version clock overflow [10].

Algorithms 1 and 2 provide the pseudocode for NesTM algorithm. We summarize the key functions below.

TxStart: This barrier is almost identical to the one in the baseline STM except that it returns "fail" when there are any doomed ancestors of the transaction we attempt to initiate. The return value can be used to restart the doomed ancestor in order to guarantee forward progress.

TxLoad: Following the conflict definition of nested parallel transactions in Section 2.2, a transaction can read a memory object only if the owner of the object is itself or its ancestor. When it is the owner, it can safely read the memory object without checking the version number (the reason will be explained in the discussion of TxStore). When the owner is its ancestor, it relies on the version number to ensure serializability. If the owner is neither itself nor its ancestor, the transaction conflicts with the owner. In lines 48–49 in Algorithm 1, it periodically calls DoomHighestConflictTx. This is to avoid potential livelock cases. Figure 2 illustrates an example. If only nested transactions (i.e., T1.1 and T2.1) abort and restart, none of them can make forward progress because the memory objects are still (crosswise) locked by ancestors. To avoid the livelock, at least one of the ancestors should abort and release the acquired memory objects. For this purpose, NesTM periodically checks and dooms ancestors. Note that we could use a more precise livelock detection mechanism, but it would also incur a large runtime overhead. Also note that similar livelock cases exist even in the baseline STM. Finally, note that *RollbackCounter* is used to avoid the *invalid-read* problem discussed in Section 4.1.

TxStore: When a transaction attempts to write to a memory object, it can safely do so if it is the owner of the memory object. Otherwise, it attempts to acquire the lock for the memory object, if the owner is an ancestor. If it fails, the transaction conflicts and DoomHighestConflictTx is also periodically called to avoid any potential livelock (lines 27–28 in Algorithm 2). If it successfully acquires the lock, it calls ValidateReaders with parameters consisting of Self, Owner (the previous owner for the object), and addr. In ValidateReaders, the transaction itself and all its ancestors that are also not an ancestor of Owner are validated for the object (lines 11–19 in Algorithm 1). The key insight of this is that once a transaction T or any of its descendants writes (i.e., acquires the lock) to a memory object, T is guaranteed to have an exclusive ownership for the object until it commits or aborts. Therefore, if we ensure that there were no conflicting writes to an object for T and all of its ancestors at the time when T first attempts to write to the object, the object is guaranteed to be valid throughout T and its ancestors' execution. If there is any invalid reader, it transfers the ownership to the previous owner and triggers rollback (lines 19–21 in Algorithm 2). Note that validating each transaction is protected by the *commit-lock* of that validated transaction to avoid the problem with non-atomic commit discussed in Section 4.2. Also, note that TxStore can be expensive when a transaction executes in a deep nesting level due to read-set search for itself and its ancestors. We will discuss this performance issue in Section 3.3.

TxCommit: If a transaction or any of its ancestors is doomed, it aborts (lines 31–32 in Algorithm 2). Otherwise, it validates all the entries in its read-set (lines 35–48 in Algorithm 2). Once the read-set is validated, it merges its read- and write-sets to its parent's (line 49 in Algorithm 2). Note that to avoid the problem with the non-atomic commit discussed in Section 4.2, the process of read-set validation and merging is protected by the commit-lock of the parent. To reduce the execution time in the critical section, merging is done by linking (instead of copying) the pointers in read- and write-sets implemented using linked-lists. Then, the version number and ownership for each object in the write-set are incremented and transferred to the parent.

TxAbort: After updating transactional metadata and incrementing RollbackCounter, the write-set is traversed backward (i.e., from the newest to oldest) to roll back the speculatively-written memory values. Then, the write-set is traversed forward (i.e., from the oldest to newest) to restore the value of voLock to the first observed value. Note that the voLock is released only when the owner of the memory location is the transaction itself (lines 60 in Algorithm 1) to avoid the double-release problem [22]. Finally, the checkpoint is restored to restart the transaction.

Note that by calling DoomHighestConflictTx in TxLoad, TxStore, and TxCommit, possible livelock scenarios similar to Figure 2 can be avoided. In addition, a randomized exponential backoff scheme is used for the contention management to probabilistically provide liveness.

3.2 Example

Figure 3 illustrates an example of how a simple application using nested parallel transactions executes on NesTM. Initially, GC=0 and TS(A)=TS(B)=0. Note that GC is incremented by 2^{10} in the real implementation. For simplicity, we assume GC is incremented by 1 in this and subsequent examples.

At (wall clock) time 0, T1 starts (RV(T1)=0). At time 2, T1 reads B. At time 3 and 4, T2 starts (RV(T2)=0) and writes to A. At time 5, T2 commits and GC=1 and TS(A)=1. At time 6, threads executing T1.1 and T1.2 (children of T1) are forked and T1.1 and T1.2 start (RV(T1.1)=RV(T1.2)=1). At time 7, both T1.1 and T1.2 successfully read A because RV(T1.1)=RV(T1.2)$\geq$TS(A). At time 8, T1.2 attempts to write to A. T1.2 validates itself and its ancestors (T1) by calling ValidateReaders. T1.2 is valid because A is in its read-set and RV(T1.2)$\geq$TS(A). T1 is not doomed because A is not in its read-set (read-sets of T1.1 and T1.2 have not been merged yet). Therefore, T1.2 can successfully write to A. At time 9, T1.2 successfully commits and GC=2 and TS(A)=2. Also, the read- and write-sets of T1.2 are merged into the ones of T1. At time 10, T1.1 attempts to commit but fails because A is in the read-set of T1.1 and RV(T1.1)$<$TS(A). GC is incremented to 3 due to this unsuccessful commit.

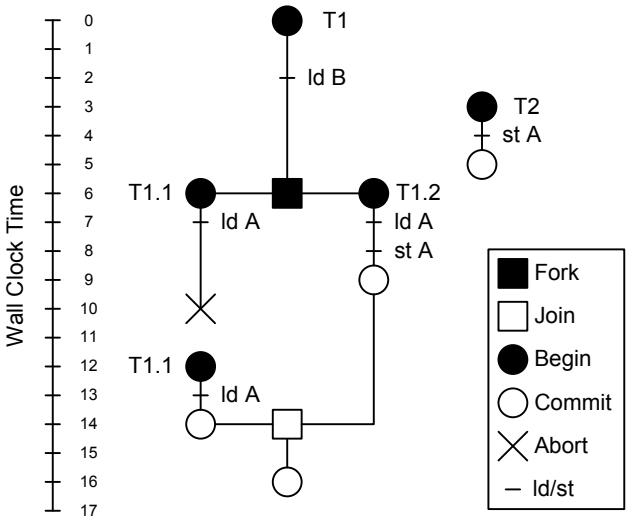


Figure 3: An example of a TM application running on NesTM.

At time 12, T1.1 restarts (RV(T1.1)=3). At time 13, T1.1 successfully reads A because the owner of A is T1 (an ancestor of T1.1) and RV(T1.1)≥TS(A). At time 14, T1.1 successfully commits, GC is incremented to 4 (but still TS(A)=2), and T1 resumes its execution after child threads join. At time 16, T1 successfully commits because it has an ownership for A (transferred from T1.2) and RV(T1)≥TS(B). GC and TS(A) are incremented to 5.

3.3 Qualitative Performance Analysis

Table 1 provides a symbolic comparison of the common- and worst-case time complexity of TM barriers in baseline STM and NesTM. NesTM has two different implementations: (1) NesTM-L: linked-lists are used to implement read- and write-sets; ancestor relationship is checked by pointer chasing and (2) NesTM-H: hash tables are used to implement read- and write-sets; ancestor bit vector (ABV) is used for fast ancestor relationship check. We assume the common case is a case in which the nesting depth is small and there is strong temporal locality between reads and writes (i.e., a transaction writes to a recently-read memory object). On the other hand, we assume the worst case is a case in which the nesting depth is large and there is weak temporal locality between reads and writes.

In the common case, the time complexity of NesTM-L TM barriers can be almost similar to the ones in the baseline STM because the nesting depth is small (i.e., $d \simeq 1$) and only a few entries in the read-set need to be looked up at each write to check the validity due to the strong temporal locality between reads and writes. However, in the worst case, the time complexity of NesTM-L TM barriers is significantly higher than the baseline STM. In contrast, NesTM-H still shows a comparable time complexity as the baseline STM due to the use of hash tables and ABV. Our current NesTM implementation follows NesTM-L; the implementation of NesTM-H is part of our future work.

In addition to the differences in the time complexity of TM barriers, there are three performance issues to note. First, temporal locality is lost when accessing transactional metadata of nested transactions. Since, when a child transaction commits, its read- and write-set entries are merged to its parent, there is no temporal locality for these entries when a new transaction begins on the same core. Second, the same memory objects in the read-set are repeatedly validated across different nesting levels. Finally, when a large number of child transactions simultaneously attempt to commit, contention on the commit-lock of the parent can become the critical performance bottleneck. We quantify these performance issues in Section 5.

4. COMPLICATIONS OF CONCURRENT NESTING

We now discuss subtle correctness issues we have encountered while developing NesTM. We also describe our on-going efforts on the correctness and liveness of NesTM.

4.1 Invalid Read

Problem: In the read barrier, reading a voLock and the corresponding memory value does not occur atomically. Because of this, eager STMs are potentially vulnerable to the *invalid-read* problem. A transaction may incorrectly read an invalid memory value speculatively written by an aborting transaction. If the aborting transaction restores the original voLock value, the validation process at the end of the reading transaction will miss the problem. In flat STMs, this problem can be simply avoided by always incrementing the timestamp values of voLocks even when an aborting transaction releases them. In NesTM, however, this technique cannot be used due to the *self-livelock* problem. If an aborting descendant increments the timestamp value of the voLock for a memory object, its ancestor that has the memory object in its read-set can be aborted due to that incremented timestamp value. Eventually, the subtree rooted by the ancestor cannot make any forward progress.

Solution: To correctly address both invalid-read and self-livelock problems at the same time, we propose the *RollbackCounter* scheme. On abort, a transaction atomically increases the global RollbackCounter in addition to restoring the values of voLocks in its write-set to the first observed values. When a transaction attempts to read a memory object, it first samples the value of RollbackCounter before reading the value of the associated voLock (line 32 in Algorithm 1). After ensuring the voLock value remains unchanged (line 39), the previously sampled value of RollbackCounter is compared with the current value. If the two values match, it is guaranteed that there has been no aborting transaction since the voLock value was read, thus no possibility of invalid read. If the two values differ, it conservatively avoids the invalid-read problem by retrying the whole process (line 41).

Performance impact: Since only a single, global RollbackCounter is used, false positives can degrade the performance by making transactions repeat the process several times even when they did not actually read invalid memory values. Furthermore, the extra code added to access the RollbackCounter in the read barrier can degrade the performance.

Possible alternatives: Instead of using the eager version management (VM) scheme, a lazy VM scheme can be used (while still using the encounter-lock scheme) to avoid the invalid-read problem. However, it can cause significant performance issues because the write-set of a transaction is frequently accessed by the transaction itself and its descendants.

4.2 Non-atomic Commit

Problem: Figure 4 illustrates a potential serializability violation scenario due to the non-atomic commit. Initially, GC and TS(A) are set to 0. After reading A at time 3, T1.1 initiates its commit at time 4. At time 5, A is validated. At time 6, T2 writes to A. At time 7, T2 commits and GC=1 and TS(A)=1. At time 8, T1.2 starts (RV(T1.2)=1). At time 10, T1.2 attempts to write to A. By calling ValidateReaders in Algorithm 1, T1.2 validates itself and

	Baseline	NesTM-L (Linked list)		NesTM-H (Hash + ABV)	
		Common Case	Worst Case	Common Case	Worst Case
Read	$O(1)$	$\sim O(1)$	$O(d)$	$\sim O(1)$	$O(d)$
Write	$O(1)$	$\sim O(1)$	$O(d \cdot (R+d))$	$\sim O(1)$	$O(d)$
Commit	$O(R+W)$	$\sim O(R+W)$	$O(d \cdot R+W)$	$\sim O(R+W)$	$O(d+R+W)$

Table 1: A symbolic comparison of the common- and worst-case time complexity of TM barriers in baseline STM and NesTM. R, W, and d denote read-set size, write-set size, and nesting depth, respectively.

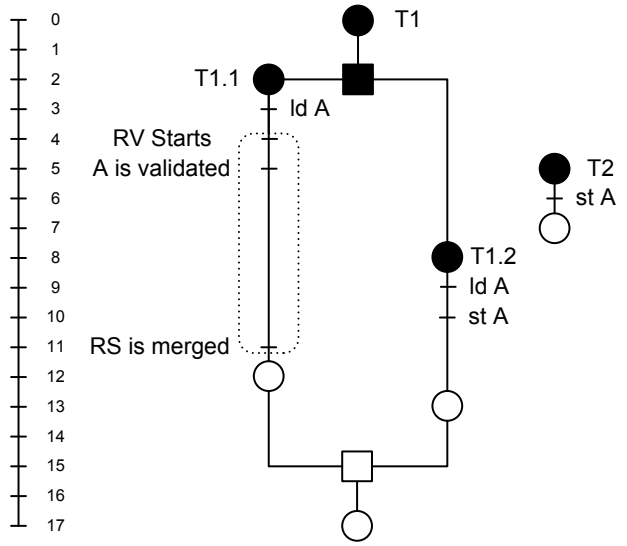


Figure 4: A potential serializability violation scenario due to the non-atomic commit.

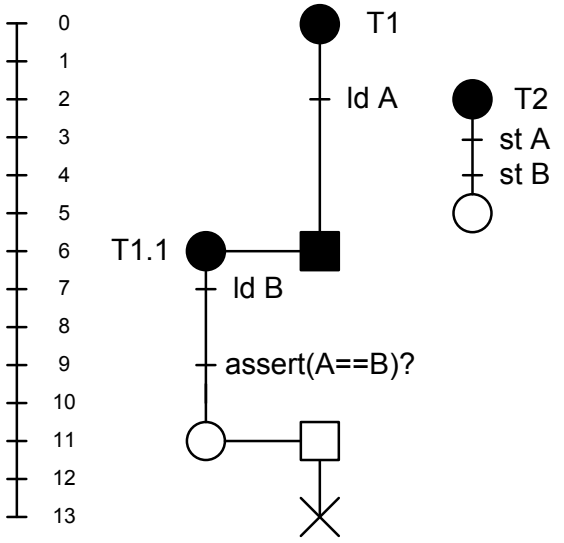


Figure 5: A problematic scenario due to a zombie transaction.

its ancestors (i.e., T1). T1.2 is valid because RV(T1.2)≥TS(A). T1 is not doomed because A is not yet in its read-set (i.e., T1.1's read-set has not been merged yet). Therefore, T1.2 can successfully write to A. At time 11, T1.1 merges its read-set to its parent's. At time 17, T1 successfully commits because it has an ownership for A (transferred from T1.2). However, this violates serializability because T1 eventually commits even when the two reads by T1.1 and T1.2 observe different versions of A.

Solution: The cause of this problem is that the commit process of T1.1 does not appear atomic to T1's descendants that validate T1 by calling ValidateReaders. To address this problem, we propose the *commit-lock* scheme. With this scheme, when a nested transaction attempts to commit, it must acquire the commit-lock of its parent. In addition, when a descendant validates its ancestor by calling ValidateReaders, it must also acquire the commit-lock of the validated ancestor. This ensures that the commit process of a transaction's child appears atomic to a validating descendant of the transaction. In the previous example, with the commit-lock scheme, T1.1's commit either happens *before* or *after* the validation by T1.2. In the first case, T1 will be doomed because RV(T1)<TS(A) and eventually aborted. In the second case, T1.1 will be aborted because A is owned by T1.2 when T1.1 attempts to commit. Therefore, no serializability violation occurs in both cases.

Performance impact: The commit-lock scheme essentially serializes the commits of child transactions. When a large number of child transactions simultaneously attempt to commit, performance can be hugely degraded due to the serialized commit.

Possible alternatives: We could also address this problem by introducing a *ValidationCounter* to each transaction. The ValidationCounter increments every time when a transaction is validated by its descendant. When a child transaction attempts to commit, it samples the value of ValidationCounter of the parent. It then validates its read-set *without* acquiring the commit-lock of the parent. After the read-set validation, it acquires the commit-lock of the parent. It then compares the previously sampled value of ValidationCounter with the current value. If the two values match, it can safely merge its read-set to its parent's because it is guaranteed that there has been no validation by any descendant of the parent. If the two values differ, it releases the commit-lock of the parent and conservatively repeats the whole process. An evaluation of this alternative is left as future work.

4.3 Zombie Transactions

Problem: Figure 5 illustrates a problematic scenario due to a zombie transaction. Initially, GC=0 and TS(A)=TS(B)=0. At time 0, T1 starts (RV(T1)=0). At time 2, T2 starts (RV(T2)=0). Then, T2 writes to A and B at times 3 and 4. At time 5, T2 commits and GC=1 and TS(A)=TS(B)=1. At time 6, T1.1 starts (RV(T1.1)=1). At time 7, T1.1 can successfully read B because B's owner is the root and RV(T1.1)≥TS(B). However, if a programmer assumes that A is always equal to B within transactions and inserts an assertion check, the program will be unexpectedly terminated by failing the assertion check. Note that if T1 could reach to its commit, it would eventually abort, thus no serializability violation. Other well-known anomalies such as infinite loops can also occur. Currently, NesTM admits zombie transactions because we have not been able to find an efficient solution to avoid them in an unmanaged environment.

Feature	Description
Processors	In-order, single-issue, x86 cores
L1 Cache	64-KB, 64-byte line, private 4-way associative, 1 cycle latency
Network	256-bit bus, split transactions pipelined, MESI protocol
L2 Cache	8-MB, 64-byte line, shared 8-way associative, 10 cycle latency
Main Memory	100 cycles latency up to 8 outstanding transfers

Table 2: Parameters for the simulated CMP system.

4.4 Correctness Status

At this point, we do not have a hand proof of the correctness (serializability) and liveness of the NesTM algorithm. Therefore, the correctness and liveness of the NesTM algorithm still remain *unchecked*. However, we hope that our paper will generate in-depth discussions on formally proving and verifying correctness and liveness guarantees of timestamp-based, concurrently-nested STM.

To establish some evidence of correctness, we have subjected the NesTM algorithm to exhaustive tests using our model checker (ChkTM) [3] and simulator. ChkTM verifies every possible execution of a small TM program running on the NesTM model. We configured ChkTM to generate every possible program with four threads (i.e., [1, 2, 1.1, 1.2]), each running only one transaction that performs at most two transactional memory operations (i.e., read or write), each accessing one of the two shared-memory words. ChkTM then explored every possible interleaving of every possible program. ChkTM, thus far, has not reported any serializability violation. Currently, ChkTM fails to verify NesTM with larger configurations (e.g., more threads or memory operations) due to the state space explosion.

To check the correctness and liveness of NesTM for a larger configuration, we performed extensive random tests by running a small microbenchmark on the implemented NesTM algorithm and simulator. The microbenchmark runs 14 concurrent threads (i.e., [1, 2, 1.1, 1.2, 2.1, 2.2, 1.1.1, 1.1.2, 1.2.1, 1.2.2, 2.1.1, 2.1.2, 2.2.1, 2.2.2]), each running one transaction that performs at most four transactional reads or writes to two shared-memory words. To better expose any potential bugs, we injected random delays at various points in the NesTM code (e.g., between lines 33 and 35 in Algorithm 1). The serializability checker compares the values observed by each transactional read and the final memory state of a concurrent run of the test program with the ones produced in a serial schedule. If this check fails, the checker reports a serializability violation. The liveness checker checks whether the test program successfully terminates or not. So far, NesTM has passed more than one million consecutive random tests without reporting any serializability or liveness violation.

5. EVALUATION

5.1 Methodology

We use an execution-driven simulator for x86 multi-core systems. Table 2 summarizes architectural parameters. All operations, except for loads and stores, have a CPI of 1.0, however all the details in the memory hierarchy timings are modeled, including contention and queueing events. We use the simulation results as our main results because they allow us to report results for larger CMP configurations and provide detailed performance breakdowns without perturbing the results.

# Threads	G	I	K	L	S	V	Y
1	13.9	13.9	3.7	0.1	7.2	22.2	4.7
2	11.5	17.4	3.6	0.1	6.5	21.6	3.4
4	15.6	14.6	3.9	0.1	5.6	21.9	-0.5
8	11.1	17.3	5.6	0.0	3.4	20.9	3.0
16	4.8	16.4	16.1	3.7	0.1	20.9	5.5

Table 3: Normalized performance difference (%) of NesTM relative to the baseline STM for STAMP applications. G, I, K, L, S, V, and Y indicate genome, intruder, kmeans, labyrinth, ssca2, vacation, and yada, respectively.

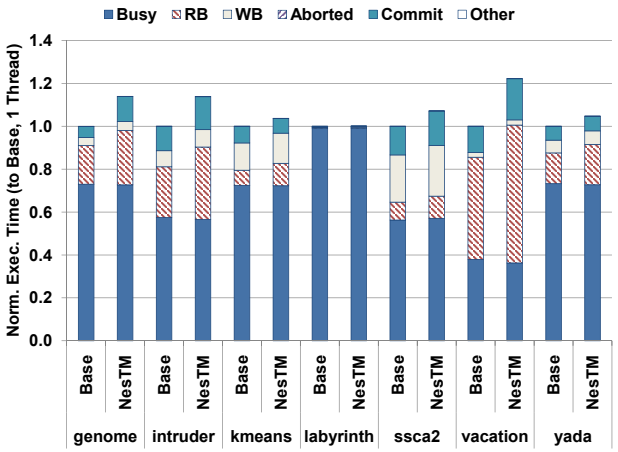


Figure 6: Execution time breakdowns of STAMP applications with 1 thread.

Our evaluation aims to answer the following three questions: **Q1**: What is the runtime overhead due to NesTM when we do not need nested parallelism (i.e., running only top-level transactions)? **Q2**: What is the incremental overhead if we push down the available parallelism to a deeper nesting level (NL)? **Q3**: How does nested parallelism improve application performance? Q1 and Q3 address the practicality of NesTM, while Q2 provides insights into the overheads and the issues that a nesting-aware runtime system should address.

For Q1, we use seven of the eight STAMP applications with the simulation datasets [6][1]. For Q2, we use two microbenchmarks that implement concurrent hash table (hashtable) and red-black tree (rbtree). Finally, for Q3, we use a microbenchmark, c-hashtable that uses composed hash tables. Further details on the benchmarks are provided later in this section.

5.2 Q1: Overhead for Top-Level Parallelism

Table 3 compares the baseline STM and NesTM running the STAMP benchmarks using only top-level transactions. It lists the normalized performance difference (NPD)[2] calculated using the following equation:

$$NPD(\%) = \frac{T_{NesTM} - T_{Base}}{T_{Base}} \times 100$$

Overall, Table 3 shows that the maximum NPD is about 20% across all benchmarks and thread counts. While NesTM barri-

[1]We exclude bayes because its non-deterministic behavior makes it difficult to compare results across STMs.
[2]A positive NPD means that NesTM is slower.

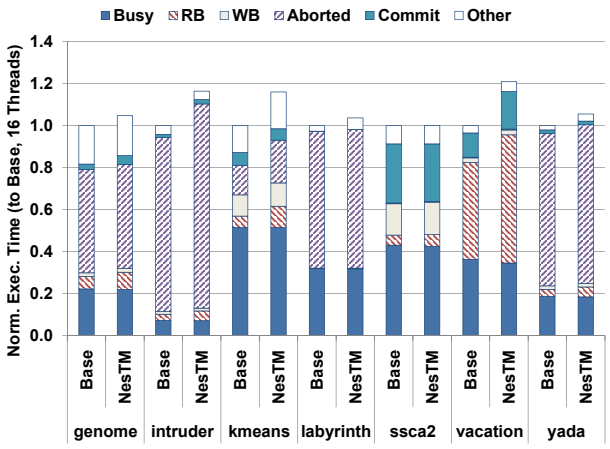


Figure 7: Execution time breakdowns of STAMP applications with 16 threads.

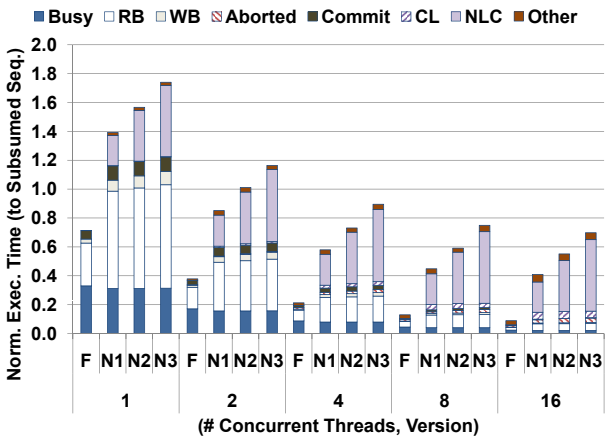


Figure 8: Execution time breakdowns of hashtable.

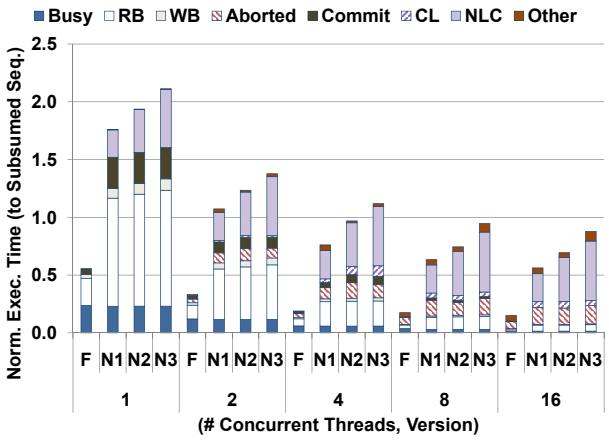


Figure 9: Execution time breakdowns of rbtree.

ers have additional code, some of it can be conditionally skipped when top-level transactions are used. To investigate the exact overheads further, we show the execution time breakdowns of STAMP applications in Figures 6 and 7. The execution time of each application is normalized to the execution time on the baseline STM with 1 (Figure 6) and 16 (Figure 7) threads, respectively. Execution time is broken into "busy" (useful instructions and cache misses), "RB" (read barriers), "WB" (write barriers), "aborted" (time spent on aborted transactions), "commit" (commit overhead), and "other" (work imbalance, etc.).

With 1 thread (Figure 6), NPD is relatively high (i.e., NesTM is slower) when transactions include a large number of TM barriers (e.g., intruder, vacation) [6]. This is mainly due to the extra overhead in NesTM barriers that cannot be amortized in this case. On the other hand, the overhead is negligible when very large transactions with few TM barriers are used (e.g., labyrinth). With more threads (Figure 7), more time is spent on aborted transactions with several applications (e.g., intruder, kmeans, yada). This is due to the validation that NesTM performs at the first write to each variable. This extra validation often detects conflicts more aggressively than the baseline STM, leading to more time spent on aborted transactions.

5.3 Q2: Incremental Overhead of Deeper Nesting

To study the incremental overhead of pushing down the available parallelism to deeper nesting levels (NLs), we use two microbenchmarks. hashtable and rbtree perform concurrent accesses to a hash table with 4K buckets and a red-black tree. Among 4K operations, 12.5% are inserts (writes) and 87.5% are look-ups (reads). Each benchmark has 4 versions. flat uses only top-level transactions, each performing 16 operations (hashtable) and 4 operations (rbtree). N1 pushes down the parallelism to NL=1, using the same code enclosed with one big outermost transaction[3]. N2 and N3 are implemented by adding more outer transactions in a repeated manner.

In Figures 8 and 9, we show the execution time breakdowns of hashtable and rbtree. The execution time of each microbenchmark is normalized to the execution time on an STM that flattens and serializes nested transactions (i.e., performs all 4K operations sequentially in a top-level transaction). In addition to the segments explained in Section 5.2, each bar contains newly added segments: "CL" (time spent acquiring the commit locks of parents), and "NLC" (time spent committing non-leaf transactions).

We observe that NesTM continues to scale up to 16 threads. For example, N1 versions of hashtable and rbtree are faster than the subsumed version by 2.4× and 1.8× with 16 threads. Due to the larger number of conflicts, rbtree does not scale as well as hashtable. Figures 8 and 9 also reveal the three major performance challenges in NesTM. First, the runtime overhead of the read and write barriers of nested transactions is more expensive than those of top-level transactions. This is mainly due to more cache misses when accessing each entry in read- and write-sets. Since previously used entries in read- and write-sets of a transaction are merged to its parent, NesTM cannot exploit temporal locality on accessing transactional metadata when it runs nested transactions. In contrast, when top-level transactions are used, there is significant locality in metadata accesses. This performance issue might be mitigated using prefetching techniques.

Second, commit time increases linearly with the nesting level mainly due to the repeated read-set validation across different nesting levels. Alternatively, a runtime may choose different policies

[3]While flat and nested versions have different transactional semantics (i.e., whether to perform 4K operations atomically or not), we compare them to investigate performance issues.

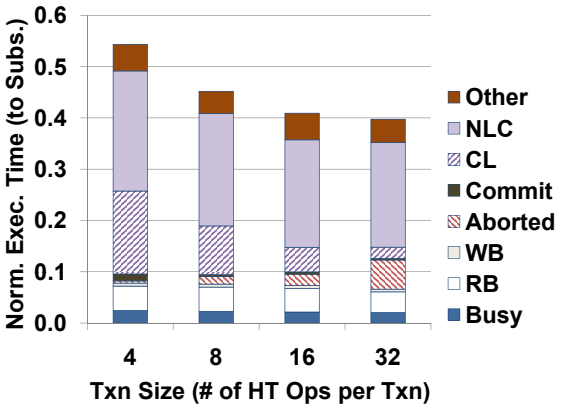


Figure 10: Execution time breakdowns of hashtable with various transaction sizes.

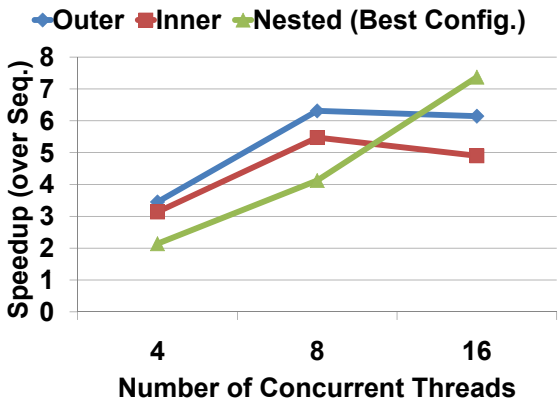


Figure 11: Scalability of the three versions of c-hashtable.

(e.g., serialization, reader-lock) depending on the nesting depth to achieve better performance. Finally, contention on the commit-locks of parents can become a performance bottleneck when a large number of nested transactions simultaneously commit. Since conflicts are infrequent in hashtable even with 16 threads, many child transactions can simultaneously commit and trigger this lock contention. In contrast, due to frequent conflicts in rbtree with 16 threads, this commit-lock contention is not a critical issue.

To understand the performance impact of transaction sizes, we measure the performance of hashtable by varying the transaction size from 4 to 32 operations per transaction. Figure 10 presents the normalized execution time with 16 threads. With smaller transactions (e.g., 4), a significant portion of the time is spent on the commit-lock contention because more (small) transactions simultaneously attempt to commit. With larger transactions (e.g., 32), the performance overhead due to the commit-lock contention is mitigated, while more time is spent on aborted (large) transactions.

To study how much work is required to amortize the overhead of nested transactions, we compare the performance of nested versions of hashtable with flat by varying the amount of computational workload in transactions. The amount of workload is proportional to the number of loop iterations. With little work, NPD is high due to the unamortized overhead of repeated read-set validation. One possible optimization is to use lightweight hardware support for validation [7]. With sufficient work, the overhead is amortized and nested versions comparably perform (e.g., N1: 39.7% with 1K iterations, N3: 9.9% with 10K iterations) to flat.

5.4 Q3: Improving Performance using Nested Parallelism

c-hashtable operates on a two-level structure with customer data with a single, first-level (L1) hash table and multiple, second-level (L2) hash tables. The L1 hash table stores customer information, and the L2 hash tables store customer orders. Each customer operation must be atomic including the updates to both levels. There are three ways in exploiting the parallelism in c-hashtable: (1) outer: parallelism in the L1 hash table across customers, (2) inner: parallelism in the L2 hash tables (multiple transactions from a single customer), and (3) nested: parallelism in both levels. Nested parallelism can be advantageous if each level alone (outer or inner) does not have sufficient parallelism to saturate a large-scale system.

In the experiment in Figure 11, the L1 hash table has 20 buckets and the L2 hash tables have 15 buckets. There are 256 randomly generated customers and each customer places 32 orders. The three lines in Figure 11 show the speedup of outer, inner, and nested over the sequential run without TM barriers. At lower thread counts (e.g., 4), outer performs best due to rare conflicts and low overhead (e.g., thread synchronization, coarse-grain transactions). With 16 concurrent threads, however, nested performs best by efficiently exploiting the parallelism at both levels. Scalability of the other versions is limited mainly due to frequent conflicts at higher thread counts.

6. RELATED WORK

Moss and Hosking discussed the reference model for closed and open nesting in transactional memory and described preliminary architectural sketches [16]. In addition, they proposed a simpler model called linear nesting in which nested transactions run sequentially. There has been previous work on supporting linear nesting in HTM [14, 15] and STM [12, 17]. Our work differs since NesTM targets concurrent nesting.

Recently, there has been research on supporting nested parallelism in STM [2, 4, 18, 22]. Agrawal et al. proposed CWSTM, a theoretical STM algorithm that supports nested parallel transactions with the lowest upper bound of time complexity [2]. In [4], Barreto et al. proposed a practical implementation of the CWSTM algorithm. While achieving depth-independent time complexity of TM barriers, their work builds upon rather complex data structures such as concurrent stacks that could introduce additional runtime (especially to top-level transactions) and state overheads [4]. In contrast, NesTM extends a timestamp-based STM. Ramadan and Witchel proposed SSTM, a lazy STM-based design that supports nested parallel transactions [18]. However, their work extends a lazy STM and does not provide a detailed performance analysis. Our algorithm differs by extending an eager STM that has lower baseline overheads. Finally, Volos et al. proposed NePaLTM that supports nested parallelism inside transactions [22]. While efficiently supporting nested parallelism when no or low transactional synchronization is used, NePaLTM serially executes nested parallel transactions using mutual exclusion locks. In contrast, NesTM implements concurrent execution of nested transactions.

7. CONCLUSION AND FUTURE WORK

This paper presented NesTM, an STM system that extends a state-of-the-art eager STM with closed-nested parallel transactions. NesTM is designed to keep state and runtime overheads small. We

also discussed the subtle corner cases of concurrent nesting. Finally, we evaluated the performance of NesTM across multiple scenarios. Our future work will focus on a more rigorous correctness argument. We will also investigate how to improve the performance of NesTM by exploring alternative implementations, nesting-aware contention management, and lightweight hardware support.

Acknowledgements

We would like to thank Richard Yoo and the anonymous reviewers for their feedback. We also want to thank Sun Microsystems for making the TL2 code available. Woongki Baek was supported by a Samsung Scholarship and an STMicroelectronics Stanford Graduate Fellowship. This work was supported by NSF Awards number 0546060, the Stanford Pervasive Parallelism Lab, and the Gigascale Systems Research Center (GSRC).

8. REFERENCES

[1] The OpenMP Application Program Interface Specification, version 3.0. `http://www.openmp.org`, May 2008.

[2] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174, New York, NY, USA, 2008. ACM.

[3] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and Evaluating a Model Checker for Transactional Memory Systems. In *ICECCS '10: Proceedings of the 15th IEEE International Conference on Engineering of Complex Computing Systems*, March 2010.

[4] J. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka. Leveraging parallel nesting in transactional memory. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 91–100, New York, NY, USA, 2010. ACM.

[5] R. Blikberg and T. Sorevik. Load balancing and OpenMP implementation of nested parallelism. *Parallel Comput.*, 31(10-12):984–998, 2005.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[7] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. June 2007.

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

[9] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.

[10] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[11] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[12] T. Harris and S. Stipic. Abstract nested transactions. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[14] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, June 2006. IEEE Computer Society.

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

[16] J. E. B. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. University of Rochester, October 2005.

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

[18] H. E. Ramadan and E. Witchel. The Xfork in the Road to Coordinated Sibling Transactions. In *The Fourth ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 09)*, February 2009.

[19] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

[20] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, Nov. 2001.

[21] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *LCR '00: Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 100–112, London, UK, 2000. Springer-Verlag.

[22] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *ECOOP*, 2009.