Technical Report RT/16/2012

Exploring Parallelism in Transactional Workloads

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

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

June 2012
Abstract

Multicores are now standard in most machines, which means that many programmers are faced with the challenge of how to take advantage of all the potential parallelism. Transactional Memory (TM) promises to simplify this task.

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. These large sections may entail writing to shared data, which typically leads to many conflicts in optimistic concurrency control mechanisms such as those used by most TM systems. Yet, sometimes these operations could be executed faster if their latent parallelism was used efficiently, by allowing a transaction to be split in several parts that execute concurrently.

In this paper we provide this increased flexibility by using parallel nesting. Moreover, we propose to overcome inherently sequential highly-conflicting workloads with the new expressiveness provided by TM. We additionally show that the use of conflict-aware scheduling provides an effective solution to maximize the benefits of parallel nesting. Finally, we show how the implementation of these ideas in a lock-free multi-version STM outperforms the original version on several known benchmarks by up to 2.8 times.

Keywords: JVSTM, Nested Parallel Transactions, Conflict-aware scheduling
Exploring Parallelism in Transactional Workloads

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

1. Introduction

The rapid proliferation of multicore processors brought concurrent programming into the forefront of mainstream software development. So, many programmers are now faced with two major challenges when developing their programs. First, how to split their stateful programs into parallel tasks. Second, how to synchronize those tasks so that they can proceed in parallel most of the time but still access the programs’ state in a safe way. These two challenges, however, are deeply intertwined: Not only the need for synchronization depends on how the programs are split into parallel tasks, but also the division of programs into tasks and subtasks depends on how difficult it is to synchronize those tasks. Thus, the lack of appropriate synchronization mechanisms, or the difficulty in using them, may hinder the ability of programmers to parallelize their programs.

Since the mid-sixties that programmers rely mostly on locks to ensure mutual exclusion in the access to a critical section of their code. Unfortunately, correct and effective usage of lock-based synchronization requires great expertise and is highly error prone. Whereas these costs could be acceptable when concurrent programming was confined to a few niches of expert programming, such as the development of operating systems, database management systems, or high-performance computing, they are no longer acceptable when concurrent programming becomes pervasive.

Thus, over the last decade, many researchers have been pushing forward the Transactional Memory (TM) abstraction as a mechanism that greatly simplifies the synchronization problem. TM, originally proposed in [17], allows programmers to concentrate on which parts of their programs should operate atomically, but, unlike locks, does not require that programmers identify which resources are accessed by those atomic operations. Moreover, to ensure the atomicity of the operations, many TM systems do not need to enforce mutual exclusion among the execution of the operations, thereby allowing for greater parallelism.

To use TM, the programmer still has to identify the parallel tasks in his program as usual. But instead of having to deal simultaneously with the intricacies of fine-grained locking—which he must if he is seeking high performance—he is faced with a more simple task of identifying atomic blocks. At runtime, these atomic blocks of code are executed within the context of a transaction. The problem, however, is that TMs inhibit the programmer from performing further division of parallel tasks within the context of transactions. This happens because each transaction is seen as a sequential set of instructions to be executed.

In this paper, we tackle this limitation imposed by TM, by providing synchronization mechanisms that allow a transaction to be itself composed of parallel tasks. This makes TM a more powerful paradigm

---

This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds, by the RuLAM project (PTDC/EIA-EIA/108240/2008) and was partially supported by the Cloud-TM project, which is co-financed by the European Commission through the contract number 257784.
with regard to the division of programs into parallel tasks. But at the same time, the challenge lies in providing such benefit in an efficient way such that we do not preclude potential gains from the parallelism being explored.

The importance of this increased expressiveness, with regard to the identification of tasks when using TM, is clear when we take into account its possible uses. Recall that, rather than relying on mutual exclusion to synchronize the access to shared data, TM starts out from the premise that concurrent tasks seldom contend for the same data. More particularly, it explores concurrency as much as possible while preserving correctness. To do so, transactions that violate correctness are aborted and repeated. Yet, when the application’s workload is write-dominated, in the sense that most transactions perform at least some writes, this may result in a highly-conflicting execution. A conflict takes place between two transactions if there is no equivalent sequential execution ordering of both transactions that explains the result of each individual operation that is part of them. Given a highly-conflicting execution, the optimistic concurrency model used by most TMs cannot overcome a logical barrier in terms of performance: Ultimately, if all the active transactions at some point conflict with each other, the time that takes to execute all of them successfully in parallel will not be less than the time it would take to execute them one at a time in a single-core machine. In practice, the single core would actually be faster due to the TM system overheads and cache invalidation concerns on the multicore.

It is therefore important to pose the following question: Is there any way TM can overcome this limitation? One possible way is to reduce the amount of work that has to be repeated when a transaction restarts. This strategy has been approached by checkpointing [25] and restartable transactions [8]. In common, they attempt to make the most out of a situation in which conflicts already happened. In this paper we depart from those ideas and propose to explore the inner parallelism of transactions. Our claim is that the fact that each transaction may be parallelized can increase the performance in terms of throughput and latency. In the previous scenario of conflicting transactions, the solution would be to parallelize each of the contending transactions and to run them one at a time, therefore reducing the time to complete each transaction and without incurring in conflicts between any two top-level transactions. The benefits of such an approach are possible whenever the code parallelized at the top-level conflicts with a high probability, whereas the parallelization of each task does not incur in that problem. Note that running one top-level transaction at a time in the solution proposed is a simplification: We may very well run other concurrent transactions to fill up all the cores if needed, but the point is that there will be less of those, thus causing less conflicts. The expectation is that the time it takes to execute the critical path of the set of parallelized transactions will be less than the time it takes to execute each of the top-level transactions sequentially. Parallel nested transactions are crucial to achieve this result if the parallelization does not guarantee that each part of the transaction being parallelized is conflict-free from its counter parts. This may be the case depending on the underlying logic of the transaction or if the parallelization is performed automatically [2].

1.1. Main Contributions

The main contribution of this paper is an efficient parallel nesting algorithm that preserves the progress guarantees of the underlying TM. This strengthens the flexibility of the TM as it no longer prevents the division of parallel tasks in some cases.

Moreover, other contributions of this paper are as follows: (1) we show that parallel nesting can improve the performance obtained with TM in some highly-conflicting workloads; (2) we show that the benefits obtained are substantially increased when a conflict-aware scheduler is used; (3) we provide a more flexible TM in which it is possible to parallelize transactions as opposed to the traditional perspective that has seen transactions as a sequential set of instructions; and (4) we show that these
results may be obtained without affecting the normal execution of the underlying TM 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 Software Transactional Memory (STM) that we extended. Next, in Section 3, we introduce parallel nesting. Then, we present a simple design to accomplish parallel nesting in Section 4, and describe the challenges that prevent it from being efficient. In Section 5, we improve on that early design and provide an efficient algorithm. We evaluate these algorithms in known benchmarks in Section 6, and additionally improve the results with scheduling in Section 7. We discuss related work in Section 8, and conclude in Section 9.

2. JVSTM Overview

The Java Versioned STM [13] is a word-based, multi-version Software Transactional Memory (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 ($VBox$) to represent transactional locations. Each $VBox$ holds a history of values for a 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 are logged in a transaction’s read-set, whereas writes are logged in a transaction’s write-set. 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 [13].

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 [20], 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 \( res = ok \)) or failed (when \( res = fail \)).
- \( S_t(t_1, t_2, ..., t_n) \) 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.\(^2\)

In a closed nesting 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.

A closed nested transaction commits by merging its read-set and write-set with its parent’s read-set and write-set, respectively. If it aborts, it may rollback only the atomic action corresponding to itself rather than the whole top-level transaction, depending on the conflict that caused the abort.

4. First attempts at providing parallel nesting

The fact that we may have parallel siblings and, therefore, different branches of a nesting tree active at a given time makes TMs more complex in practice: In linear nesting, a nested transaction can always assume that the write-sets of its ancestors never change during the nested transaction lifetime, whereas for parallel nested transactions that may not be true, because concurrent siblings may commit during the execution of a transaction.

To illustrate the problems raised by parallel nesting, consider the following partial execution of a program that is using parallel nesting:

\[
W_A(y, 5); S_A(B, C); W_B(x, 10);
\]
\[
S_B(D, E); W_E(x, 15) \tag{1}
\]

According to the model described in Section 3., if at this point transaction \( D \) reads \( x \), it must obtain the value 10 written by \( B \), rather than the tentative value 15 written by \( E \). Yet, if after this read by \( D \), transaction \( E \) commits, and, consequently, propagates its write of \( x \) to its parent, \( B \), which value should \( D \) obtain if it reads \( x \) again? Reading the value 15 committed by \( E \) would break 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.

\(^2\)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.
correctness criterion (assuming opacity [14]). Thus, the alternatives are to return the value 10, if the TM is multi-version, or to detect a conflict and abort. This example shows that reads in parallel nesting are more complex than in linear nesting.

Moreover, regardless of whether D rereads x after the commit of E, if D writes to some variables and then tries to commit, it must detect a conflict between its read of x and the new updated value of x on B, and, consequently, fail the commit. Again, this shows that implementing a parallel nested transaction is more complex than a linear nested transaction, because, unlike in the latter case, we must keep a log of values read from ancestors and we must validate those reads on commit (or else, use some other mechanism that allows us to detect or avoid such conflicts).

Finally, another source of added complexity is that nested transactions may need to access concurrently the data structures of their (common) ancestors. This may happen either during the commit, when they must propagate their logs to their parents, or when they read values of variables that may have been written by some of their ancestors. This means that transaction’s data structures must be thread-safe.

4.1. A naive approach

A simple design for implementing closed nesting is to maintain a read-set and a write-set on each nested transaction. Then, when a nested transaction commits, both its read-set and write-set are merged into the parent’s respective sets, overwriting any previous writes of the parent for the same transactional locations. This is the design used by the original JVSTM to implement its linear nesting model, and a similar approach may be used for implementing parallel nesting. We have implemented such a design on top of the JVSTM to serve as a baseline. In the remainder of this paper, we shall refer to this implementation as the Naïve algorithm, and in Figure 1 we show a representation of the nesting tree that is generated by the execution (1) shown above in Section 4.

In such a naive approach, however, the commit of a nested transaction has a complexity that is proportional to $O(R + W)$, where $R$ and $W$ are the sizes of the read-set and of the write-set of the committing nested transaction, respectively. Whereas the propagation of the write-set may be implemented in a more efficient way, a TM with a lazy write-back cannot avoid the cost of validating its read-set, as shown by [1]. Thus, the component of the cost corresponding to $R$ cannot be eliminated in this approach.

Yet, in this design there is a more important challenge to address, which is the cost of the read
To read the value of a VBox, the read operation not only needs to check the write-set of the current transaction, but it must also check recursively the write-set of each of the transaction’s ancestors until a write is found, returning a globally committed value from the VBox if no tentative write is found. This means that a read made on a nested transaction at depth $d$ has, in the worst case, to check the private write-sets of $d$ transactions. Only when we are in the case of a read-after-write (i.e., we are reading from a VBox that was written to before) may the cost be lower, because we stop looking as soon as a write is found. So, to have a better idea about the average cost of a read operation in this design, we need to know how often reads correspond to read-after-write operations.

To assess this, we ran several benchmarks with varying workloads, and counted the total number of reads performed and, of those, how many were read-after-write operations. In these results we do not take into account the reads performed by read-only transactions. In Figure 2 we show the results that we obtained, where the last column shows the percentage of read operations that are read-after-writes.

The majority of the workloads contain very few read-after-writes—that is, a read operation will almost always have to pay the cost of doing $O(d)$ operations, where $d$ is the depth of the nesting tree. So, the worst case of the read operations is also its average case.

### 4.2. A better design using a shared write-set

To tackle the problems identified with the naive approach described above, we proposed a different design in previous work [9]: Make all the transactions within a nesting tree maintain their write entries in a single shared structure, stored at the top-level transaction. The underlying motivation is that this allows for a one-time, depth-independent check to answer the question raised when a nested transaction attempts to read a VBox: Does any ancestor have a private write entry for that VBox?

As previously mentioned, with the scattered private write-sets in the Naive design, writing a value into a VBox entailed buffering it in the private write-set of the nested transaction doing that write. In this SharedWS design, the write is publicized to all transactions that belong to the same nesting tree. The shared data structure that allows this is a ConcurrentHashMap, which, similarly to the private write-sets of the previous design, allows associating VBoxes to tentative write entries. But, whereas the private write-set of each transaction maps each VBox to a single tentative write, the sharedWriteSet maps each VBox that has been written in the nesting tree to all the write entries that were tentatively written to it in that nesting tree.
In Figure 3 we present the nesting tree for the same execution presented previously in Section 4, and depicted in Figure 1, but this time with the SharedWS design applied. The write entries corresponding to the same VBox form a linked list in which the most recent write entry is at the head of the list. The key point in this structure is that we can rely on the following invariant: Given some read lookup on the entries of a VBox in a nesting tree, as soon as the transaction reaches a write entry that may be read, then it is guaranteed that no other entry further down the list has to be read instead of that one. We show in [9] that this new design may entail a worst case on the read operation that is bounded by $O(n)$, where $n$ is the number of transactions in the nesting tree. Even though we have that $d \leq n$, where $d$ is the depth of the nesting tree, the average case is $O(k)$ where $k \ll d, n$. Thus, with this new design, the read operation is typically faster, and, more importantly, independent of the nesting depth.

To validate these claims, we profiled the average time that a parallel nested transaction takes to read, to write, and to commit during the execution of the Vacation benchmark of the STAMP [19] suite. We present the results obtained in Figure 4. Similar results were obtained when profiling several workloads both in the STMBench7 benchmark [15] and in the Lee-TM benchmark [3]. To obtain these results, we forced each top-level transaction to spawn a child and to execute all of its code within the child transaction.

Figure 4(a) shows that the performance of the read operation in the SharedWS design is independent of the nesting depth, leading to a better performance relatively to the Naive design. This happens even at one level of depth, as in that case the Naive design already has to check two different write-sets.

On the contrary, as shown in Figure 4(c), the time to commit still increases with the depth in both approaches, even though the SharedWS design performs better. Despite being more efficient, the propagation of the write-set still entails operations that are proportional to the size of the write-set [9].

Where the SharedWS design stops being on par with the expected result is on the performance of the
writes, as shown in Figure 4(e). When we increase the number of siblings, the cost of writing increases as well. The result in this case was expected to be similar to what is obtained with a single sibling and increasing depth, as shown in Figure 4(b). This is a consequence of the concurrent nature of the data structure underlying that design.\(^3\) With multiple siblings, their actions have to be correctly synchronized, which is especially costly for modifications. When we have just one nested transaction, however, the optimizations made by the Java runtime remove all the synchronization costs, leading to a performance on par with the Naive design.

### 5. A new design for parallel nesting

The SharedWS design solved some of the problems identified in Section 4.1. Namely, it made the cost of the read operation significantly lower and independent of the depth in the average case. But, as shown in Section 4.2., this was accomplished at the expense of making the write operation more costly.

Because a transaction typically reads more than it writes, we claim that the SharedWS design is better than the Naive design with regard to parallel nesting. Yet, in this paper we seek to improve on this design and present an alternative that does not suffer from the problems identified for the SharedWS design.

First, we discuss some properties of the underlying STM and how they should be mapped to the new parallel nesting algorithm. Then, we describe in detail the new design that not only improves the write operation, but that also makes the constants behind the commit time much smaller.

\(^3\)We tested both with Doug Lea’s ConcurrentHashMap and with Cliff Click’s NonBlockingHashMap, but the results were similar.
5.1. Progress guarantees

The STM underlying this work, briefly presented in Section 2., is lock-free, which means that a top-level transaction cannot prevent all other concurrent top-level transactions from progressing, as the system as a whole must make progress.

But, when we take into account the parallel nested transactions spawned in the scope of a nesting tree, we are implicitly attributing the following semantics to them: All of the siblings must commit successfully for the parent to be able to commit—i.e., it does not make sense for the parent to succeed while some of its children are still executing. Therefore, if one sibling does not progress, the parent will not be able to progress either. In this sense, it is not harmful if a nested transaction inhibits its siblings from progressing, as the progress of the nesting tree as a whole requires that all the transactions in it succeed. For this reason, the algorithms presented in this paper for parallel nested transactions are blocking. Note that this does not change in any way the fact that the JVSTM is still lock-free: A parallel nested transaction may never prevent other transactions (nested or not) from progressing unless they have a common ancestor, meaning that they belong to the same nesting tree.

Thus, top-level transactions continue to execute in a lock-free manner and to proceed with their validation in parallel. In particular, this means that there is no lock that must be acquired at commit-time, and for which the top-level transactions contend. On the other hand, parallel nested transactions contend for the same lock only if they are siblings, which reduces substantially the window of opportunity for contention. This is yet another reason that strengthens the need to shorten the commit time in parallel nesting, so that we may apply a blocking nesting scheme that does not suffer from contention.

In this new design, the read operation of a parallel nested transaction executes in a bounded number of steps and independently of concurrent transactions, thus being wait-free. In contrast to this, the write operation may be delayed by concurrent transactions in the same nesting tree. These two
operations are described in Sections 5.4. and 5.5., respectively. Moreover, the commit of a parallel nested transaction, described in Section 5.7., contends for the same lock as the commit of its siblings only. Therefore, commits of parallel nested transactions at different nesting levels or nesting trees proceed independently.

5.2. Multi-versioning for nested transactions

Multi-versions allow the implementation of read-only transactions that always commit and that have very few overheads because no read-set is needed. In the case of nested transactions, however, we cannot discard the read-set of a nested read-only transaction that is part of a read-write nesting tree; we still have to register the reads in the read-set to propagate them to the top-level read-write transaction.

Even though we cannot benefit of the optimizations related to the read-set, we believe that it is still worthwhile to have nested read-only transactions that never abort: Applications tend to contain blocks of code in which they perform several read operations first and then perform the writes towards the end of the block. Therefore, having nested read-only transactions that never abort allows us to run both parts in parallel, without being concerned that the writes of the second part may cause the first part to restart and see values that would change the semantics of the entire operation. Therefore, in this new design we maintain the decision of using multi-versions for boxes that are written by more than one nested transaction within a nesting tree.

5.3. Algorithm with in-place writes

We have previously identified the maintenance of a shared data structure per nesting tree as a source of overhead, specially when we need to change it. Recall that transactional locations, represented by VBoxes, point to a history of all the versions for that location that may still be readable by some active transaction in the system. We propose to extend this design such that transactions perform their writes on these locations rather than having to maintain some private mapping of each location written to its new value (regardless of being a private write-set per transaction or per nesting tree). Consequently, a VBox now contains both permanent and tentative versions. The permanent versions have been consolidated via a commit of some top-level transaction, whereas the tentative versions belong to an active top-level transaction or any its children that form a nesting tree.

To illustrate this new design, we show in Figure 5 how the example previously shown in Figure 3 changes to accommodate the different way to handle the tentative writes. We call this new design Inplace.

In this case, the tentative writes of the represented nesting tree have been moved from the shared write-set of that nesting tree to the tentative values of the corresponding VBoxes. But the shared write-set continues to exist as a fallback mechanism to store tentative writes. This is needed because only one nesting tree may use the in-place tentative location of a box at a given time. So, if more than one nesting tree writes to the same box, only one of them will be able to use the box; all the others will have to use their shared write-set to store their tentative values. This strategy is detailed in Section 5.6..

So, as long as different nesting trees do not contend for writing in the same VBox, transactions will be able to follow the fast path of the new write algorithm, thereby making writes faster. In practice, we expect this to be the common case.

Figure 6 presents the total number of writes and, of those, the total number of write-after-read oper-
Figure 6. Total number of writes and write-after-read (wars) performed in various workloads of the STMBench7, the Lee-TM, and the Vacation benchmarks.

<table>
<thead>
<tr>
<th>Test</th>
<th>Writes ($10^3$)</th>
<th>Wars ($10^3$)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>bench7-r-notrav</td>
<td>6158</td>
<td>6158</td>
<td>100</td>
</tr>
<tr>
<td>bench7-rw-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-rw</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>

Figure 5 also depicts that the tentative writes no longer point directly to their current owner. Instead, we use the concept of Orecs (short for ownership records [23]) that provide a level of indirection to a reification of the ownership. This way, we are able to propagate the write-entries at commit time independently of the size of the write-set, thus addressing another challenge identified earlier. As shown in Figure 5, an Orec contains:

- **owner**: The transaction that is being represented by this Orec.
- **status**: Identifies whether the transaction is still running, has aborted, or has committed.
- **commitVer**: Timestamp acquired by incrementing the global clock in the top-level commit of the owner of this Orec. This is the value used to timestamp the versions created globally in the write-back procedure of a top-level transaction. If the owner of this Orec has not committed, or if it is not a top-level transaction, then this value is -1.
- **nestedVer**: Timestamp acquired in a nested commit into some parent by incrementing the parent clock. This value only makes sense when paired with the owner. It is used to ensure that reads on tentative entries are consistent.

To simplify the presentation, in the description of the previous designs, we omitted the details about the use of versions. Yet, to ensure consistent reads and to preserve multiple versions, each transaction contains a number representing the latest version that has been committed into it by its children. Each nested transaction will use these numbers present on each ancestor to compose a set of version numbers associated to each of its ancestors (we name it the ancestorVersions set): When a nested transaction is spawned, it computes the union between its parent’s current number and its parent’s ancestorVersions set to obtain its own ancestorVersions set. The ancestorVersions set
effectively represents the restrictions that the nested transaction has on the view of the nesting tree’s write entries that it may read. The usage of the `ancestorVersions` will be presented in the next section where we explain the read operation in this new design.

We shall also be referring to structures that each transaction has to maintain. In particular, a parallel nested transaction holds the following fields:

- **FileVersion**: Timestamp obtained by reading the global clock in the start of the transaction. It is used to ensure that the transaction always sees a consistent view of the shared data.

- **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, a reification of the in-place tentative write is registered.

- **rootAncestorWriteSet**: 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. In this design, this structure is used in the fallback mechanism when the transaction cannot write in-place to some `VBox`.

- **boxesWritten**: Maintains the 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 new versions in the `VBoxes`.

- **nestedCommitClock**: Incremented by each children of this transaction that commits into it. It is used by the children spawned by the transaction, so that they can create their `ancestorVersions` map.

- **ancestorVersions**: Maps the ancestors of the transaction to the timestamps obtained from their clocks.

- **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). This field is used at commit time of a nested transaction.

Given this description of the data structures used by the new parallel nested transactions, we may now describe in detail, in the following sections, the transactional operations performed by a nested transaction.

A key aspect of this implementation of parallel nesting for the JVSTM is that the algorithms described in the following affect only the nested transactions. This means that the operations of a top-level transaction remain the same [13], and that the changes in the original algorithm of the JVSTM affect its performance only if parallel nesting is being used. Thus, the original performance is not affected by the extension that we performed. In particular, read-only transactions preserve their optimizations and fast paths.
Algorithm 1 Reading from a VBox.

1: \textbf{GetBoxValue}(tx,vbox):
2: \hspace{5pt} wInplace \leftarrow vbox.tentative
3: \hspace{5pt} orec \leftarrow wInplace.orec
4: \hspace{5pt} \textbf{if} orec.status = Committed \land orec.commitVer \leq tx.startVersion \textbf{then}
5: \hspace{20pt} value \leftarrow \text{readFromPermanent}(vbox)
6: \hspace{20pt} tx.globalReads.add(vbox)
7: \hspace{20pt} \textbf{return} value
8: \hspace{5pt} \textbf{end if}
9: \hspace{5pt} \textbf{while} wInplace \neq \text{null} \textbf{do}
10: \hspace{10pt} orec \leftarrow wInplace.orec
11: \hspace{10pt} nVer \leftarrow orec.nestedVer
12: \hspace{10pt} owner \leftarrow orec.owner
13: \hspace{10pt} \textbf{if} tx.ancestorVersions.contains(owner) \textbf{then}
14: \hspace{15pt} \textbf{if} nVer \geq tx.ancestorVersions.get(owner) \textbf{then}
15: \hspace{20pt} \text{revertWrittenEntries}()
16: \hspace{20pt} \text{abortUpTo}(owner)
17: \hspace{14pt} \textbf{end if}
18: \hspace{10pt} tx.nestedReadSet.put(vbox,wInplace)
19: \hspace{10pt} \textbf{return} wInplace.value
20: \hspace{10pt} \textbf{end if}
21: \hspace{10pt} \textbf{if} owner = tx \textbf{then}
22: \hspace{15pt} \textbf{return} wInplace.value
23: \hspace{10pt} \textbf{end if}
24: \hspace{10pt} wInplace \leftarrow wInplace.previous
25: \hspace{10pt} \textbf{end while}
26: \hspace{5pt} value \leftarrow \text{rootAncestorWriteSet.get}(vbox)
27: \hspace{5pt} \textbf{if} value \neq \text{null} \textbf{then}
28: \hspace{10pt} tx.nestedReadSet.put(vbox,value)
29: \hspace{10pt} \textbf{return} value
30: \hspace{5pt} \textbf{end if}
31: \hspace{5pt} value \leftarrow \text{readFromPermanent}(vbox)
32: \hspace{5pt} tx.globalReads.add(vbox)
33: \hspace{5pt} \textbf{return} value
Algorithm 2 Writing a value to a VBox.

1: \textbf{SetBoxValue}(tx, vbox, value):
2: \hspace{1em} orec ← vbox.tentative.orec
3: \hspace{1em} \textbf{if} orec.owner = tx \textbf{then}
4: \hspace{1em} \hspace{1em} wInplace.value ← value
5: \hspace{1em} \hspace{1em} return
6: \hspace{1em} \textbf{end if}
7: \hspace{1em} \textbf{while} true \textbf{do}
8: \hspace{2em} wInplace ← vbox.tentative
9: \hspace{2em} orec ← wInplace.orec
10: \hspace{2em} \textbf{if} orec.status \neq \textit{Alive} \textbf{then}
11: \hspace{3em} \textbf{if} orec.version \leq tx.startVersion \textbf{then}
12: \hspace{4em} \hspace{1em} \textbf{if} wInplace.CASowner(orec, tx.orec) \textbf{then}
13: \hspace{5em} \hspace{1em} \hspace{1em} wInplace.value ← value
14: \hspace{5em} \hspace{1em} \hspace{1em} tx.boxesWritten.add(vbox)
15: \hspace{5em} \hspace{1em} \hspace{1em} return
16: \hspace{4em} \hspace{1em} \textbf{end if}
17: \hspace{3em} \hspace{1em} continue // a concurrent tx succeeded
18: \hspace{2em} \textbf{end if}
19: \hspace{2em} break // not enough version, fallback
20: \textbf{end if}
21: \hspace{2em} \textbf{if} tx.ancestorVersions.contains(orec.owner) \textbf{then}
22: \hspace{3em} newW ← (value, tx.orec, wInplace)
23: \hspace{3em} \textbf{if} vbox.CASinplace(wInplace, newW) \textbf{then}
24: \hspace{4em} tx.boxesWritten.add(vbox)
25: \hspace{4em} return
26: \hspace{3em} \textbf{end if}
27: \hspace{2em} // another tx in same nesting tree succeeded
28: \hspace{2em} continue
29: \textbf{end if}
30: \hspace{2em} \textbf{if} sameNestingTree(tx, orec.owner) \textbf{then}
31: \hspace{3em} releaseOwnership()
32: \hspace{3em} owner ← orec.owner
33: \hspace{3em} \textbf{while} owner.isAborted \land \neg tx.ancestorVersions.contains(owner) \textbf{do}
34: \hspace{4em} \hspace{1em} owner ← orec.owner
35: \hspace{4em} \textbf{end while}
36: \hspace{3em} abortUpTo(lowestCommonAnc(owner))
37: \hspace{2em} \textbf{end if}
38: \hspace{2em} break // out of options, fallback
39: \textbf{end while}
40: \hspace{1em} releaseOwnership()
41: \hspace{1em} executeAsTopLevel()
5.4. Reading a VBox

When reading a VBox, we may fall in the case of the aforementioned read-after-write. For this reason, the pseudo-code presented in Algorithm 1 either returns a global value, represented by a consolidated version in a VBox, or a tentative value written by some transaction in this nesting tree.

In lines 4-8 we test for a fast path that allows us to decide if there is any chance that a write may have been done to the VBox in this nesting tree. If we find it to be impossible, then the global value is returned. To reach such conclusion, we need to verify the following condition: The transaction controlling the VBox has committed before this transaction started. The information required for this is present in the Orec, namely, the status and the commitVer fields.

If the fast path is not used, then the algorithm iterates over the writes in the tentative list of the VBox until one of the following conditions is verified:

1. The current owner is an ancestor, in which case the write must have been made before this nested transaction started (lines 13-20). We require this because, if this nested transaction ever attempted to read a stale version, it would be guaranteed to fail the validation at commit-time, as there is already a more recent version. This happens because the serialization point of the transaction takes place at commit-time (assuming it performs writes). If the maximum version readable from that ancestor is outdated, this transaction causes a chain abort such that the nested transactions up to that ancestor restart with the most recent versions on their ancestorVersions map.

2. The current owner is the transaction attempting the read, in which case no further verification is needed to read it (lines 21-23).

In Figures 7(a) and 7(d) we may see that the read operation with the Inplace design is able to achieve better performance while remaining independent of the nesting depth.

The algorithm presented here dictates that once a nested transaction finds a tentative write that it may read, it stops and returns that value. For this to be correct, we rely on the maintenance of the following invariant: Given a read operation in 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. Consider a simple scenario that illustrates why this is important: A deep nesting tree, in which every transaction of one branch wrote to the same VBox. In this case, a read would possibly need to traverse many tentative writes. By ensuring this invariant, we are able to shorten those traversals as all the transactions will read only the tentative write at the head of the list.

5.5. Writing to a VBox

Algorithm 2 presents the pseudo-code of the write operation under parallel nesting. When accessing the VBox, it fetches the tentative write at the head (line 2). By reading the Orec in the first field of the tentative write entry, it is able to tell whether that VBox is currently owned by the transaction. Otherwise, after line 6, the algorithm repeats until one of the following conditions is verified:

- The current VBox owner has committed 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 10-19). To do so, a compare-and-swap (CAS) is attempted, to change the ownership
Figure 7. Profiling data of the transactional operations updated from Figure 4 with the Inplace design.

of the first tentative write. If the CAS fails, the process is retried. If the previous owner had committed, but 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 proceeds to lines 40-41, to the fallback mechanism. This restriction is what allows us to maintain the fast path presented in the read operation in line 4 in Algorithm 1.

- The VBox is owned by an ancestor of this transaction (lines 21-29). This is true if the current owner is present in the ancestorVersions restriction map 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 process is repeated.

- We are left with two alternatives: Either the VBox is controlled by another nesting tree, or by a different branch of this nesting tree. In the former case, the algorithm proceeds to lines 40-41, to the fallback mechanism. In the latter case, this transaction releases its owned VBoxes and repeats once that VBox belongs to an ancestor (lines 30-37). This condition will necessarily happen as the current owner is from the same nesting tree.

Note that in the initial verification for a write-after-write, we only check the first tentative write. From the algorithm that we have described, we may conclude that if the head tentative write does not belong to this transaction, then no other tentative write down the list may belong to it, as for that to happen, there had to be two transactions executing concurrently that had successfully performed a write in-place to the same VBox.

Much of this complexity arises from the fact that we are maintaining the previous writes in the nesting tree. Yet, that is a requirement from which we cannot escape, as it becomes clear in the following scenario:

\[ W_A(x, 1); W_A(y, 42); S_A(B, C); R_B(x, 1); W_B(y, 0); W_C(x, 2); C_C(ok); C_B(fail) \]

Assume that the write performed by nested transaction \( B \) overwrote the write performed by its parent \( A \) in VBox \( y \). Given that \( B \) aborts due to a read-write conflict, then, in the end, the write performed
by $A$ to $y$ is missing as it was speculatively overwritten by $B$, which failed to commit. Also, note that upon the restart of $B$, its re-execution may follow a different path that does not cause it to write to $y$ again.

Given this new algorithm for the write operation, we may now look into Figures 7(b) and 7(e). We may see that this new design allows much cheaper write operations that are independent of both the number of siblings and the depth.

Even though it would be possible to support concurrent writes in-place from different top-level transactions (or their nesting trees), that would require polluting the fast path with additional verifications. This is particularly relevant for the algorithm for top-level transactions, which we omit here, as it is outside the scope of this paper. Moreover, as pointed out earlier with regard to Figure 6, 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.

We additionally inhibit concurrent writes in the same nesting tree, as a nested transaction is able to acquire ownership of a $VBox$ only if the $VBox$ belongs to an ancestor (or to no active transaction at that time). The reason behind this decision is that allowing concurrent writes in the same nesting tree would force us to do additional work at commit-time, so that we maintain the invariant mentioned in Section 5.4. for faster reads. To illustrate the problem, 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 write entry at the head of the tentative list of $VBox$ $x$ 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 work around this issue, the solution is to perform work proportional to the size of the write-set at commit-time [9], given that we want to preserve the invariant for faster reads. Thus, we chose not to allow concurrent writes in the same nesting tree.

Finally, there should not be any significant contention for writing to the same $VBox$. The rationale presented above for top-level transactions apply also to nested transactions in the same nesting tree. Moreover, recalling the motivation provided in Section 1., one of the reasons behind using parallel nesting in the first place is that we are expecting to move to a less conflicting workload by exploring a different scheduling approach. Consequently, the inner parallelization of a transaction should be such that we do not incur in write-write contention for the same variables, thus making this possible trade-off, of not having concurrent writes in the same nesting tree, negligible.

5.6. Fallback mechanism

In the event of write-write concurrency in the same $VBox$, a fallback mechanism is put in place that allows transactions to proceed. 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 attempts to control the $VBox$ (in case it was not a write-after-write), if the current owner has committed before it started; otherwise, it resorts to the 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 $VBox$ is cheaper than maintaining the mentioned mapping, for which reason this algorithm also yields benefits for the top-level transactions. More importantly, they also benefit from the fast path in the read procedure. This same idea was also implemented in the top-level transactions of the early designs presented in this paper, thus making the comparison between the various designs for parallel nesting fair.
Therefore, when a parallel nested transaction traverses all the tentative writes in a VBox to read it, without success, it still needs to check the private write-set of the root (top-level) transaction of its nesting tree (lines 26-29) in Algorithm 1. It may have happened that the root, prior to the execution of its children, attempted to write to that VBox without success, having thus had to fallback to the private write-set. Otherwise, the nested transaction is sure that it is not incurring in a read-after-write and fetches the global value.

On the other hand, when the write procedure determines that the VBox is currently controlled by another nesting tree (or simply by another top-level transaction), it also must deal somehow with the write-write concurrency gracefully (lines 40-41). In that event, we abort this nested parallel transaction up to the root, and the affected nested transactions in the chain are re-executed by the root (top-level) transaction. In particular, the specific write that caused this fallback will necessarily be performed by the top-level transaction in its private fallback write-set. We could employ a less radical fallback mechanism, such as one of the early designs that we presented in this paper. Yet, the validation procedure would have to be particularly complicated, with many combinations of write procedures taking place, and thus slowing considerably the performance at commit-time. Therefore, in this case simplicity also meant better performance.

5.7. Committing 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 propagate the read-set and write-set of the transaction to the parent. The validation of a transaction $T_i$ is performed by accessing directly the VBoxes that were read and looking into their tentative writes. If $T_i$ read a global value from VBox $x$, because there was no tentative write from its nesting tree at the time that it could read, then there cannot exist at this time a tentative write in $x$ that it may read, unless it is currently owned by $T_i$. 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. Once again, this is correct only because we ensure the invariant that was described earlier.

This validation is speculatively executed up to the most recent commit at that time present in the parent. Afterwards, the commit lock of the parent is acquired, at which point the nestedCommitClock on the parent is compared to the one used for the speculative validation. If it is the same, which
happens if there is little contention in committing to the same parent, there is no need to revalidate the transaction under the lock.

Regarding the merging of the write-set, we sought to make it as lightweight as possible. For that, we used Oreces that allow us to change the owner to the parent, and the nestedVer to the commit version obtained on that parent. This way, we are publicizing the writes to the parent (and its other children) independently of the size of the write-set. Naturally, this requires the propagation of the Oreces themselves, so that the process may be recursively executed at different nesting levels. This is achieved with the committedChildren field in each transaction that contains a list of its children that successfully committed into it, which it will also propagate to the parent at commit-time.

The reads must also be merged into the parent. To do this, we use the committedChildren field, because it allows accessing the reads performed by those children. Consequently the read-set ends up being scattered among the nesting tree. At commit-time, this results in the validation procedure of a transaction traversing different read-sets of the children in addition to its own. The slight overhead of having a level of indirection for accessing the different read-sets is far overcome by the benefit of not having to perform work proportional to the size of the read-set at each commit.

In the end, Figures 7(c) and 7(f) show that the commit procedure benefits from these improvements, particularly with the increasing depth. Although the time to commit still increases as the depth increases, it happens at a much slower rate. This happens because the commit execution (under the lock acquisition) is now bounded by $O(\#\text{children})$ rather than $O(\text{size(writeSet)} + \text{size(readSet)})$. Otherwise, it is bounded by $O(\#\text{children} + \text{size(readSet)})$. Therefore there is less room for contention to the same commit locks given that the time within those has been largely shortened. Recall that the lazy write-back nature of the underlying STM, which allows further concurrency, also dictates that some work proportional to the size of the read-set must be executed at each nested commit [1].

In the event of an abort, the transaction simply changes the status field of the Oreces that it controls 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 has aborted. There is, however, a caveat. Consider the following execution: $W_A(x, 1); S_A(B); W_B(x, 2); \ldots; C_B(\text{fail})$. After the abort of $B$, if some concurrent top-level transaction (or a nested transaction in its nesting tree) 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. Therefore the aborting nested transactions must delete their writes from the tentative lists, but only when those writes were performed when an ancestor was already controlling the VBox.

6. Discussion of the inner parallelization

In this section, we discuss the benefits that we may attain from using parallel nesting. All the results presented next (as well as throughout the paper) were obtained from the average of five runs 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.

The Vacation benchmark of the STAMP suite represents the typical scenario that we motivated for earlier: Under high contention, it becomes increasingly hard to obtain improvements in terms of performance by adding more threads (and having a correspondingly number of available processors). That is clear in Figure 8(a), where we may see that the nonest approach with only top-level transactions is unable to scale properly.
Figure 9. Each plotted execution uses an increasing number of nesting depth in which each top-level transaction creates a nesting tree with a branching factor of one. The speedup is computed relatively to the execution without nesting.

In this case, however, we were able to parallelize the transactions that compose the workload of the benchmark, and, therefore, run fewer top-level transactions at a time with each one spawning an increasing number of parallel nested transactions. Comparing the results of the different designs described in this paper, we see that the Inplace design is able to obtain the best results by obtaining 2.8 times better performance than top-level transactions.

On the other hand, Figure 8(b) exemplifies a workload with low contention. In this case, the nonest is already achieving reasonable performance as the thread count increases. Thus, applying the parallel nested transactions does not yield any extra performance. In practice, we may actually see that there is a slight overhead from executing the transactions with some nesting.

In Figure 9(a), we evaluate that overhead in the Vacation benchmark. The workload was executed with an increasing forced nesting depth with the various parallel nesting designs, and we measured the speedup relative to an execution without nesting. The nesting trees formed used a branching factor of one, thus taking no advantage of parallel nesting, and enabling us to assess the overhead of the algorithms. It is visible that the Naive design increases significantly its overheads as the depth is increased. On the other hand, both SharedWS and Inplace remain steadily stable, with the latter presenting on average 5% of overhead in performance relative to top-level execution. This behavior was similar in a write-dominated workload with long traversals in the STMBench7 presented in Figure 9(b). Note that the performance of the original algorithm of the JVSTM, against which this comparison is being performed, has not had its performance affected when only executing top-level transactions.

We also present individual results for the inner parallelization of STMBench7 in Figure 10. The read-write long traversals present in this benchmark are particularly troublesome for obtaining an increase in performance in workloads that contain them. Therefore, we exploited the inner parallelism in each of those traversals and show the resulting performance relatively to an execution of those traversals with a top-level transaction only. Traversals $t_2b$, $t_2c$, $t_3b$, and $t_3c$ are particularly large in terms of their footprint, which allows the Inplace design to widen the benefits attained relatively to the other two designs. In particular, some slowdowns are visible with both Naive and SharedWS. Traversal $t_5$ proved to be difficult to parallelize: None of the designs to was particularly helpful in this case. Besides the individual results, we present also the results for a combined mix of these traversals under the label tx.
6.1. Relinquishing the overheads of parallel nesting

Despite the overheads identified and quantified in the previous sections, our expectation was that those overheads could be masked by the newly explored parallelism and a more efficient scheduling that avoids conflicts. As shown in Section 6., that expectation proved to be correct for the Vacation and the STMBench7 benchmarks.

When applying the same strategy to the Lee-TM benchmark, however, it generally yielded slowdowns. But while parallelizing the code that composed the transactions in Lee-TM, we realized that we were in the presence of embarrassingly parallelism. More specifically, we could guarantee that the parallel nested transactions that we were creating would never conflict with each other in the same nesting tree.

![Figure 10. Speedup of the read-write long traversals of the STMBench7 relative to its sequential execution. The three parallel nesting designs were used to explore the inner parallelism of the traversals with up to three threads each. The tx execution refers to a combined mix of all the read-write long traversals.](image)

In these cases, we may apply the same general idea, with the advantage that we can use several threads running in the context of the same top-level transaction. The benefit of this is that the transactional operations follow the same algorithm as a top-level transaction, rather than the more complex version presented in this paper for nested transactions. Additionally, at commit-time, these threads need not execute any validation.

In Figure 11, we show the application of this strategy compared with the usage of single-threaded top-level transactions. Once again, we may see that adding more top-level transactions does not necessarily translate into better scaling. In turn, allowing each top-level transaction to run faster yields a better performance, being 2.2 times better at 48 threads.

The expansion phase of the Lee-TM benchmark collects many transactional reads, which induce an increased likelihood of conflict as the transactions get larger. This can be overcome by applying an early release technique [16, 12] to obtain better scaling with top-level transactions. Yet, it was an interesting case to illustrate embarrassingly parallelism that may arise in top-level transactions. The solution of running more than one thread in the context of a single transaction is applicable only if the structures of the transaction are thread-safe. To reduce this concern, we maintain the footprint scattered across the threads executing in the same transaction, such that they never have to write in shared structures. Therefore this solution may be seen as a lower bound in terms of overhead of parallelizing a transaction.
Figure 11. Usage of embarrassingly parallelized top-level transactions in Lee-TM with the mainboard test. The top executes single-threaded top-level transactions only whereas the mt-top uses more threads per top-level transaction. The threads used are shown as the number of top-level transactions and number of threads 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.

7. Conflict-aware scheduling

We have shown, with regard to Figure 10, that the individual read-write long traversals of STMBench7 may benefit from parallel nesting. Yet, we still have not looked into the results of applying this idea to the workloads of the benchmark.

In Figure 12, we use all three workloads with long-traversals in STMBench7: Read-dominated, read-write, and write-dominated. We disabled structural modifications in all three workloads. In the dashed lines, we may see the normal execution with top-level transactions, which obtains very small performance gains and most of the time even slowdowns, due to the disruption created by read-write long traversals.

We also show the executions that resort to parallel nesting in those traversals. Once again, the labels in the thread count shows the number of nested transactions available to each top-level transaction in the case of the parallel nested approach. Parallel nesting starts out with improvements while using a single top-level transaction, but after that it drops considerably in the most conflicting workloads. The reason behind this behavior is that those workloads are causing conflicts between different nesting trees. Generally, the result is still on par or better than the normal execution. However, we are far from the expected results.

Recalling the motivation introduced in Section 1., we have identified the inner parallelizable tasks inside conflicting top-level transactions, but we have not yet ensured that these tasks are scheduled in such a way that exploits their advantages. In practice, we need to schedule the transactions in a different way to ensure that the benefits are attained all the time.

7.1. Scheduling transactions

There has been some work that explored scheduling of transactions as opposed to the usage of Contention Managers [16]. The latter are merely reactive, in the sense that they act upon a conflict detection, and decide on the faith of the conflicting transactions by declaring which one must abort. On the other hand, scheduling acts pro-actively, by attempting to run concurrently transactions that do not conflict with each other.

Yoo and Lee [26] proposed Adaptive Transactional Scheduling, in which threads maintain a notion
Figure 12. Speedup of the three workloads with long-traversals available in STMBench7 with various degrees of read-only transactions with an increasing number of threads, relative to the sequential execution of a top-level transaction. The dashed lines use only top-level transactions and thus the thread count is the multiplication of the threads available in the horizontal axis.

of contention: When a threshold is reached in a thread, it is declared to be under high contention, and the transaction that it is executing is serialized in a queue. CAR-STM [10] uses a serialization technique that guarantees that if two transactions conflict with each other, one of them is executed by the thread of the other such that it is guaranteed not to conflict with the same transaction in its re-execution. Also similarly, Steal-on-Abort [4] allows a thread to steal conflicting transactions from concurrent threads.

Based on this related work, we created a simple conflict-aware scheduler that allows us to confirm our expectations and improve on the previous results. We do not seek any novelty in the scheduler itself, but rather use the related work as a means to an end. When a transaction starts, the scheduler is queried with a unique identifier that is related to the transaction being executed. An example of this, also used in the related work in unmanaged languages, is to use the current program counter at the start of the transaction. This way, the scheduler is able to maintain a table of conflicts between the uniquely identified transactions of the application. This table is populated when a conflict happens.

Figure 13. Performance obtained in various workloads with long-traversals of the STMBench7 with and without scheduling.

On a different fashion, Shrink [11] provides a technique that uses the previous accesses in the same thread to predict the footprint of the next transaction. Thus, it is able to estimate heuristically if the next transaction will conflict with an active concurrent transaction.
between two transactions, among which, one already committed, given the nature of lazy write-back of the JVSTM. Consequently, when the scheduler is queried at the start of a transaction, it is able to tell if there is any conflicting transaction running at that moment. If that is not the case, it adds this transaction to the running transactions. Otherwise, the transaction is enqueued in the thread that is already running the conflicting transaction, effectively serializing it and avoiding the conflict. Therefore, we sometimes refer this scheduler in the results as serial. The worker threads verify if they were given any work during their last execution before proceeding to the next task.

We used this scheduler in the workloads of STMBench7 with long traversals, and we show in Figure 13 the speedup relative to a sequential transaction. Note that we are not yet using any inner parallelism; this experiment is meant to assess the benefits obtained from the scheduler itself. The scheduling itself provides an increase of 12% of performance in the read-dominated workload, and 20% in both read-write and write-dominated workloads, over the best performance obtained without scheduling.

Considering the behavior of the scheduler, its give-away mechanism is inherently creating a critical path composed of the transactions that would conflict with each other if ran concurrently. The predictable consequence of having under-usage of processors is visible in Figure 14, where we present how much time each thread executed transactions during a write-dominated workload in the STMBench7. In particular, the right figure contains a much more erratic distribution due to the usage of the scheduler when compared to the execution in the left figure that did not use scheduling; note that the vertical axis is significantly different in both figures. As a matter of fact, it is relevant to take into account that there is more execution time spent in transactions that conflict and abort in the scenario without scheduling.

### 7.2. Exploring unused processors

So far, we have seen that parallel nesting cannot always provide positive results, as shown in Figure 12. On the other hand, Section 7.1. built on the related work in scheduling to improve the time to execute the conflicting workloads with long traversals in STMBench7. Yet, this resulted in having part of the available processors being idle for the execution of those workloads. This result was expected: If the workload is inherently conflicting, the transactions will contain some sequential critical path free of conflicts that will lead to the successful execution. This critical path necessarily imposes an under-usage of the available processors. Therefore, the idea is that we may now explore these resources to apply parallel nesting and improve the results even further.

When the scheduler is queried for the start of a new transaction, it is able to estimate the current
Figure 15. Different workloads of the STMBench7 with long traversals with four combinations: (1) normal execution with top-level transactions; (2) the first case with the serial scheduler; (3) using parallel nesting; and (4) using parallel nesting and the serial scheduler.
usage of the underlying processors based on how many transactions are currently executing. This heuristic allows it to return to the JVSTM whether it should allow parallel nesting in the current transaction, or whether it should execute it only as a top-level transaction sequentially. In Figure 15 we present the results obtained with this new strategy that takes advantage of both parallel nesting and scheduling applied to the STMbench7 workloads with long traversals. Looking at the best results obtained with top-level transactions we may see that the proposed strategy is approximately 2 times better throughout all the workloads.

8. Related Work

There are some TM implementations that 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 [22] 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 [24], which was built on top of OpenMP and Intel’s STM to integrate parallel and atomic blocks. NePalTM uses an abstract lock [21] 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 [6] 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 [18] 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 [5] and are intensively used for conflict detection. Finally, the PNSTM [7] 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 key 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-versions. We also depart from the motivation of allowing multi-threaded code within transactions: We recognize its importance but rather as a means to an end to increase the performance of applications using TM in highly-conflicting workloads.
9. Conclusions

In this paper we proposed a novel solution to overcome the limitations imposed by optimistic concurrency control mechanisms, such as TM, in highly-conflicting workloads. For that, we presented a low-overhead parallel nesting algorithm for a lock-free multi-version STM. We have provided the first complete implementation and evaluation that took advantage of parallel nesting to overcome highly-conflicting workloads in known benchmarks. We additionally explored transaction scheduling to maximize the benefits of parallelizing long-running disruptive read-write transactions. To reach the results presented, we introduced solutions for the challenges that arise with parallel nesting. In particular, we approached the dependencies on the footprint size and the nesting depth in the commit and in the read operations, respectively.

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) each top-level transaction contains some substantial computation that is efficiently parallelizable.

It is perfectly acceptable that an application programmer identifies some code as meant to run within a transaction, while at the same time willing to parallelize that same code. In that sense, we provide the tools that allow him to do so. In particular, the reason behind doing that may very well be to overcome the limitations of TM in conflicting workloads that we identified in this paper. Yet, we point out that it is also acceptable to consider automatic parallelization tools that may create parallel tasks within the control flow of parallel code. At the same time, this has been achieved by using TM for synchronization of access to shared data [2], in which case the work proposed in this paper is once again crucial.

References


