Technical Report RT/01/2013

Practical and Lock-free Parallel Nesting for Software Transactional Memory

Nuno Diegues
INESC-ID/IST
nmld@ist.utl.pt

João Cachopo
INESC-ID/IST
joao.cachopo@ist.utl.pt

January 2013
Abstract

Transactional Memory (TM) provides a strong abstraction to tackle the challenge of synchronizing concurrent tasks that access shared state. Yet, at the same time, TM inhibits the programmer from fully exploring the latent parallelism in his application. In particular, it does not allow a transaction to contain parallel code. This fact limits the expressiveness of TM as a synchronization mechanism.

Many applications contain large operations that must be performed atomically. Such large sections have an increased likelihood of creating conflicts with concurrent transactions. Yet, in some cases these sections could be executed faster if their latent parallelism was used efficiently, by allowing a transaction to be split in several parts that execute concurrently.

It is possible to achieve that with parallel nesting. In this paper we provide practical and efficient support for parallel nesting. We present the first lock-free parallel nesting algorithm with support for multi-versions, which we used to outperform the original STM on several known benchmarks by up to 2.8 times. We also show improvements over state-of-the-art parallel nesting alternatives of up to 3.6 times.

**Keywords:** JVSTM, Nested Parallel Transactions, Conflicts
1. Introduction

Transactional Memory (TM), originally proposed in [10], promises to tackle one of the major challenges in the development of a concurrent program: How to synchronize concurrent tasks that access shared mutable state. The other major challenge—how to split a program into concurrent tasks—is not addressed by TM, but is largely influenced by it.

In fact, we claim that most TM systems hinder the use of fine-grained parallelism, because they treat each transaction as a sequential set of instructions to be executed. Thus, large transactions limit the amount of parallelism that can be explored in applications, because those large transactions cannot be split into parallel subtasks.

To illustrate the problem, consider a graph of transactional objects as represented in Fig. 1: There are root objects that provide access to one or more other objects, such that an object accessible from a root may also be accessible from another root. This arbitrary structure is representative of the state of transactional programs.

In Listing 1, we show a simple program that uses such a graph of objects. There is a method (updateGraph) that traverses all the objects in the graph and updates some of them, and another method (changeStatus) that traverses only a few objects and modifies at most one of them. Both methods should execute atomically. This means that the updateGraph method is very likely to cause a conflict with any other transaction manipulating the graph (such as changeStatus). In some sense, this kind of workloads challenges the optimistic concurrency model used by most TMs: As updateGraph calls continuously conflict and restart, they will accumulate and get delayed; any attempt to enforce their successful execution will severely hinder concurrency and the throughput of the application.

But this is no longer true if we are able to parallelize the contending transaction, as shown in Listing 2, where we explore the paths of each root in parallel. To be able to do this, however, the TM system must have support for parallel nesting. Yet, parallel nesting has deserved very little attention from TM implementors. Work in Hardware TMs (HTM) has pushed towards minimalistic changes in the hardware, to avoid making the verification of the correctness of the components more difficult. In particular, nesting has been identified as an extension that, despite its usefulness, is most likely to be used in software implementations, to which the HTM falls back [12]. But still, almost none of the Software TM (STM) implementations supports parallel nesting.

In this paper, we seek to fill this gap, by presenting a practical, lightweight algorithm for parallel nesting that we implemented on top of JVSTM [7]. A key requirement of a practical parallel nesting
Listing 1. Program that manipulates the graph represented in Fig. 1 and that is synchronized with TM.

implementation is that its overhead does not annihilate the gains of exploring finer-grained parallelism. We show that our implementation of parallel nesting allows us to get significant performance gains in a variety of highly-conflicting workloads, and that it has low overhead when no new parallelism is being exploited.

Moreover, the algorithm that we present is the first parallel nesting algorithm with support for multi-versions, preserves the lock-freedom progress guarantee of the underlying STM, and does not affect the normal execution of the underlying STM when no parallel nesting is used.

The remainder of this paper is organized as follows. In Section 2, we provide an overview of the underlying STM that we extended. Next, in Section 3, we introduce parallel nesting and provide our algorithm in Section 4. We evaluate it in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

Figure 1. Graph of transactional objects in an application.
Listing 2. Parallelizing the updateGraph method.

2. JVSTM Overview

The Java Versioned STM (JVSTM) [7] is a word-based, multi-version STM that was specifically designed to optimize the execution of read-only transactions: In the JVSTM, read-only transactions have very low overheads and never contend against any other transaction. In fact, once started, the completion of read-only transactions is wait-free in the JVSTM.

To achieve this result, JVSTM uses Versioned Boxes (VBoxes) to represent transactional locations. Each VBox holds a history of values for the transactional location, by maintaining a list of bodies (VBoxBody), each with a version of the data. The access to VBoxes is always mediated by a transaction, which is created for that sole access if none is active at that moment.

A read-only transaction always commits successfully in the JVSTM because it reads values in a version that corresponds to the most recent version that existed when the transaction began. Thus, all reads are consistent and read-only transactions may be serialized at the instant they begin, i.e., it is as if they had atomically executed at that instant. Read-write transactions, however, must be serialized when they commit. Therefore, they are validated at commit-time to ensure that values read during their execution are still consistent with the current commit-time, i.e., that values have not been changed in the meantime by another concurrent transaction.

Transactions mediate all accesses to VBoxes because they need to record each transactional access in their local logs: Reads and writes are logged in the transaction’s read-set and write-set, respectively. Both logs are used at commit time: If the read-set is still valid, the tentative writes logged in the write-set are written back, producing a new version for each of the boxes written and effectively publicizing the new values in a lock-free manner [7].

The original design of the JVSTM follows a linear nesting model, in which a thread that is executing a transaction may start, execute, and commit a nested transaction (which itself may do the same), effectively forming a nesting tree with only one active branch at a time. The leaf of that active branch represents an active nested transaction that is guaranteed to be the only one accessing and modifying the read- and write-sets of that nesting tree. This simplifies the algorithm and makes it very easy for the JVSTM to deal with the nesting model. Yet, this simple model does not allow the decomposition of long transactions into concurrent parts, which we now provide as one of the major contributions of this work.

3. The Parallel Nesting Model

In this section, we briefly introduce the model of parallel nesting that we use, which is based on the model of closed nesting described by Moss [15], and we introduce some terminology.

To describe an execution involving one or more transactions, we shall use, throughout this paper, the following notation for the operations occurring within the transactions, where we assume that each
operation executes atomically:

- \( W_t(x, k) \) means that transaction \( t \) writes the value \( k \) to the transactional variable \( x \).
- \( R_t(x, k) \) means that transaction \( t \) reads the transactional variable \( x \) and finds the value \( k \).
- \( C_t(res) \) means that transaction \( t \) attempted to commit and either succeeded (when \( res = \text{ok} \)) or failed (when \( res = \text{fail} \)).
- \( S_t(t_1, t_2, ..., t_n) \) means that transaction \( t \) spawns the parallel nested transactions \( t_1, t_2, ..., t_n \).

Two nested transactions are said to be siblings if they have the same parent. In parallel nesting, siblings may run concurrently. In our model, each top-level transaction may unfold a nesting tree in which a transaction performs transactional accesses only when all its children are no longer active.

Conceptually, in this model, a nested transaction maintains its own read-set and write-set, much in the same way of a top-level transaction. Yet, given the compositional nature of transactions, reading a transactional location within a nested transaction must always access the value that was most recently written to that location in the sequence of operations performed by the transaction reading and by all of its ancestors (but not by any of its siblings).

The set of ancestors of a transaction with parent \( T \) is composed by adding \( T \) to the set of ancestors of \( T \). A top-level transaction has no parent, and, therefore, its ancestor set is empty.

When \( T \) performs a read operation in this model, it may not suffice to access the write-set of the transaction: If it does not find any write there, it still has to access the ancestors’ write-sets. In that case, the read operation may retrieve a globally uncommitted value that resulted from the execution of a sibling or ancestor of \( T \). Therefore, siblings must synchronize and validate their commits such that they respect the correctness criterion (here assumed to be opacity).

4. A new design for parallel nesting

The algorithm that we propose in this work has three major features that make it efficient in practice: (1) a fast mode for writing, backed up by a slow mode for fallbacks; (2) a fast path in the read operation that performs in constant time; and (3) a commit operation that is independent of the write-set size.

In the next section we describe the key ideas of our design and present the rationale for our choices. Then, in Section 4.2, we describe the data structures that are essential to understand the algorithms of the various transactional operations, which are detailed in the following sections.

4.1. High-level overview of the algorithm

Recall that a VBox, representing a transactional location, points to a history of all the versions for that location that may still be readable by some active transaction. The parallel nesting algorithm that we propose in this paper extends VBoxes so that transactions may now write directly to the VBoxes rather than having to maintain some private mapping of each location written to its new value. But we need to distinguish between globally committed values and the tentative values of ongoing transactions. Thus, a VBox now contains both permanent and tentative versions, as shown in Fig. 2.

\[ \text{This restriction simplifies the model and does not impose any significant limitation to the expressive power of the parallel nesting model, because a transaction that needs to execute concurrently with its children may spawn a nested transaction to execute its own code.} \]
When a VBox \( x \) is created, it points to a dummy tentative value, and the algorithms that we present ensure that it always points to that dummy value or some write performed by a transaction. For a nested transaction \( T \) to write to \( x \), \( T \) must acquire ownership over the tentative slot of \( x \). In that case, it is able to store the value in the first item of the tentative list. After this, new tentative values may be enqueued in \( x \) by transactions that are descendant of the owner (or the owner itself) of the first tentative in \( x \). When the root transaction \( A \) (of the nesting tree of \( T \)) commits, it enqueues a new permanent item in \( x \) corresponding to the first tentative item in \( x \), which is necessarily owned by \( A \) at that point. So, the idea is that a permanent value has been consolidated via a commit of some top-level transaction, whereas a tentative value belongs to an active top-level transaction (or any of its children nested transactions) and is thus part of its write-set.

An immediate consequence of this design is that only one nesting tree may resort to each in-place tentative location at a given time. So, when \( T \) is able to acquire the ownership of the tentative location of a VBox, it performs the write in the faster mode. But if that location is already being tentatively used by another transaction, then \( T \) must use another, slower mode for writing—a fallback mode. This fallback mode, described in detail in Section 4.5., uses a write-set shared by the entire nesting tree. This write-set is modified only by \( A \), the root ancestor of the nesting tree. Thus, when \( T \) needs to resort to this fallback mode, \( T \) and all of its ancestors up to \( A \) (excluding) are aborted and re-executed by the thread executing \( A \), effectively flattening that nesting branch. Recall that in this parallel nesting model \( A \) proceeds only when its children have finished. This means that this re-execution faces no concurrency within the nesting tree, because \( A \) will re-execute those forcefully aborted children only after all other children are finished.

So, as long as transactions do not contend for writing to the same VBox, transactions will be able to write in the faster mode. In practice, we expect this to be the common case. Table 1 presents the total number of writes and, of those, the total number of write-after-read operations that were collected from several benchmarks. These results show that, typically, a transaction writes to transactional variables that it read before. Therefore, if two transactions write to the same variable, they will most likely have already read it. If that is the case, they will cause a read-write conflict and only one will be able to commit. Because conflicts should be rare, or otherwise the parallelization is not effective, we claim that successful transactions will most of the time have written in the tentative slots.

On the other hand, the read operation must account for the possibility of read-after-writes, and thus may become expensive due to lookups in ancestors’ write-sets. With this design, we are also able to provide a very cheap read operation. Each tentative write points to an orecc object that contains the owner (a transaction) and the version of the write. So, whenever \( T \) reads \( x \), it is able to tell if the tentative value of \( x \) is from an older committed transaction simply by looking into the orecc of that tentative value. If that is the case, then neither \( T \) nor its ancestors wrote to \( x \), and, thus, our algorithm skips the read-after-write check and reads a permanent value in constant time (namely, independently of the nesting depth).
Finally, because each tentative write points to its owner indirectly through the oreo object, another advantage of this design is that the writes controlled by $T$ are all affected by changing $T$’s oreos. In fact, our nested commit algorithm makes the writes of $T$ visible in its parent by merely changing $T$’s oreos. The number of oreos of $T$ are typically negligible in comparison to the number of writes controlled by $T$, which means that this commit procedure becomes much faster than a typical lazy write-back commit.

4.2. Data structures and auxiliary functions

As seen in Fig. 2, the metadata associated with an in-place write is available in an oreo object (short for Ownership Record [18]), which represents the transaction that owns the write. An oreo contains the following information:

- **owner**: The transaction that is being represented by this oreo.
- **status**: Identifies whether the transaction is still running, has aborted, or has committed. It always starts as running, and then either changes to aborted or to committed in the commit of a top-level transaction. The committed status is represented by the timestamp acquired by incrementing the global clock during the top-level commit of the owner of this oreo.
- **nestedVer**: Version of the writes that this oreo owns. This version only makes sense when paired with the owner. It is used to ensure that reads on tentative write-entries respect the correctness criterion.

To ensure consistent reads and to preserve multiple versions, each transaction $T_i$ maintains an **nClock** (short for nesting clock) that is incremented by the commit of each children of $T_i$. A nested transaction also contains an **ancVer** (short for ancestors versions) map: When it starts, it computes the map by adding the parent’s current nClock to the parent’s ancVer to obtain its own map. $T_i$’s ancVer may contain a mapping such as: Transaction $T_k$ to value $n$, meaning that $T_i$ may read values that are owned by $T_k$ with a nestedVer that is at most $n$ but not larger. Fig. 3 shows the state of the nesting tree resultant of the following execution:

<table>
<thead>
<tr>
<th>benchmark-workload</th>
<th>writes ($\times 10^3$)</th>
<th>wars ($\times 10^3$)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>bench7-rr-notrav</td>
<td>6158</td>
<td>6158</td>
<td>100</td>
</tr>
<tr>
<td>bench7-rrw-notrav</td>
<td>6646</td>
<td>6646</td>
<td>100</td>
</tr>
<tr>
<td>bench7-w-notrav</td>
<td>3967</td>
<td>3967</td>
<td>100</td>
</tr>
<tr>
<td>bench7-r</td>
<td>46267</td>
<td>46267</td>
<td>100</td>
</tr>
<tr>
<td>bench7-rrw</td>
<td>65107</td>
<td>65107</td>
<td>100</td>
</tr>
<tr>
<td>bench7-w</td>
<td>74744</td>
<td>74744</td>
<td>100</td>
</tr>
<tr>
<td>lee-mainboard</td>
<td>160</td>
<td>160</td>
<td>100</td>
</tr>
<tr>
<td>lee-memboard</td>
<td>148</td>
<td>148</td>
<td>100</td>
</tr>
<tr>
<td>lee-sparselong</td>
<td>16</td>
<td>16</td>
<td>100</td>
</tr>
<tr>
<td>lee-sparseshort</td>
<td>8</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td>vac-reservations</td>
<td>0.3</td>
<td>0.3</td>
<td>100</td>
</tr>
<tr>
<td>vac-deletions</td>
<td>1684</td>
<td>1526</td>
<td>91</td>
</tr>
</tbody>
</table>

Table 1. Total number of writes and write-after-read (wars) performed in various workloads of the STMBench7, the Lee-TM, and the Vacation benchmarks.
In this example the ancVer of $E$ allows it to read the writes controlled by either $A$ or $D$, but not by $C$. Moreover, the information available in the ancVer also restricts the versions of the tentative writes that $E$ may read, as we shall see in Section 4.3. Note that $A$’s nClock was incremented by $B$’s commit. $C$ and $D$ were spawned after the commit of $B$ and thus obtain the incremented clock from $A$ to create their ancVer maps.

A nested transaction also holds the following information:

- **startVersion**: Timestamp obtained by a top-level transaction when it starts. A nested transaction uses the timestamp of its root ancestor to ensure that the transaction always sees a consistent view of the permanent versions in VBoxes.
- **globalReads**: Read-set that maintains the VBoxes from which permanent values were read by the transaction.
- **nestedReadSet**: Read-set used for reads that obtain values from tentative writes in the same nesting tree. For each VBox read in this manner, an in-place tentative write read is registered.
- **rootWriteSet**: Write-set that maintains VBoxes and the values written to them by a top-level transaction. This structure is allocated by a root transaction, and the nested transactions in the same nesting tree only hold a pointer to it. This structure is used in the fallback mechanism when the transaction cannot write in-place to some VBox.
- **boxesWritten**: Set of VBoxes that were written in-place by the transaction. This is used in the commit of the root top-level transaction, which is responsible for the write-back that consolidates the tentative versions in the VBoxes.
- **orec**: The ownership record that this transaction uses when it gains control over VBoxes to write to their tentative slot.
- **committedChildren**: The transactions that have committed into this transaction (and thus are its children).

Given this description of the data structures used by parallel nested transactions, we may now describe in detail, in the following sections, the transactional operations performed by a nested transaction. In the description of the those algorithms, we resort to the following auxiliary functions:

![Diagram](image.png)

Figure 3. Part of the state maintained in a nesting tree resulting of Execution (1).
• abortUpTo(tx, conflicter) function that causes the execution of the algorithm to stop and leads to the abort of tx as well as its ancestors up to conflicter (including).

• lowestCommonAnc(tx, other) returns the deepest transaction in the nesting tree that is ancestor of both tx and other.

• sameNestingTree(tx, other) returns true if tx and other are transactions of the same nesting tree.

• executeAsTopLevel(tx) has the same effect as calling abortUpTo(tx, TOP) (meaning that it aborts up to the top-level, excluding the root). In addition to that, the code encapsulated in tx and its ancestors (which were forcefully aborted) is re-executed in the scope of the root ancestor of tx. This re-execution takes place once all the currently executing nested transactions in this nesting tree have finished, before the root ancestor resumes its own execution.

4.3. Reading a VBox

When reading a VBox, we have to take into account a possible read-after-write. For this reason, the pseudo-code presented in Algorithm 1 either returns a tentative value written by some transaction in this nesting tree, or a global value, represented by a consolidated version in a VBox.

In lines 6-10 the algorithm tests the aforementioned fast path that allows it to decide if there is any chance that a write may have been done to the VBox in this nesting tree. If it finds that to be impossible, then a globally consolidated value is returned, which is typically the common case.

If the fast path is not used, then the algorithm checks if the VBox is being controlled by the nesting tree (line 12). In that case, it iterates over the writes in the tentative list of the VBox until one of the following conditions is verified:

1. The owner of the write entry being iterated is the transaction attempting the read, in which case no further verification is needed to read it (lines 17-19).

2. The owner of the write entry being iterated is an ancestor of the reader T. When this happens, T may read that entry only if it was made visible in its current owner before T started (lines 20-26). This is enforced by looking up in the ancVer of T what is the maximum version readable from that ancestor. If the write has a more recent version than T can read, T causes a chain abort up to that ancestor such that those nested transactions restart with the most recent versions on their ancVer map.

The algorithm presented here stops when a nested transaction finds a tentative write that it may read, returning that value. This is correct because we ensure the following invariant: Given a read operation of a VBox, as soon as the nested transaction reaches a tentative write that may be read, then it is guaranteed that no other write further down the list has to be read instead of that one to maintain correctness. By ensuring this invariant, we are able to shorten traversals as most transactions will read only the tentative write at the head of the list.

4.4. Writing to a VBox

Algorithm 2 presents the pseudocode of the write operation of a parallel nested transaction. When accessing the VBox, it fetches the tentative write at the head (lines 2-3). By reading the orecc in the first tentative write entry, it is able to tell whether that VBox is currently owned by the transaction,
Algorithm 1 Read procedure in a parallel nested transaction.

1: GetBoxValue(tx,vbox):
2: wInplace ← vbox.tentative
3: status ← wInplace.orec.status
4: // check if this is not a read-after-write
5: // a positive integer in status means that the status is COMMIT
6: if status > 0 ∧ status ≤ tx.startVersion then
7: value ← readFromPermanent(vbox)
8: tx.globalReads.add(vbox)
9: return value
10: end if
11: // it is possible to be a read-after-write
12: if sameNestingTree(tx, wInplace.orec.owner) then
13: while wInplace ≠ null do
14: orec ← wInplace.orec
15: nVer ← orec.nestedVer
16: owner ← orec.owner
17: if owner = tx then
18: return wInplace.value
19: end if
20: if tx.ancVer.contains(owner) then
21: if nVer > tx.ancVer.get(owner) then
22: abortUpTo(tx, owner)
23: end if
24: tx.nestedReadSet.put(vbox,wInplace)
25: return wInplace.value
26: end if
27: wInplace ← wInplace.previous
28: end while
29: end if
30: // no in-place write may be read, check fallback set
31: value ← rootWriteSet.get(vbox)
32: if value ≠ NONE then
33: // do not need to register the read in this case
34: return value
35: end if
36: value ← readFromPermanent(vbox)
37: tx.globalReads.add(vbox)
38: return value

in which case it simply overwrites the previous write. Otherwise, after line 9, the algorithm follows one of these cases:

- The VBox owner has finished before this transaction started, in which case this transaction attempts to acquire ownership of the tentative write at the head of the list of that VBox (lines 9-17). To do so, it attempts a compare-and-swap (CAS) to change the ownership of the first tentative write. If the CAS fails, the algorithm proceeds to the fallback in line 26 (because some other transaction acquired the ownership of the box). If, on the other hand, the previous owner
Algorithm 2 Write procedure in a parallel nested transaction.

1: SetBoxValue(tx,vbox,value):
2: wInplace ← vbox.tentative
3: orecc ← wInplace.orec
4: // check for a write-after-write
5: if orecc.owner = tx then
6: wInplace.value ← value
7: return
8: end if
9: if orecc.status ≠ ALIVE then
10: if orecc.status ≤ tx.startVersion then
11: // attempt to acquire ownership
12: if wInplace.CASorec(orecc, tx.orec) then
13: wInplace.value ← value
14: tx.boxesWritten.add(vbox)
15: return
16: end if
17: end if
18: else if tx.ancVer.contains(orecc.owner) then
19: // belongs to an ancestor
20: newW ← (value, tx.orec, wInplace)
21: if vbox.CAStentative(wInplace, newW) then
22: return
23: end if
24: end if
25: // cannot write in-place, fallback mechanism
26: executeAsTopLevel(tx)

finished after this transaction started, then no transaction in this nesting tree (and particularly the one attempting the write) will ever be able to write to that VBox in-place. In that case the algorithm also proceeds to the fallback mechanism. This restriction is what allows us to maintain the fast path presented in line 6 of the read operation in Algorithm 1.

- The VBox is owned by an ancestor of this transaction (line 18). This is true if the current owner is present in the ancVer of this nested transaction. In this case, the transaction attempts to enqueue a new in-place tentative write by performing a CAS on the head of the list of that VBox. If this CAS fails, then some other transaction in this nesting tree succeeded, in which case the algorithm proceeds to the fallback.

- We are left with the fallback mechanism in line 26. To get here, there must be a concurrent transaction controlling the VBox. Therefore, the idea is that the fallback writes in an alternative way that we describe in Section 4.5.

Much of the complexity of this operation arises from the fact that we are maintaining the previous writes in the nesting tree. To understand why we need to keep the multiple versions, consider the following scenario:

\[
\begin{align*}
&W_A(x, 1); W_A(y, 42); S_A(B, C); R_B(x, 1); \\
&W_B(y, 0); W_C(x, 2); C_C(ok); C_B(fail)
\end{align*}
\]
Assume that the write performed by nested transaction $B$ overwrote the write performed by its parent $A$ in $\text{VBox} \ y$. Given that $B$ aborts due to a read-write conflict with $C$, then, in the end, the write performed by $A$ to $y$ is missing as it was speculatively overwritten by $B$, which failed to commit.

The algorithm that we provide detects write-write conflicts. To understand this choice, consider the following execution:

$S_A(B, C); W_B(x, 5); W_C(x, 10); C_C(ok); C_B(ok)$

At the end of this execution, the first write entry of the tentative list of $\text{VBox} \ x$ would be the one written by $C$, because $C$ performed the most recent write. Yet, the most recent commit was $B$’s, which consequently should make its writes the most recently publicized to the parent $A$. Therefore, the underlying problem is one of lag between the write-time and the commit-time. To maintain the invariant mentioned in Section 4.3, for faster reads, the solution is to perform work proportional to the size of the write-set at commit-time [6], which we avoid in this design by precluding nested write-write concurrency. Moreover, we claim that this design decision should not preclude much concurrency in real workloads: As pointed out earlier in Table 1, we do not expect write-write concurrency to be very frequent without yielding read-write conflicts that would preclude that concurrency in the first place.

4.5. Fallback mechanism

In the event of write-write concurrency in the same $\text{VBox}$, later writes use a fallback mechanism, allowing their transactions to proceed in an alternative way to writing in-place. At this point it is important to explain briefly how top-level transactions address this same problem. Their writing procedure is a lightweight version of what was presented for parallel nested transactions: A top-level transaction $T_i$ attempts to control a $\text{VBox} \ x$ (in case it never wrote to it), if the owner of $x$ finished before $T_i$ started; otherwise, it resorts to its private write-set that maintains a mapping from transactional locations to their tentative values in the transaction. Being able to write in-place in the $\text{VBox}$ is cheaper than maintaining the mentioned mapping, for which reason this algorithm also yields benefits for the top-level transactions.

Therefore, when $T$ (a parallel nested transaction) fails to perform a read from the tentative writes in a $\text{VBox}$, it still needs to check the write-set of the root transaction of its nesting tree (lines 31-35) in Algorithm 1. It may have happened that the root ancestor, prior to the execution of its children, attempted to write to that $\text{VBox}$ without success, having thus had to fallback to the private write-set. If $T$ finds such a write, it need not bookkeep that read because no concurrent transaction in its nesting tree will be able to overwrite that value. If no write is found, $T$ knows it is not a read-after-write and fetches a globally consolidated value (lines 36-38).

On the other hand, when the write procedure determines that the $\text{VBox}$ is currently controlled by another transaction, the algorithm proceeds to line 26 of Algorithm 2. In that event, it aborts the nested transaction up to the root ancestor, and the affected nested transactions in the chain are re-executed by the root (top-level) transaction, once all its children are finished. That re-execution flattens the transactions such that the code being re-executed is encapsulated in the top-level transaction of the nesting tree. In particular, the specific write that triggered this fallback, if repeated, will necessarily be performed by the top-level transaction in the fallback write-set.

4.6. Commit and abort of a parallel nested transaction

At commit-time we are left with two tasks: To ensure that all the reads are still up-to-date, and to make the read-set and the write-set of the transaction visible to its parent. In the following we explain how each task is performed. Then, in Section 4.7, we describe how the commit procedure is
performed in a lock-free manner.

The validation of a transaction $T_i$ is performed by accessing directly the VBoxes that were read and looking at their tentative writes. If $T_i$ read a global value from VBox $x$, it was because there was no tentative write from its nesting tree at the time of the read. Then, the validation of $T_i$ at commit time fails if there exists a tentative write in $x$ that $T_i$ is able to read (unless it is owned by $T_i$). This means that a read-write conflict was detected, which causes $T_i$ to abort. Similarly, this verification is extensible for reads that obtain values tentatively written in the nesting tree. In this case, the verification stops when it finds the tentative write that was read, which means that there is no newer value for the VBox. Once again, this is correct only because we ensure the invariant in the order of tentative writes that was described earlier.

Regarding the publication of the write-set in the parent, we sought to make it as lightweight as possible. Note that every write of a nested transaction is performed in-place. This means that each of the VBoxes written has a tentative write pointing to the oreo of the nested transaction that currently owns the write. To publicize the write-set, a transaction $T$ (that is committing) changes the nestedVer of its oreo to have the timestamp obtained from the parent’s clock for that commit. Finally, it suffices to change the owner in $T$’s oreo to point to the parent of $T$ and to propagate that oreo to the parent. Because commits take place at different nesting depths, in practice, $T$ has a list of its committed children (the committedChildren field). This way, $T$ can access every oreo propagated to it, in addition to its own, so that it updates and propagates them recursively to its parent when it commits.

The reads must also be made accessible to the parent. To do this, we use the committedChildren field, because it allows accessing the reads performed by those children. Consequently, the read-set of a transaction ends up being scattered across the nesting tree. The slight overhead of having a one-time level of indirection for accessing the read-sets is far overcome by the benefit of not having to perform additional work proportional to the size of the read-set at each commit to propagate it to the parent.

In the event of an abort, the transaction simply changes the status field of the oreos that it controls (through the committedChildren field) to an ABORTED value. After this, the write entries at the head of the VBoxes controlled by the transaction are effectively ignored, because concurrent transactions see that their owner is not alive. There is, however, a caveat. Consider the following execution:

$$W_A(x, 1); S_A(B); W_B(x, 2); \ldots; C_B(fail)$$

After the abort of $B$, if some concurrent transaction attempts to write to $x$, it will find that an aborted transaction is controlling it. Therefore, it will be able to acquire control over $x$. However, $A$ was still supposed to control $x$, except that its child $B$ wrote to it and then released the control, due to its abort. Thus, when a nested transaction aborts, it must delete its tentative writes, but only in the case it acquired ownership over those VBoxes from an ancestor.

In practice, the abort operation iterates over the VBoxes written and checks, for each VBox, if there is a write $w$ that is not owned by the aborting transaction and that is owned by a transaction that is alive. If that is the case, $w$’s value and oreo are used to replace the first write in the tentative list. Finally, the aborting transaction changes the status of its oreos to ABORTED.

4.7. Lock-free commit

So far we have omitted how we actually synchronize the commit of sibling parallel nested transactions to their parent. This is important because the two steps of the commit procedure (validation and publication of footprint in the parent) must happen atomically.
We achieved this by adapting the top-level commit algorithm of JVSTM [7] to nested commits: Each transaction contains a queue of NestedCommitRecords that provides a committing order to concurrent children. Each committing nested transaction \( T \) reads the head of the queue. Next, it performs the validation described above and proceeds to obtain its order in the queue by using a CAS to add a new entry on it. The CAS fails if the head of the queue is no longer the same that was read before the validation. This means that some other sibling of \( T \) managed to validate successfully and enqueue first. Therefore \( T \) validates again before retrying to enqueue. Note that by the moment a transaction obtains its place in the parent’s queue it is guaranteed to be valid to commit.

For that to be true, the commit algorithm preserves the following guarantee: When \( T \) is enqueued successfully, all previously enqueued transactions, in the parent of \( T \), finished their commit. This is important to ensure that the validation that \( T \) performs, before enqueuing, can see the changes performed by the commit of all previously enqueued transactions. For this to happen, a lock-free helping mechanism was used so that sibling nested transactions, which are concurrently attempting to commit, first attempt to help previously enqueued siblings. This ensures that there is always global progression: As long as there are active transactions attempting to commit to some parent, there will always be one committing at a time, even if the underlying threads are allowed to fail silently. This way, we guarantee that a sibling always ensures that other siblings that are first in the commit order have already committed successfully before performing its own commit.

Algorithm 3 shows the procedure that helpers execute. Its objective is to propagate the oreecs controlled by commitTx into its parent. This propagation corresponds to a nested commit that acquired the timestamp commitNumber in the commit order of the parent of commitTx. An initial check is performed to verify if the commit has been finished before (lines 4-6). Otherwise, the propagation of the oreecs is performed (lines 7-14). This includes merging those transactions into the committedChildren list of the parent. That list is immutable, and thus only line 15 causes changes in the parent. Setting the new list in the parent uses a CAS to prevent delayed helpers from setting an old list in the parent, which would delete children propagated to the parent by more recent commits. The CAS expects the swapped value to be parentOreecs, which corresponds to the committedChildren of the parent before commitTx enqueued in the parent.

Beyond the lock-free commit, we also have that the read operation of a parallel nested transaction executes in a bounded number of steps and independently of concurrent transactions, thus being wait-free. Moreover, the write operation is also wait-free because the transaction either successfully writes in-place, which it attempts to do once, or it necessarily succeeds to write in the fallback mechanism. Consequently, this parallel nesting algorithm is able to preserve the lock-free progress guarantee of the JVSTM.

4.8 Correctness in the Java Memory Model

The description done so far of the algorithms is not enough to assess their correctness when implemented in Java, because of the relaxed memory model of Java: There need to exist some visibility guarantees of the memory operations to ensure that they behave as expected. Next, we explain how we used the Java Memory Model [4] synchronization primitives to ensure the correct behavior of the algorithms. In particular, which variables are volatile.

We use volatile in the nClock, available in every transaction, so that a nested transaction that starts with version \( v \), is guaranteed to view all changes that were performed up to, and including, the commit with version \( v \). This happens because the committing transaction writes to nClock after making the write entries visible in the parent, and a new transaction reads the nClock of its parent before starting.
Algorithm 3 Propagation of oreCs.

1: HelpCommit(commitTx, parentOrecs, commitNumber):
2: parent ← commitTx.parent
3: newParentOrecs ← parent.commitChildren
4: if newParentOrecs ≠ parentOrecs then
5: return // some helper already committed this tx
6: end if
7: commitTx.orec.nestedVer ← commitNumber
8: commitTx.orec.owner ← parent
9: newParentOrecs ← newParentOrecs.add(commitTx.orec)
10: for child in commitTx.committedChildren do
11: child.orec.nestedVer ← commitNumber
12: child.orec.owner ← parent
13: newParentOrecs ← newParentOrecs.add(child.orec)
14: end for
15: parent.CAScommittedChildren(parentOrecs, newParentOrecs)

Then, we also use volatile in the owner of the oreC. This ensures that concurrent nested transactions always see a nestedVer of the oreC consistent with its owner: The commit is the only operation that changes those fields in the oreC, and it always changes the nestedVer before the owner, whereas another transaction reading or writing always fetches the owner before the nestedVer. The former read creates a happens-before relation with the write of the owner during commit or abort. In particular, when a transaction aborts and deletes some of its write entries (as described in Section 4.6.), it also reassigns the owner field between writing to value and oreC in the in-place write.

5. Evaluation

The results presented in this Section were obtained on a machine with four AMD Opteron 6168 processors (48 cores total) with 128GB of RAM, running Red Hat Enterprise 6.1 and Oracle’s JVM 1.6.0.24. All the performance values shown are the average of five independent runs of each benchmark.

Our evaluation is split in two parts. First, in Section 5.1, we present results for several benchmarks that we modified to parallelize their transactions and, therefore, explore finer-grained parallelism through parallel nesting; the goal is to show that we may get performance gains when using parallel nesting, thereby validating that our algorithm is practical. Then, in Section 5.2, we compare our implementation with the implementation of two other state-of-the-art STMs that support parallel nesting.

5.1. Parallel nesting versus top-level only

To understand how effective our parallel nesting algorithm is, we compare the performance obtained with JVSTM when using only top-level transactions against fewer top-level transactions, with each one using parallel nesting.

Fig. 4 shows these experiments for three known benchmarks:

- STMBench7 [9]: In Figs. 4(a), 4(b) and 4(c) we show the results for the three existing workloads
Figure 4. Speedup relative to a single top-level transaction in several benchmarks and workloads obtained with top-level transactions only and when using parallel nested transactions. The threads used are shown as the number of top-level transactions and number of parallel nested transactions that may run in each top-level transaction. In the top approach, the number of top-level transactions used is the multiplication of those two numbers.

with long traversals enabled. The read-write long traversals access most of the object graph of the benchmark, both with read and write accesses. This precludes much of the possible concurrency, as TMs have to detect conflicts in most concurrent accesses with these traversals. For this reason, we can see that adding more top-level transactions scales very poorly.

We exploited the inner parallelism in each of those traversals \((T_2a, T_2b, T_2c, T_3a, T_3b, T_3c, T_5)\). For this, we identified the parts of those traversals that were meant to be run in parallel. The idea is similar to the example provided in Section 1., as we allow a traversal to explore different paths of the object graph in parallel. At 48 threads, this yielded an increase of 102% in the read-dominated workload; 129% in the read-write workload; and 131% in the write-dominated workload.

- **Lee-TM** [2]: In Figs. 4(d), 4(e), 4(f), and 4(g) we perform the same experiment as above, using different boards. We used parallel nesting by parallelizing the expansion phase, and this approach also yielded some improvements, namely 119% and 170% for mainboard and memboard, respectively. In the case of the sparseshort board, the transactions are very small, which makes the time spent managing nested transactions an overhead that supplants the gains obtained from less conflicts. Therefore parallel nesting only obtained 4% improvement in that board.

- **Vacation** [14]: In this benchmark we explored the parallelism in the transactions by paralleliz-
Fig. 5. Throughput in a high contention workload in Vacation and in the write-dominated workload of the STMBench7 benchmark with long-traversals enabled.

5.2. Comparison with other parallel nesting algorithms

We implemented both NesTM and PNSTM in Java according to the algorithms in their publications. This was the only alternative given that they are not publicly available nor were we able to reach the corresponding authors. To make the comparison fair, we defined an API for these implementations that allows the applications to specify which locations are transactional. This way, we ensure that we are adding the same level of instrumentation to all STMs. Moreover, it is important to take into consideration that this evaluation is necessarily comparing also the underlying TM designs. Because of that, we also executed the workloads using only top-level transactions, so that we may have an idea of how the baseline TMs compare with each other.

Figs. 5(a) and 5(b) present the throughput of each STM in a scenario with high-contention in Vacation. Looking at Fig. 5(a) we may see that the JVSTM is considerably faster than the alternatives when using only top-level transactions. In particular, it obtains 1.58 speedup over NesTM and 5.41 speedup over PNSTM with a single top-level transaction. Given that the baseline JVSTM
is already faster than NesTM and PNSTM, it is expected that using parallel nesting is also faster in JVSTM. We can see that in Fig. 5(b). Still, the actual improvement (for 48 threads) is greater for JVSTM (2.8 times) than for NesTM (2.2 times) and PNSTM (no improvements).

We also show a comparison between the three STMs in a write-dominated workload of the STMBench7 benchmark, with long-traversals enabled. The results are shown in Fig. 5(c), where we may see that JVSTM is again faster already with only one thread: It obtains a speedup of 3.6 and 4.4 over NesTM and PNSTM, respectively, when using only top-level transactions. Moreover, even though we do not show them, the results are very similar across the other workloads of the STMBench7.

Yet, unlike the results obtained in the Vacation benchmark, in this case the parallel nesting algorithm of NesTM is unable (together with PNSTM) to obtain improvements over its execution with only top-level transactions, whereas JVSTM more than doubles its throughput when using parallel nesting.

Finally, we modified both benchmarks so that they execute all of their transactions entirely within a single nested transaction at a certain depth. This yields a nesting tree with a single branch that is increasingly deeper. We present this experiment for STMBench7 in Fig. 6, where we may see how each STM performs when the nesting depth increases. We also performed this experiment in Vacation, and in both cases using 16 threads, and obtained similar results (not shown here).

These results are consistent with the theoretical complexity bounds of each STM. Namely, PNSTM performs independently of the nesting depth, whereas the other two degrade their performance. However, JVSTM not only performs significantly better, but it also degrades at a much slower rate than NesTM. This behavior is similar in both benchmarks and is representative of the data that we collected in several other workloads.

So, just as PNSTM gets better results than NesTM for a sufficiently high depth, we expect the same to happen also, at some depth, with regard to JVSTM. Yet, given the slow decay of the JVSTM, that will require a much higher nesting depth (note that the horizontal axis is growing exponentially). In fact, we argue that such depth would seldom, if at all, be seen in real applications.

6. Related Work

Some TM implementations provide support for linear nesting, which limits a transaction to have only one nested transaction active at a given time. In this case, nested transactions are used only for their compositional properties, as transactions cannot parallelize their contents by running concurrent children. Thus, in the following, we consider only TMs with support for parallel nesting.

The CWSTM [1], which builds on the Cilk language, introduced the combination of the parallel
and spawn constructs to create new threads with assigned nested transactions. It was the first work to show a depth-independent nesting algorithm, but did not provide any implementation or evaluation.

In a different setting, the Sibling STM [17] considered that sibling nested transactions may have relationships and be dependent among each other under the notion of coordinated sibling transactions. This unique perspective means that nested transactions may interfere with each other’s outcome. Yet, their algorithm is not provided in a detailed manner and concentrates more on how to make use of the underlying runtime of choice.

Another approach is NePalTM [19], which was built on top of OpenMP and Intel’s STM to integrate parallel and atomic blocks. NePalTM uses an abstract lock [16] to serialize executions of transactions with the same transactional parent. This is accomplished by having direct children of top root atomic blocks (shallow nested transactions) to proceed optimistically resorting to transactions while deep nested parallel transactions (spawned a given nested transaction) run sequentially in mutual exclusion. As a result, its model is not too powerful, but still allows unveiling some concurrency in transactions as long as they only compose within one level of depth.

Conversely, in the Nested STM [4] the authors extended their earlier work to allow parallel nested transactions. Yet, their algorithm synchronizes commits with mutual exclusion on the parent, which may cause performance penalties when many transactions attempt to commit and the owner of the lock is delayed. The authors also identified that it is possible for their nested transactions to livelock, which they attempt to solve with heuristics. Moreover, their property of invisible reads incurs in an effort that is proportional to the depth of the nesting tree during accesses to constantly revalidate the read-set. HParSTM [11] allows a parent to execute concurrently with its nested transactions, but failed to present any evaluation. It is likely that some of the global structures they used inhibit scalability as it breaks the disjoint-access parallelism property [3] and are intensively used for conflict detection. Finally, PNSTM [5] was based on the ideas that CWSTM pioneered. It provided an efficient depth-independent algorithm, but all accesses are assumed to be writes, which precludes some read-only potential concurrency.

A common trait of all this previous work is that it uses single-version, whereas the implementation that we describe in this paper has support for multi-version. Moreover, our solution also yields better results than the state-of-the-art alternatives with support for parallel nesting while providing a stronger progress guarantee (lock-freedom).

7. Conclusions

In this paper we proposed a novel algorithm for parallel nesting. This algorithm is lock-free and was designed to add minimal overhead to a multi-version STM. This was accomplished by designing the operations of parallel nested transactions with fast paths that correspond to the (expected) common cases of a transactional application using parallel nesting. Moreover, these operations do not affect the operations for top-level transactions.

Our experiments with various benchmarks show that, on one hand, we can get significant performance gains with parallel nesting, and, on the other hand, parallel nesting does not add significant overhead into an application that does not benefit from it.

A key observation of our evaluation is that the best results are attained when two conditions are met: (1) top-level transactions fail to deliver significant improvements with the increase of parallel threads, because of contention among the transactions that inhibits the optimistic concurrency severely; and (2) top-level transactions contain substantial computation that is efficiently parallelizable.
Finally, we showed that our algorithm yields considerable improvements over the state-of-the-art alternatives with support for parallel nesting while providing a stronger progress guarantee.

References


