Making Nested Parallel Transactions Practical using Lightweight Hardware Support

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

ABSTRACT

Transactional Memory (TM) simplifies parallel programming by supporting parallel tasks that execute in an atomic and isolated way. To achieve the best possible performance, TM must support the nested parallelism available in real-world applications and supported by popular programming models. A few recent papers have proposed support for nested parallelism in software TM (STM) and hardware TM (HTM). However, the proposed designs are still impractical, as they either introduce excessive runtime overheads or require complex hardware structures.

This paper presents filter-accelerated, nested TM (FaNTM). We extend a hybrid TM based on hardware signatures to provide practical support for nested parallel transactions. In the FaNTM design, hardware filters provide continuous and nesting-aware conflict detection, which effectively eliminates the excessive overheads of software nested transactions. In contrast to a full HTM approach, FaNTM simplifies hardware by decoupling nested parallel transactions from caches using hardware filters. We also describe subtle correctness and liveness issues that do not exist in the non-nested baseline TM.

We quantify the performance of FaNTM using STAMP applications and microbenchmarks that use concurrent data structures. First, we demonstrate that the runtime overhead of FaNTM is small (2.3% on average) when applications use only single-level parallelism. Second, we show that the incremental performance overhead of FaNTM is reasonable when the available parallelism is used in deeper nesting levels. We also demonstrate that nested parallel transactions on FaNTM run significantly faster (e.g., 12.4×) than those on a nested STM. Finally, we show how nested parallelism is used to improve the overall performance of a transactional microbenchmark.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – parallel programming; C.1.4 [**Processor Architectures**]: Parallel Architectures

ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.

General Terms

Algorithms, Design, Performance

Keywords

Transactional Memory, Nested Parallelism, Parallel Programming

1. INTRODUCTION

Transactional Memory (TM) [11] has been proposed as a promising solution to simplify parallel programming. With TM, programmers can simply declare parallel tasks as *transactions* that appear to execute in an atomic and isolated way. TM manages all concurrency control among concurrent transactions. A large number of TM implementations have been proposed based on hardware [9, 13], software [8, 10, 17], and hybrid [6, 7, 16] techniques.

To date, most TM systems have assumed sequential execution of the code within transactions. However, real-world parallel applications often include *nested parallelism* in various forms including nested parallel loops, calls to parallel libraries, and recursive function calls [19]. To achieve the best possible performance with the increasing number of cores, it is critical to fully exploit the parallelism available at all levels. Several popular programming models that do not use transactions have already incorporated nested parallelism [1, 18]; TM should be extended to efficiently support the case of nested parallelism.

A few recent papers investigated the semantics of concurrent nesting and proposed prototype implementations in STM [2–4, 15, 21]. While compatible with existing multicore chips, most STM implementations already suffer from excessive runtime overheads of TM barriers even for single-level parallelism [6]. To make the problem worse, supporting nested parallelism solely in software may introduce additional performance overheads due to the use of complicated data structures [2, 4] or the use of an algorithm whose time complexity is proportional to the nesting depth [3]. For example, as shown in our performance evaluation, a single-threaded, transactional version of the red-black tree microbenchmark runs 6.2× slower with single-level transactions and 17.0× slower with nested transactions than a non-transactional, sequential version. Nested parallel transactions in STM will remain impractical unless these performance issues are successfully addressed.

A recent paper investigated how to support nested parallelism in HTM [20]. However, supporting nested parallelism solely in hardware may drastically increase hardware complexity, as it requires intrusive modifications to caches. For instance, apart from the additional transactional metadata bits in tags, the design proposed in [20] requires that caches are capable of maintaining multiple blocks with the same tag but different version IDs, and provide version-combining logic that merges speculative data from multi-

ple ways. Given the current trend in which hardware companies are reluctant to introduce complicated hardware components to implement transactional functionality even for single-level parallelism, this hardware-only approach is unlikely to be adopted.

To address this problem, we propose *filter-accelerated, nested transactional memory* (FaNTM) that provides practical support for nested parallel transactions using hardware filters. FaNTM extends a baseline hybrid TM (SigTM) [6] to implement nesting-aware conflict detection and data versioning. Since hardware filters provide continuous, nesting-aware conflict detection, FaNTM effectively reduces the excessive runtime overheads of software nested transactions. In contrast to a full HTM approach, FaNTM simplifies hardware by decoupling nested transactions from caches. As a result, FaNTM makes nested parallel transactions practical in terms of both performance and implementation cost.

The specific contributions of this work are:

- We propose FaNTM, a hybrid TM system that supports nested parallel transactions with low overheads. FaNTM provides eager data versioning and conflict detection at cache-line granularity across nested parallel transactions.
- We describe subtle correctness and liveness issues such as a *dirty-read* problem that do not exist in the non-nested baseline TM. We also propose solutions to address the problems.
- We quantify the performance of FaNTM across multiple use scenarios. First, we demonstrate that the runtime overhead of FaNTM is small when applications use only single-level parallelism. Specifically, FaNTM is slower than the baseline hybrid TM by 2.3% on average when running STAMP applications. Second, we show that the incremental overhead of FaNTM for deeper nesting is reasonable. We also show that nested transactions on FaNTM run significantly faster (e.g., 12.4×) than those on a nested STM. Finally, we demonstrate how FaNTM improves the performance of a transactional microbenchmark using nested parallelism.

The rest of the paper is organized as follows. Section 2 reviews the semantics of concurrent nesting and TM systems. Section 3 presents FaNTM. Section 4 discusses subtle correctness and liveness issues. Section 5 quantifies the performance of FaNTM. Section 6 reviews related work, and Section 7 concludes the paper.

2. BACKGROUND

2.1 Semantics of Concurrent Nesting

We discuss only a few concepts for concurrent nesting [3]. We refer readers to [2, 14] for additional discussions.

Definitions and concepts: Each transaction is assigned a *transaction ID (TID)*, a positive integer. No concurrent transactions can have the same TID. The *Root* transaction (TID 0) represents the globally-committed state of the system. *Top-level transactions* are the ones whose parent is the root transaction. Following the semantics in [14], we assume that a transaction does not perform transactional operations concurrently with any of its (live) descendants. Finally, $\mathsf{family}(T)$ of a transaction T is defined as a union of $\mathsf{ancestors}(T)$ and $\mathsf{descendants}(T)$.

Transactional semantics: For a memory object l, $\mathsf{readers}(l)$ is defined as the set of active transactions that have l in their read-sets. $\mathsf{writers}(l)$ is defined similarly. When T accesses l, the following two cases are conflicts:

- If T reads from l, it is a conflict if there exists T' such that $T' \in \mathsf{writers}(l)$, $T' \neq T$ and $T' \notin \mathsf{ancestors}(T)$.
- If T writes to l, it is a conflict if there exists T' such that $T' \in \mathsf{readers}(l) \cup \mathsf{writers}(l)$, $T' \neq T$ and $T' \notin \mathsf{ancestors}(T)$.

Field	Description
TID	T's TID
FV	A bit vector that encodes family(T). If a bit is set, the corresponding transaction belongs to family(T)
CTID	The TID of the transaction that conflicted with T
RSig	Read signature
WSig	Write signature
abt	If set, T has a pending abort.
act	If set, this TMB is the active TMB.
nackable	If set, the nackable bit in outgoing memory requests is set.

Table 1: State information stored in each TMB. T denotes the transaction that is mapped on the TMB.

If a committing transaction T is not a top-level transaction, its read- and write-sets are merged to its parent. Otherwise (i.e., top-level), the values written by T become visible to other transactions. If any transaction T aborts, all the changes made by T are discarded and previous state is restored [14].

2.2 NesTM

We use NesTM [3] as a proxy for a timestamp-based STM with support for concurrent nesting. While it is an open research issue to formally check the correctness and liveness guarantees of timestamp-based nested STMs, we use NesTM to investigate performance differences between software and hybrid nested TMs. We only provide a brief description and refer to [3] for additional information on NesTM.

NesTM [3] extends an eager variant of TL2 [8] to support concurrent nesting. NesTM uses a global version clock to establish serializability. Each memory word is associated with a version-owner lock that simultaneously encodes the version and owner information. Transactional metadata and barriers are extended to implement nesting-aware conflict detection and data versioning.

Since all the nesting-aware transactional functionality is solely implemented in software, NesTM introduces substantial runtime overheads to nested transactions. One of the critical performance bottlenecks of NesTM is repeated read-set validation, where the same memory object must be repeatedly validated across different nesting levels [3]. Since this performance overhead increases linearly with the nesting depth, it limits the applications for which NesTM can improve performance [2, 3]. Furthermore, NesTM barriers are more complicated, impacting performance even for top-level transactions. We quantify the runtime overheads of NesTM in Section 5.3.

2.3 Baseline Hybrid TM

As our starting point, we use an eager-versioning hybrid TM that follows the SigTM design [5,6]. It uses hardware signatures to conservatively track read- and write-sets of each transaction. Hardware signatures provide fast conflict detection at cache-line granularity by snooping coherence messages. Data versioning is implemented in software using undo-logs.

We chose eager versioning to avoid the runtime overheads of lazy versioning, which are higher for a nested TM. Since each update is buffered in the write buffer in a lazy TM, nested transactions must examine their parent's write buffer to handle reads that follow a speculative write. These accesses are expensive, because they must be synchronized with the changes to the parent's write buffer when a sibling transaction commits. An eager-versioning

Message	Contents	Description
RSigMerge	destTid, RSig	Merge the read signature in the destination TMB and the RSig in the packet.
WSigMerge	destTid, WSig	Merge the write signature in the destination TMB and the WSig in the packet.
remoteFvSet	destTid, FV	Set each bit in FV in the destination TMB if the corresponding bit is set in the FV in the packet.
remoteFvReset	destTid, FV	Reset each bit in FV in the destination TMB if the corresponding bit is set in the FV in the packet.

Table 2: Filter-request messages used in FaNTM.

Instruction	Description
`R/WSigReset`	Reset all the state in the read/write signature of the active TMB.
`R/WSigInsert` r1	Insert the address in register r1 in the read/write signature of the active TMB.
`R/WSigEn/Disable`	Enable/Disable the read/write signature of the active TMB to look up coherence messages.
`R/WSigEnableNack` r1	Configure the read/write signature of the active TMB to nack conflicting requests (their types in register r1).
`fetchExclusive` r1	Send an exclusive load request for the address in register r1 over the network.
`mergeR/WSig` r1	Send a R/WSigMerge request with a destination TID in register r1 over the network.
`get/setTid` r1	Get/Set the TID value of the active TMB in register r1.
`getConfTid` r1	Get the CTID value of the active TMB in register r1.
`set/resetNackable`	Set/Reset the nackable bit of the active TMB.
`localFvSet/Reset` r1	Set/Reset the FV entry associated to a TID in register r1.
`remoteFvSet/Reset` r1	Send a remoteFvSet/Reset request with a destination TID in register r1 over the network.
`increaseNL`	Increase nesting level (NL) by allocating the TMB at one NL above and setting it as the active TMB.
`decreaseNL`	Decrease NL by releasing the current active TMB and setting the TMB at one NL below as the active TMB.

Table 3: User-level instructions in FaNTM.

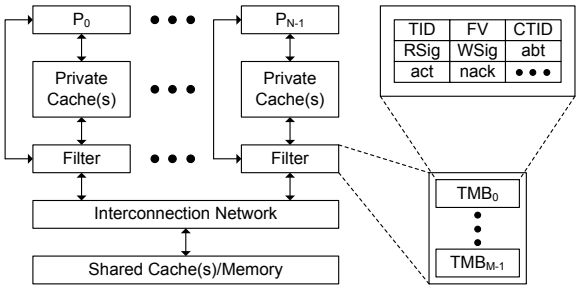


Figure 1: The overall architecture of FaNTM.

TM requires no such look-up. This is because memory holds the speculative value.

3. DESIGN OF FANTM

3.1 Overview

The baseline hybrid TM only supports single-level parallelism. Therefore, to support concurrent nesting, FaNTM hardware must be extended to provide nesting-aware conflict detection and retain multiple transactional contexts per processor core. FaNTM software must be extended to implement nesting-aware data versioning and handle liveness issues of nested transactions. This section presents our FaNTM design that implements high-performance nested parallel transactions in a manner that keeps hardware and software complexity low. We also provide a qualitative performance analysis on FaNTM.

3.2 FaNTM Hardware

Figure 1 shows the overall architecture of FaNTM. Each processor has a hardware filter. Filters are connected to the interconnection network to snoop coherence messages such as requests for shared/exclusive accesses and negative acknowledgements (nacks) to these requests. We assume that the underlying coherence protocol provides the nack mechanism. Filters may handle or propagate incoming messages to their associated cache depending on the characteristics of the messages. We also assume that each coherence message includes additional fields such as TID to encode the transactional information on the transaction that generated the message.

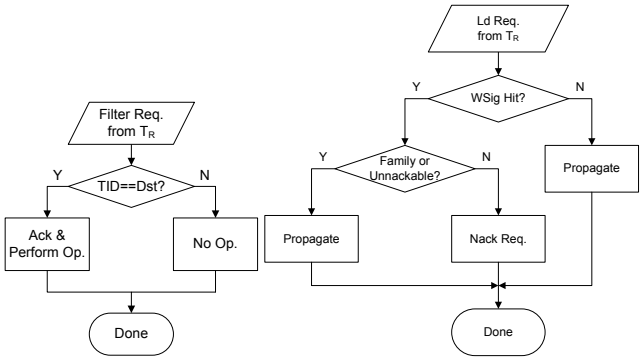
Each filter consists of a fixed number of *transactional metadata blocks* (TMBs) that summarize the state information of transactions mapped on the corresponding processor. The number of TMBs in the filter limits the number of transactions that can be mapped on each processor without the need for virtualization. When nesting depth overflows, we currently rely on software solutions such as switching back to a nested STM [3, 4, 15] or subsuming (i.e., serializing and flattening) nested transactions to avoid increasing hardware complexity. We leave hardware techniques for depth virtualization as future work.

Table 1 summarizes the state information stored in each TMB. We discuss how each field is used as we describe FaNTM operations later.

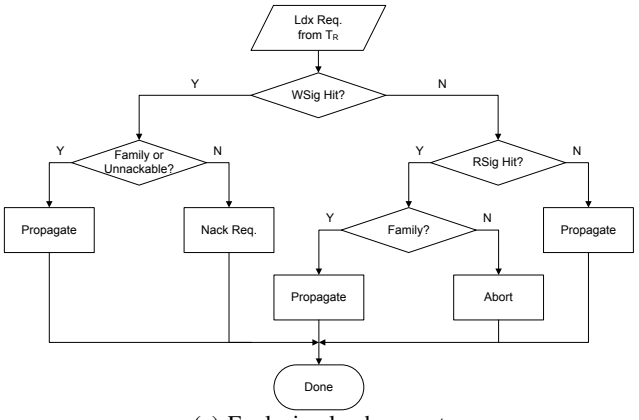
Table 2 summarizes filter-request messages. R/WSigMerge messages are used when committing transactions remotely merge their read/write signatures to their parent. remoteFvSet/Reset messages are used when nested transactions remotely update their ancestor's FV. Note that every filter-request message is intercepted by filters, thus no need to modify the caches for filter-request messages.

Table 3 summarizes user-level instructions used to manipulate filters. The TID-related instructions are used to manipulate the TID of the active TMB. Outgoing memory requests are associated with this TID. The FV-related instructions are used to update the transactional hierarchy information. The `in/decreaseNL` instructions are used to switch TMBs when a nested transaction executes on the same core where its parent was running. We provide details on the other instructions as we describe the FaNTM algorithm in Section 3.3.

Figure 2 illustrates common-case TMB operations when receiving coherence messages. On receiving a filter-request message with a matching destination TID (Figure 2(a)), the TMB performs the requested operation. On receiving a shared-load request (Figure 2(b)), the TMB nacks the request if the requested address is contained in its write signature and the requesting transaction does



(a) Filter request (b) Shared-load request



(c) Exclusive-load request

Figure 2: Flowcharts of common-case TMB operations.

not belong to its family. Note that the TMB does not nack the request if the nackable bit in the message is reset, which will be further discussed in Section 4.3. Otherwise, the TMB transfers the request to the associated cache. On receiving an exclusive-load request (Figure 2(c)), the TMB nacks it if the request satisfies the aforementioned nacking conditions. If the requested address is contained in the read signature (but not in the write signature), the following two cases are considered. First, if the request is from its family, the TMB propagates the request to the associated cache without aborting the transaction. Otherwise, the TMB sets its abt bit to eventually abort the transaction, disables its read signature to prevent repeated aborts, and resets its nackable bit to avoid deadlock (Section 4.3).

Figure 3 illustrates an example execution of a simple FaNTM program. T1 and T2 are top-level transactions running on P0 and P1. T3 is T1's child transaction running on P2. From step 0 to 2, T1 and T2 have performed transactional memory accesses to x and y. At step 3, P2 sends an exclusive load request prior to updating y. Since the request is from T1's family, Filter 0 simply propagates the request to the associated cache without aborting T1. On the other hand, since the request is not from T2's family (R/W conflict), Filter 1 interrupts P1 to abort T2. The exclusive load request by P2 is successful (not nacked). Therefore, P2 acquires exclusive ownership for the cache line holding y and proceeds with the execution of T3.

3.3 FaNTM Software

Figure 4 presents the transaction descriptor, a software data structure that summarizes the transactional metadata. Each transaction

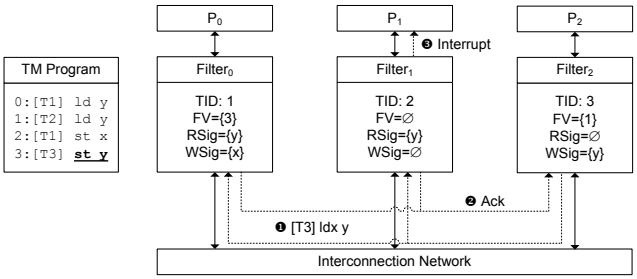


Figure 3: An example execution of a simple FaNTM program.

```
struct transaction {
 int Tid;
 Log UndoLog;
 struct transaction* Parent;
 lock commitLock;
 bool Doomed;
 bool Active;
 int Aborts;
 ...
}
```

Figure 4: Transaction descriptor.

maintains an undo-log implemented using a doubly-linked list to provide eager versioning in software. It has a pointer to its parent transaction to access its parent's metadata as necessary. Each transaction maintains a commit-lock to synchronize concurrent accesses to its undo-log by its children. There are additional fields such as Doomed, Active, and Aborts that will be discussed later.

Algorithms 1 and 2 present the pseudocode for the FaNTM algorithm. We summarize its key functions below.

TxStart: After making a checkpoint, this barrier checks whether there is any doomed ancestor. If so, it returns "failure" to initiate recursive aborts to provide forward progress. Otherwise, it starts the transaction by initializing its metadata. Note that the nackable bit in the TMB is set to ensure that any conflicting memory access by the transaction is correctly handled.

TxLoad[1]: This barrier inserts the address in the read signature and attempts to read the corresponding memory object. If this load request is successful (not nacked), the read barrier returns the memory value. Otherwise, the processor is interrupted and the program control is transferred to the software abort handler (TxAbortHandler) to initiate an abort.

TxStore: This barrier inserts the address in the write signature. It then sends an exclusive load request for the address over the interconnection network using the fetchExclusive instruction (Table 3). If this request fails, the filter interrupts the processor to abort the transaction. Otherwise, the transaction inserts the current memory value into its undo-log and updates the memory object in-place.

TxCommit: This barrier first resets the nackable bit in the active TMB to handle the deadlock issues discussed in Section 4.3. If there is any doomed transaction in the hierarchy, a transaction

[1]Similar to TxStart, TxLoad (and TxStore) could also check doomed ancestors. There are three possible schemes to check doomed ancestors such as always, periodic, and never. The always and periodic schemes introduce a runtime overhead that increases linearly with the nesting depth. The never and periodic schemes admit an execution scenario in which a nested transaction keeps running (temporarily), even when its ancestor has been already doomed. In our performance evaluation (Section 5), we performed experiments with a FaNTM design that implements the never scheme.

```
1:  procedure DoomConflictingAnces(Self, T)
2:    p ← Self.Parent
3:    while p ≠ NIL and IsAnces(p,T) = false do
4:      p.Doomed ← true
5:      p ← p.Parent
6:    return

7:  procedure TxStart(Self)
8:    checkpoint()
9:    if isDoomedAnces(Self) then
10:     return failure
11:   Self.Doomed ← false
12:   Self.Active ← true
13:   enableRSigLookup()
14:   enableWSigLookupAndNack()
15:   setNackable()
16:   return success

17: procedure TxLoad(Self, addr)
18:   RSigInsert(addr)
19:   val ← Memory[addr]
20:   return val

21: procedure TxStore(Self, addr, data)
22:   WSigInsert(addr)
23:   fetchExclusive(addr)
24:   Self.UndoLog.insert(addr, Memory[addr])
25:   Memory[addr] ← data
26:   return

27: procedure TxAbortHandler(Self)
28:   if Self.Aborts%period = period − 1 then
29:     confTx ← getConfTx()
30:     DoomConflictingAnces(Self, confTx)
31:   if Self.Active = false then
32:     Self.Doomed ← true
33:   else
34:     initiateAbort(Self)
```

Algorithm 1: Pseudocode for the FaNTM algorithm.

```
1:  procedure TxCommit(Self)
2:    resetNackable()
3:    if isDoomedOrDoomedAnces(Self) then
4:      TxAbort(Self)
5:    if isTopLevel(Self) = false then
6:      mergeRSigToParent(Self)
7:      disableRSigLookup()
8:      mergeWSigToParent(Self)
9:      acquireCommitLock(Self.Parent)
10:     mergeUndoLogToParent(Self)
11:     releaseCommitLock(Self.Parent)
12:   else
13:     disableRSigLookup()
14:   RSigReset()
15:   WSigReset()
16:   disableWSigLookup()
17:   Self.UndoLog.reset()
18:   Self.Aborts ← 0
19:   return

20: procedure TxAbort(Self)
21:   resetNackable()
22:   disableRSigLookup()
23:   RSigReset()
24:   for all addr in Self.UndoLog do
25:     Memory[addr] ← Self.UndoLog.lookup(addr)
26:   WSigReset()
27:   disableWSigLookup()
28:   Self.UndoLog.reset()
29:   Self.Aborts ← Self.Aborts + 1
30:   Self.Doomed ← false
31:   doContentionManagement()
32:   restoreCheckpoint()
```

Algorithm 2: Pseudocode for the FaNTM algorithm.

aborts. Otherwise, a top-level transaction finishes its commit by simply resetting its metadata. A nested transaction merges its read signature into its parent by sending a RSigMerge message. The nested transaction should detect any potential conflict until it receives the ack from its parent for the RSigMerge message. After receiving the ack from its parent, the transaction can disable its read signature because its parent will detect any subsequent conflict on behalf of the transaction. The transaction then merges its write signature by sending a WSigMerge message and its undo-log to its parent. When merging its undo-log, the transaction must acquire its parent's commit-lock to avoid data races. To reduce the execution time in the critical section, undo-log entries are merged by linking the pointers (instead of copying the entries). Finally, the transaction finishes its commit by resetting the transactional metadata.

TxAbort: This barrier restores the speculatively written memory values. It then resets the transactional metadata including the write signature. After performing contention management (exponential backoff), the transaction restarts by restoring the checkpoint.

TxAbortHandler: This software interrupt handler is pre-registered. It is called when a processor is interrupted due to a conflict. To handle the liveness issue discussed in Section 4.2, an aborting transaction periodically dooms its ancestors by setting their Doomed variable. If the transaction is currently inactive (i.e., has live children), it sets its Doomed variable and defers the actual abort until it becomes active again. Otherwise, the transaction initiates an abort.

Barrier	TL2	SigTM	NesTM	FaNTM
Read	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Write	$O(1)$	$O(1)$	$O(d \cdot R)$	$O(1)$
Commit	$O(R+W)$	$O(1)$	$O(d+R+W)$	$O(d)$

Table 4: A symbolic comparison of the time complexity of performance-critical TM barriers in (eager) TL2, (eager) SigTM, NesTM, and FaNTM. R, W, and d denote read-set size, write-set size, and nesting depth, respectively.

3.4 Qualitative Performance Analysis

Table 4 presents a symbolic comparison of the time complexity of TM barriers in TL2, SigTM, nested STM (NesTM), and FaNTM. Note that all the TMs discussed here perform eager versioning. We assume that NesTM maintains a data structure used for fast ancestor relationship check. Time complexity of the read barrier in all TMs is $O(1)$[2]. Time complexity of the write barrier in NesTM is high ($O(d \cdot R)$) because a transaction should validate its ancestors before it updates a memory object. In contrast, all the other TMs still have $O(1)$ complexity. As for the commit barrier, SigTM has the fastest one because committing transactions simply reset their metadata. FaNTM has $O(d)$ complexity because it checks doomed ancestors. TL2 has $O(R+W)$ complexity because a committing transaction validates its read-set and releases the locks in its write-set. NesTM has slightly higher complexity ($O(d+R+W)$) as it checks doomed ancestors.

Apart from the differences in time complexity, there are three FaNTM performance issues. First, nested transactions cannot ex-

[2]We assume that transactions do not check doomed ancestors in read and write barriers in NesTM and FaNTM.

Assume that initially x==0

```
   // Running on P0    // Running on P1
0: Begin(T1)          ...
1: st x,1             ...
2: ...                Begin(T1.1)
3: ...                ld x // cache miss
4: ...                End(T1.1)
5: ...                // Th1.1 fin./Th2 starts
6: ...                Begin(T2)
7: ...                ld x // cache hit!
8: ...                End(T2)
9: // T1 aborts       ...
```

Can T2 observe x==1?

Figure 5: A dirty-read problem due to an unexpected cache hit.

ploit temporal locality when accessing undo-log entries. When a nested transaction commits, it merges its undo-log entries into its parent. Hence, temporal locality is lost for these entries when a new nested transaction starts on the same core. Second, when a large number of child transactions commit at the same time, contention on the commit-lock of their parent may degrade performance. Finally, the extra code in TM barriers (e.g., sending R/WSigMerge messages over the network when nested transactions commit, checking doomed ancestors) may introduce additional runtime overheads. We quantify their performance impact in Section 5.

4. COMPLICATIONS OF NESTING

In this section, we discuss subtle correctness and liveness issues encountered while developing FaNTM.

4.1 Dirty Read

Problem: The key assumption for guaranteeing the correctness of FaNTM is that any transactional memory access that conflicts with other transactions causes a cache miss. Filters then snoop the conflicting request and correctly resolve it. In the presence of nested parallel transactions, however, this assumption may not hold if threads are carelessly scheduled. Figure 5 illustrates a *dirty read* problem that may occur when this assumption does not hold. At step 1, T1 on P0 attempts to write to x. P0's cache acquires exclusive ownership for the line holding x and the corresponding lines in the other caches are invalidated. At step 3, T1.1 on P1 attempts to read x. This access causes a cache miss due to the prior invalidation. Since T1.1 belongs to T1's family, T1 acks the request. Therefore, the cache lines holding x in P0's and P1's caches are now in shared state. After T1.1 commits, another top-level thread is scheduled on P1 and executes T2. At step 7, T2 attempts to read x. While this access conflicts with T1, it cannot be nacked by T1 due to an unexpected cache hit. At step 8, T2 successfully commits even after it read a value speculatively written by T1, which is incorrect.

Solution: The root cause of this problem is that unexpected cache hits can occur when a potentially-conflicting transaction (T2 in Figure 5) runs on a processor where a nested transaction (T1.1) ran and its top-level ancestor (T1) has not been quiesced yet. To address this problem without requiring complex hardware, we rely on a software thread-scheduler approach. When the number of available processors in the system is no less than the number of threads in the application, the thread scheduler pins each thread on its dedicated processor[3]. If the number of processors is not large enough, the thread scheduler attempts to schedule a thread on a processor where a transaction whose family has been quiesced ran. If there is no such processor, the thread scheduler maps the thread to any processor. When the thread is about to start a transaction, it may defer its execution until the family of the previously-executed transaction is quiesced or invalidate the private cache to prevent unexpected cache hits. We leave an exhaustive exploration of the thread-scheduler approach as future work.

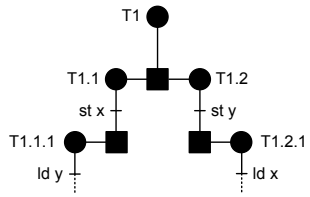


Figure 6: A livelock scenario avoided by eventual rollback of the outer transaction.

```
// T1 on P0    // T1.2 on P1
atomic {       work(arg) {
 ...            // Any mem. access can be
 fork(...);     // potentially nacked until
                // TID is set.
                setTid(tid);
}              }
```

Figure 7: A self-deadlock scenario caused by carelessly enforcing strong isolation.

4.2 Livelock

Problem: When a nested transaction detects a conflict, it only aborts and restarts (instead of aborting its ancestors) to avoid any unnecessary performance penalty. However, this can potentially cause livelock. Figure 6 illustrates an example. If only nested transactions (T1.1.1 and T1.2.1) abort and restart, none of them can make forward progress because the memory objects are still (crosswise) locked by their ancestors (T1.1 and T1.2).

Solution: To address this problem, a nested transaction periodically dooms its ancestors when executing TxAbortHandler (line 30 in Algorithm 1). Specifically, when a TMB detects a conflict, it records the TID of a conflicting transaction in its CTID. Using the CTID, the transaction periodically dooms every ancestor that is not the ancestor of the conflicting transaction. For instance, when T1.1.1 in Figure 6 is aborted by T1.2, it dooms its parent (T1.1) because T1.1 is not T1.2's ancestor. However, T1 is not doomed because it is T1.2's ancestor. If T1 is (carelessly) doomed, it causes a *self livelock* where a transaction cannot make forward progress as it is aborted by its descendants, even without any conflicting transaction.

4.3 Deadlock

Problem: In line with its baseline TM, FaNTM provides strong isolation where a transaction is isolated both from other transactions and non-transactional memory accesses [12]. However, carelessly enforcing strong isolation can cause various deadlock issues due to the inexact nature of signatures. Figures 7 and 8 illustrate potential deadlock scenarios. In Figure 7, any memory access by T1.2 on

[3]In our performance evaluation (Section 5), we assumed that the number of available processors is no less than the number of threads.

```
   // Running on P0          // Running on P1
0: Begin(T1)                 Begin(T2)
1: TxStore(x,1)              TxStore(y,2)
2: TxLoad(y)                 TxLoad(x)
3: // TxAbort                // TxAbort
4: access UndoLog(T1,x)      access UndoLog(T2,y)
5: // No progress           // No progress
```

Figure 8: A deadlock scenario caused by carelessly enforcing strong isolation.

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 Arbitration latency: 6 pipelined cycles
L2 Cache	8-MB, 64-byte line, shared 8-way associative, 10 cycle latency
Main Memory	100 cycle latency up to 8 outstanding transfers
Filters	4 TMBs per filter 2048 bits per R/W signature register The hash functions reported in [6]

Table 5: Parameters for the simulated CMP system.

P1 (before TID is set) can be potentially nacked by T1 on P0, if it generates a false positive in T1's write signature. T1.2 is nacked by T1 which re-activates only after T1.2 finishes, creating a cycle in the dependency graph. Thus, a self deadlock occurs.

Note that deadlock can occur even in the baseline hybrid TM. Figure 8 illustrates another deadlock scenario. T1 on P0 and T2 on P1 abort each other, after accessing memory locations (`x` and `y`) in a crosswise manner. When T1 and T2 attempt to restore memory objects, they can potentially deadlock, if their memory accesses are nacked by each other due to the false positives in write signatures.

Solution: To address this problem, we enforce the following two rules. First, by default, every non-transactional memory request is associated with TID 0 (the root transaction). Since, by definition, the root transaction belongs to the family of every transaction, filters do not respond to the request with TID 0. If strong isolation is desired, it should be explicitly enabled by associating non-transactional memory accesses with a non-zero TID using the `setTid` instruction in Table 3. Second, every memory request by a committing or aborting transaction resets its nackable bit. Filters do not nack this memory request even if it hits in their write signatures.

5. EVALUATION

5.1 Methodology

We used an execution-driven simulator for x86 multi-core systems. Table 5 summarizes the main architectural parameters. The processor model assumes that all instructions have a CPI of 1.0 except for the instructions that access memory or generate messages over the interconnection network. However, all the timing details in the memory hierarchy are modeled, including contention and queueing events.

Through our performance evaluation, we aim to answer the following three questions: **Q1**: What is the performance overhead of FaNTM when applications only use top-level parallelism? **Q2**: What is the performance overhead when the available parallelism is exploited in deeper nesting levels? **Q3**: How can we exploit nested parallelism to improve transactional application performance?

# T	G	I	K	L	S	V	Y	Avg.
1	2.1	2.5	0.9	0.1	6.2	0.5	0.4	1.8
2	4.3	2.8	0.5	0.2	6.2	0.8	4.3	2.7
4	5.0	5.9	1.0	1.9	2.7	1.0	0.3	2.5
8	4.1	1.8	1.5	2.5	4.7	1.6	0.4	2.4
16	4.5	0.0	2.4	2.0	2.3	0.3	1.1	1.8
Avg.	4.0	2.6	1.3	1.3	4.4	0.8	1.3	2.3

Table 6: Normalized performance difference (%) of FaNTM relative to SigTM for STAMP applications. G, I, K, L, S, V, and Y indicate genome**,** intruder**,** kmeans**,** labyrinth**,** ssca2**,** vacation**, and** yada**, respectively. The rightmost column and bottommost row present average values.**

For Q1, we used seven of the eight STAMP benchmarks [5] except for bayes as its non-deterministic behavior may make it difficult to compare the results across different TMs. For Q2, we used two microbenchmarks based on concurrent hash table (hashtable) and red-black tree (rbtree). For Q3, we used a microbenchmark (np-rbtree) that uses multiple red-black trees. We provide additional details on the benchmarks later in this section.

5.2 Q1: Overhead for Top-Level Parallelism

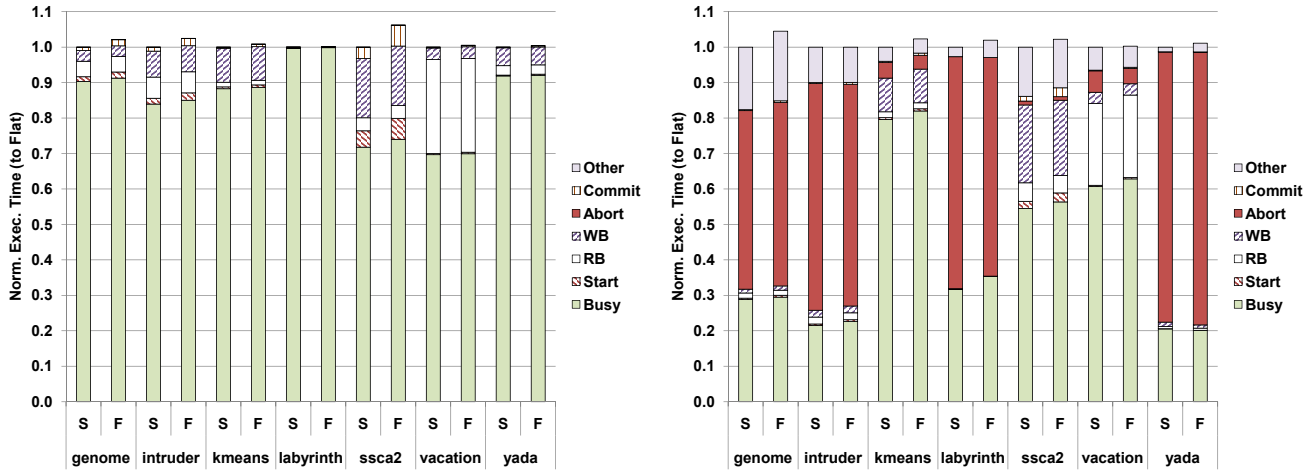
Table 6 shows the performance differences between SigTM and FaNTM using STAMP benchmarks that only use top-level transactions. It summarizes the normalized performance difference (NPD) defined as $NPD(\%) = \frac{T_{FaNTM} - T_{SigTM}}{T_{SigTM}} \times 100$ (the larger NPD, the slower FaNTM).

As shown in Table 6, the average NPD is 2.3% across all the benchmarks and thread counts, which is small. While FaNTM barriers such as TxStart and TxCommit include extra code, their performance impact is insignificant because they are infrequently executed (compared to TxLoad and TxStore). To understand the exact runtime overheads, we show execution time breakdowns in Figures 9(a) and 9(b). The execution time of each application is normalized to the one on SigTM with 1 (Figure 9(a)) and 16 threads (Figure 9(b)). Execution time is broken into "busy" (useful instructions and cache misses), "start" (TxStart overhead), "RB" (read barriers), "WB" (write barriers), "abort" (time spent on aborted transactions), "commit" (TxCommit overhead), and "other" (work imbalance, etc.).

With 1 thread (Figure 9(a)), NPD is relatively high (FaNTM is slower) when small transactions are used and account for a significant portion of the execution time (e.g., intruder and ssca2) [5]. This is because the runtime overhead due to the extra code in TxStart and TxCommit is not fully amortized with small transactions. In contrast, when larger transactions are used (e.g., labyrinth, vacation, and yada), the performance difference becomes small. We observe a similar performance trend with 16 threads (Figure 9(b)) except that several applications spend a significant portion of the time on aborted transactions.

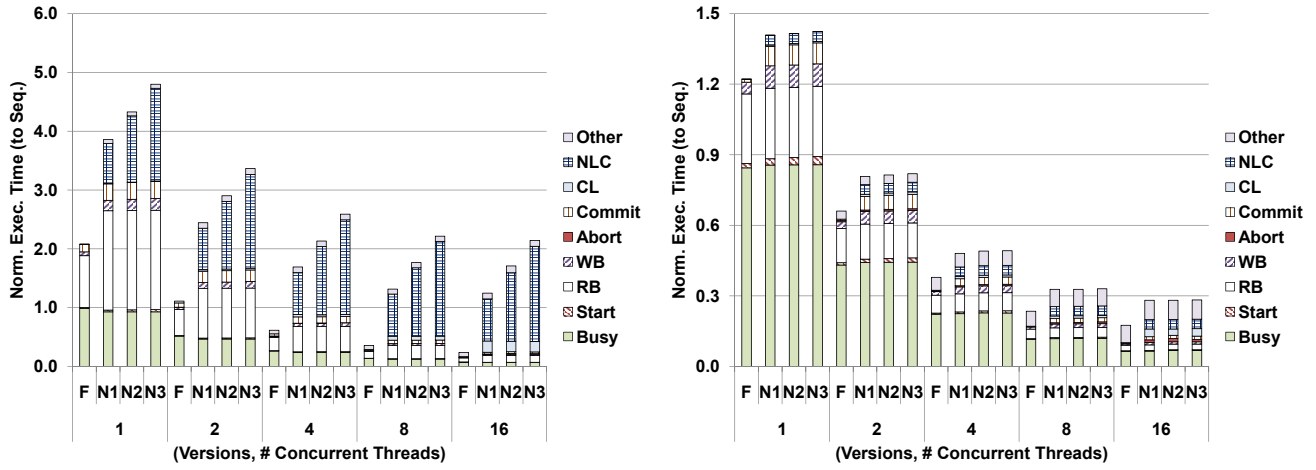
5.3 Q2: Overhead of Deeper Nesting

We quantify the incremental overhead when the available parallelism is used in deeper nesting levels (NLs). We used two microbenchmarks: hashtable and rbtree. They perform concurrent operations to a hash table (hashtable) with 4K buckets and a red-black tree (rbtree). hashtable performs 4K operations where 12.5% are inserts (reads/writes) and 87.5% are look-ups (reads). rbtree performs 4K operations where 6.25% are inserts (reads/writes) and 93.75% are look-ups (reads). Each transaction in hashtable and



(a) 1 thread (b) 16 threads

Figure 9: Execution time breakdowns of STAMP applications. S and F indicate SigTM and FaNTM.



(a) NesTM (software) (b) FaNTM (hybrid)

Figure 10: Execution time breakdowns of hashtable **at various nesting levels.**

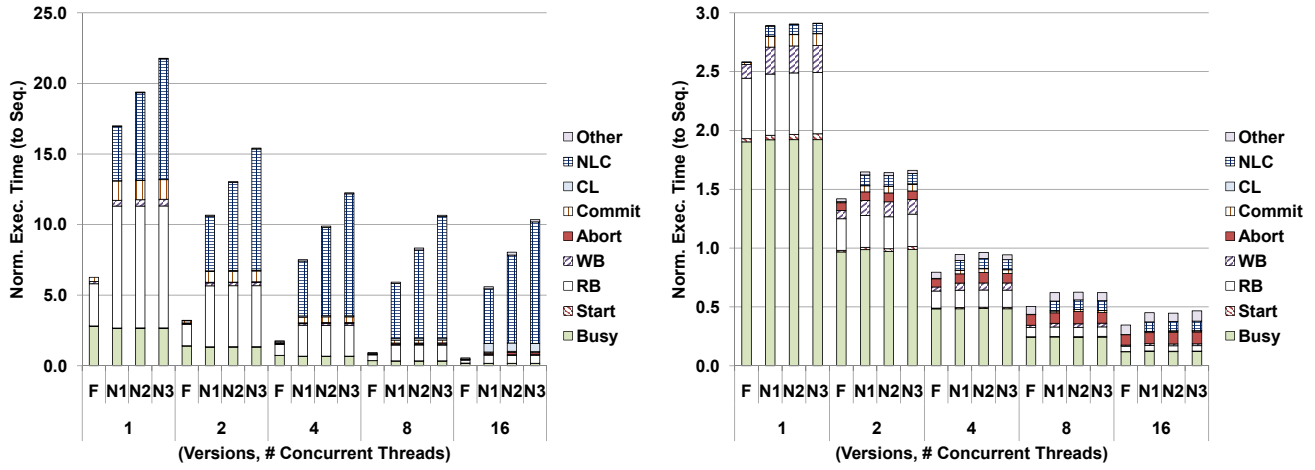
rbtree performs 8 and 4 operations. Each microbenchmark has 4 versions. flat uses only top-level transactions. N1 pushes down the available parallelism to NL=1, by enclosing the same code with a big outer transaction. We implemented N2 and N3 by repeatedly adding more outer transactions. Note that flat and nested versions have different transactional semantics (i.e., perform all the operations atomically or not). We compare them to investigate the runtime overheads of nested transactions.

Figures 10 and 11 provide the execution time breakdowns of hashtable and rbtree normalized to that with the non-transactional, sequential version. Figures 10(a) and 11(a) present the results with NesTM, while Figures 10(b) and 11(b) show the results with FaNTM. Apart from the aforementioned segments, each bar contains additional segments: "CL" (time spent acquiring the commit-locks of parents), and "NLC" (time spent committing non-leaf transactions).

We aim to answer the following three sub-questions: **Q2-1**: What is the incremental performance overhead of nested parallel transactions over top-level transactions? **Q2-2**: How much faster is FaNTM than a nested STM (NesTM) when running nested parallel transactions? **Q2-3**: How much computational workload is required to amortize the overhead of deeper nesting?

Q2-1: Figures 10(b) and 11(b) show that FaNTM continues to scale up to 16 threads. With 16 threads, N1 versions are faster than the sequential version by 3.6× (hashtable) and 2.2× (rbtree). Scalability of rbtree with 16 threads is limited by conflicts among nested transactions. Figures 10(b) and 11(b) also reveal the three FaNTM performance issues. First, the runtime overhead of the write barrier becomes more expensive when running nested transactions. This is due to more cache misses caused when accessing undo-log entries. Since previously-used undo-log entries of a nested transaction are merged to its parent, temporal locality is lost when accessing these entries. However, since writes are relatively infrequent (compared to reads) in both microbenchmarks, the performance impact is not significant.

Second, performance can be degraded due to the contention on the commit-lock of the parent when a large number of nested transactions simultaneously attempt to commit. In hashtable, nested transactions rarely conflict with each other even with 16 threads



(a) NesTM (software)

(b) FaNTM (hybrid)

Figure 11: Execution time breakdowns of rbtree at various nesting levels.

(Figure 10(b)). Therefore, many child transactions can simultaneously commit and cause this commit-lock contention. In contrast, conflicts among nested transactions are relatively frequent in rbtree with 16 threads (Figure 11(b)). Thus, the performance impact of this commit-lock contention is small. Finally, the extra code in TxStart and TxCommit may introduce additional runtime overheads. However, since they are amortized using reasonably large transactions in both microbenchmarks, their performance impact is not critical.

Q2-2: Figures 10(a) and 10(b) (and also Figures 11(a) and 11(b)) demonstrate that FaNTM significantly outperforms NesTM, when executing nested transactions. For example, N1 versions with 16 threads run 4.4× (hashtable) and 12.4× (rbtree) faster on FaNTM than NesTM. This performance improvement is achieved by addressing the two critical performance issues of NesTM.

First, FaNTM eliminates the linearly-increasing runtime overheads of NesTM such as repeated read-set validation. Since NesTM repeatedly validates the same memory objects in the read-set across different nesting levels, it suffers from excessive runtime overheads that linearly increase with the nesting depth. On the other hand, since hardware filters continuously provide nesting-aware conflict detection, FaNTM does not suffer from this performance pathology.

Second, the performance of the FaNTM read barrier is almost unaffected when running nested transactions, whereas the performance of the NesTM read barrier is drastically degraded. This is mainly due to more cache misses when accessing read-set entries in NesTM. Since committing transactions merge their read-set entries to their parent, NesTM cannot exploit temporal locality when accessing these entries. In contrast, since software read-sets are replaced with hardware signatures, FaNTM does not suffer from this performance pathology.

Q2-3: We investigate how much computational workload is required to amortize the runtime overheads of nested transactions on FaNTM and NesTM. To this end, we compare the performance of the nested versions of hashtable with the flat version by varying the size of computational workload within transactions. Figure 12 presents the normalized performance difference (NPD) with 16 threads. With little work, NPD is very high on NesTM because the extra overheads of nested transactions cannot be amortized. In contrast, even with little work, the nested versions on FaNTM perform comparably with the flat version because FaNTM introduces reasonable runtime overheads to nested transactions. Furthermore, as the nesting depth increases, NesTM requires even larger workloads to amortize the linearly-increasing overheads. On the other hand, the performance of nested transactions on FaNTM is almost unaffected by nesting depth, requiring no extra workload for deeper nesting.

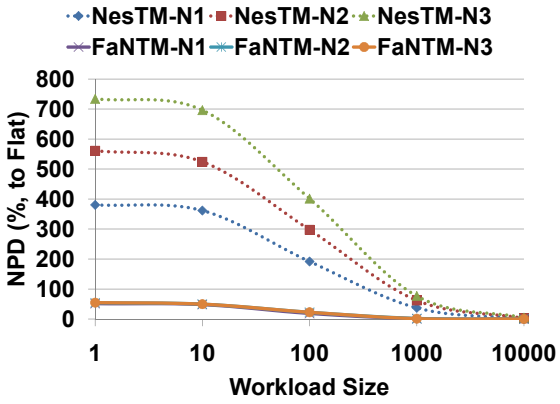
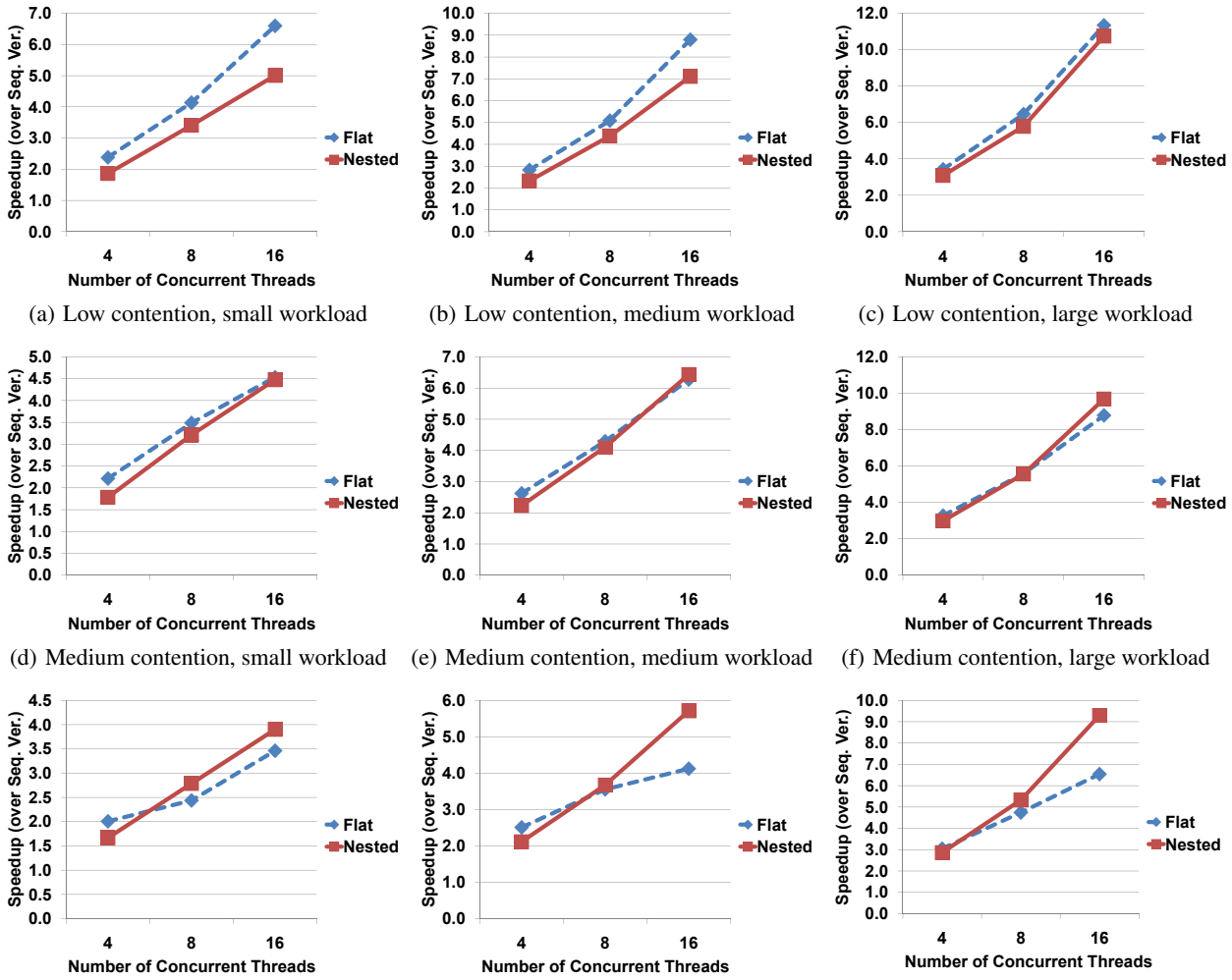


Figure 12: Performance sensitivity to workload size.

5.4 Q3: Improving Performance using Nested Parallelism

np-rbtree operates on a data structure that consists of multiple red-black trees. It performs two types of operations on the data structure: *look-up* operations that look up (read) entries and *insert* operations that insert (read/write) entries in the red-black trees. Insert operations often modify the trees in a global manner, causing many other transactions to abort. The ratio of look-up to insert operations is configurable. Each operation atomically accesses all the trees in the data structure within a transaction. After accessing each tree, np-rbtree executes computational workload whose size is also configurable.

We exploit the parallelism in np-rbtree in two ways – (1) flat: parallelism at the outer level (i.e., inter-operation) and (2) nested:



(a) Low contention, small workload (b) Low contention, medium workload (c) Low contention, large workload

(d) Medium contention, small workload (e) Medium contention, medium workload (f) Medium contention, large workload

(g) High contention, small workload (h) High contention, medium workload (i) High contention, large workload

Figure 13: Scalability of np-rbtree**.**

parallelism at both levels (i.e., inter-operation and inter-tree). If the percentage of the insert operations is high, the scalability of the flat version can be limited due to the frequent conflicts among top-level transactions.

We performed experiments by varying the two configurable parameters: the degree of contention and the size of computational workload. Low, medium, and high contention cases are the ones where 1%, 5%, and 10% of the operations are inserts. Small, medium, and large workload cases are the ones where the computational workload iterates over 32, 128, and 512 loop iterations, respectively. In addition, np-rbtree performs 1024 operations, each accessing 8 red-black trees atomically.

Figure 13 demonstrates that the flat version significantly outperforms the nested version with low contention and small workload. This performance difference results from the sufficient top-level parallelism (i.e., low contention) effectively exploited by the flat version and the unamortized overheads (i.e., small workload) of the nested version such as runtime overheads of nested transactions. In contrast, the nested version greatly outperforms the flat version with high contention and large workload. This is because the nested version can effectively exploit the parallelism available at both levels and its overheads are sufficiently amortized using large workload. On the other hand, the scalability of the flat version is mainly limited due to the frequent conflicts among top-level transactions.

6. RELATED WORK

Moss and Hosking proposed the reference model for concurrent nesting in TM [14]. Based on the proposed model, a few recent papers investigated nested parallelism in STM [2–4, 15, 21]. While compatible with existing multicore chips, this software-only approach may introduce excessive runtime overheads due to the use of complicated data structures [2, 4] or the use of an algorithm whose time complexity is proportional to the nesting depth [3], limiting its practicality. Our work differs because FaNTM aims to eliminate substantial overheads of software nested transactions using hardware acceleration.

Vachharajani proposed an HTM that supports nested parallelism within transactions [20]. While insightful, the proposed design drastically increases hardware complexity by intrusively modifying hardware caches to implement nesting-aware conflict detection and data versioning. In contrast, FaNTM simplifies hardware by decoupling nesting-aware transactional functionality from caches using hardware filters.

7. CONCLUSION

This paper presented FaNTM, a hybrid TM that provides practical support for nested parallel transactions using hardware filters. FaNTM effectively eliminates excessive runtime overheads of software nested transactions using lightweight hardware support. FaNTM simplifies hardware by decoupling nested parallel transactions from hardware caches. Through our performance evaluation, we showed that FaNTM incurs a small runtime overhead when only single-level parallelism is used. We also demonstrated that nested transactions on FaNTM perform comparably with top-level transactions and run significantly faster than those on a nested STM. Finally, we showed how nested parallelism can improve the performance of a transactional microbenchmark.

Acknowledgements

We would like to thank Daniel Sanchez, 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 Award 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 nested parallel transactions in software transactional memory. In *22nd ACM Symposium on Parallelism in Algorithms and Architectures*. June 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] 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.

[6] 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.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, October 2006.

[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] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, June 2004.

[10] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[11] 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.

[12] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Synthesis Series, 2006.

[13] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *12th International Conference on High-Performance Computer Architecture*, February 2006.

[14] 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.

[15] 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.

[16] B. Saha, A. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO '06: Proceedings of the International Symposium on Microarchitecture*, 2006.

[17] 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.

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

[19] 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.

[20] N. A. Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Princeton University, 2008.

[21] 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.