Parallel nesting in a lock-free multi-version Software Transactional Memory [*]

Nuno Diegues   Sérgio Fernandes   João Cachopo

INESC-ID Lisboa / Instituto Superior Técnico, Technical University of Lisbon

{nmld, sergio.fernandes, joao.cachopo}@ist.utl.pt

Abstract

Many applications contain large operations that must be performed atomically, which typically leads to many conflicts in optimistic concurrency control mechanisms such as those used by most Transactional Memory (TM) systems. Yet, sometimes these operations could be executed faster if their latent parallelism was used efficiently, but unfortunately few TM systems allow a transaction to be split in several parts that execute concurrently.

In this paper we extend the JVSTM to add support for closed parallel nesting, resulting in the first lock-free, multi-version, and efficient implementation of closed parallel nesting. Moreover, we show how the new implementation with parallel nesting outperforms the original version by up to 10 times in the `Vacation` benchmark, by exploring the latent parallelism within top-level transactions of a highly-contending, write-dominated workload.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming - Parallel Programming

General Terms Algorithms, Transactional Memory, Lock-freedom

Keywords JVSTM, Nested Parallel Transactions, Multi-version

1. Introduction

The rapid proliferation of multicore processors has risen many challenges regarding the synchronization of concurrent programs that attempt to take advantage of this technological evolution. The Transactional Memory (TM) [11] abstraction is meant to simplify that task.

One of the promises of the TM approach is that transactions compose naturally whereas traditional synchronization mechanisms such as locking do not. One way in which a TM may handle this composability is by having transactions spawn nested transactions that may themselves create nested transactions. The emphasis of this work lies in the model that allows more than one transaction to be composed in the same transaction and still run concurrently thus forming a set of sibling transactions with a common parent. As we shall see, this parallel nesting model not only allows more flexibility to the programmer, but may also increase the performance of a program synchronized with a TM system.

The lack of a parallel nesting model in a TM may greatly limit the use of multi-threaded code within a transaction: Depending on the TM in use, the spawn of new threads may not be carefully taken care of by the TM and induce erroneous behavior on the transactional code.

On the other hand, the transactions themselves may represent sequential bottlenecks on the application. If in insufficient number, they may not occupy all the cores of a machine. The fact that the code inside a transaction cannot be multi-threaded may be troublesome for long transactions that could otherwise take advantage of all the available cores in the given scenario.

Moreover, even if we have enough concurrent transactions in the application's typical workload, it may happen that they conflict too much with each other (e.g., due to heavy write-dominated workloads). In these cases, 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 is approximately equal to 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.

Our claim is that the parallelization of those transactions can increase the performance in terms of throughput and latency. In the previous scenario, the solution would be to internally 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 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 split 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 [10]. Above all, we are looking into providing a more flexible TM in which it is possible to parallelize transactions without incurring into excessive overheads that supplant the benefits of the parallelization.

The following sections are organized as follows. Section 2 introduces the current design of the JVSTM, which is the STM that we extended to support the parallel nesting model that we present in Section 3. Then, in Section 4, we present the adaptation of the JVSTM to support parallel nested transactions in a lock-free manner. We present our evaluation in Section 5. Finally, we refer the related work in Section 6 and provide our conclusions in Section 7.

[*] This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and by the RuLAM project (PTDC/EIA-EIA/108240/2008).

2. JVSTM overview

The Java Versioned STM [2] 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 implement 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, created for that sole access if none is active in 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 in the instant they begin, i.e., it is as if they had atomically executed in 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 its execution are still consistent with the current commit-time, i.e., that values have not been changed in the meantime by another concurrent transaction.

Given that transactions mediate access to `VBoxes`, those interceptions are used to record each transactional access in the transaction local log (split between read- and write-sets). Both logs are used at commit time: If the read-set is considered valid, the write-set is written back, producing a new version of the values and effectively publicizing them.

There is a global queue of `ActiveTransactionsRecord` in which transactions enqueue to obtain their order of commit. A transaction only reaches this point if it is validated against all past committed transactions as well as against transactions enqueued before it but not yet committed. After the enqueue, a transaction is guaranteed to commit and that happens with the help of other transactions waiting for their turn to commit, resulting in a lock-free algorithm [2].

This design of the JVSTM followed 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 leaf node. The fact that the nested transaction at the leaf of that tree is guaranteed to be the only one accessing and modifying the read- and write-sets of that nesting tree 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 the main contribution of this work.

3. The Parallel Nesting Model

In this section we specify the model in use adopted from the work of Moss. We use closed nested transactions [3] as the basis for parallel nested transactions. Upon commit, a closed nested transaction merges its read- and write-sets with its parent's. If the closed nested transaction aborts, it may rollback only the atomic action corresponding to itself rather than the whole top-level, depending on the conflict that caused the abort.

The composability in a nested transaction results in accesses to a variable to always obtain the most recent value known by its ancestors. The set of ancestors of a given transaction is composed by itself and its parent's ancestor set. If a transaction has no parent, because it is a top-level transaction, its ancestor set is composed only by itself.

In parallel nesting, a transaction may have multiple nested transactions running concurrently. Two nested transactions are said to be siblings if they have the same direct parent. Each top-level transaction may now unfold a nesting tree with the following characteristics:

- Every node in the tree may have an arbitrary number of children nodes. These are connected by a direct parenthood edge.
- A node performs transactional accesses only when all its children nodes are no longer active.
- The ancestor set is calculated in the same way as for the linear nesting model.

We consider that the need for a parent to execute concurrently with its children may be satisfied by having the parent spawn a nested transaction to execute the following code that belonged to the parent. Therefore, in practice, the parent thread executes a nested transaction encapsulating the parent code. If the parent code spawns any further transactions in the following code, a barrier is placed so that the previously spawned nested transactions finish together with the artificially created nested transaction on the parent.

4. Adding parallel nesting to the JVSTM

To ensure that within a transaction a read-after-write operation of a transactional location returns the correct value—the value that was last written to that location—an STM that defers the writes to commit-time must check first in its write-set if any previous write exists for any location that it is about to read. This is the algorithm implemented in the JVSTM. A simple extension of this behavior to nested transactions, however, means that this check must be done recursively in each ancestor until a write is found or all ancestors have been checked, which means that reads may entail an overhead that is proportional to the nesting depth.

To tackle this problem, in this paper we propose a different design for the JVSTM: 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 by a possible read-after-write when a nested transaction attempts to read a `VBox`: Does any ancestor have a private write entry for that `VBox`?

In fact, the question is more complex than that because more than one ancestor may write to the same `VBox`. Consider the following execution that results in the nesting tree represented in Figure 1

$W_A(x, 5)$ $S_A(B, C, D)$ $W_B(x, 10)$ $W_C(y, 2)$

$S_B(E, F)$ $W_F(x, 15)$ $R_E(x, 10)$

where $W_t(x, y)$ means that transaction t writes the value y to the `VBox` x, $S_t(t_1, t_2, ..., t_n)$ means that transaction t spawns the

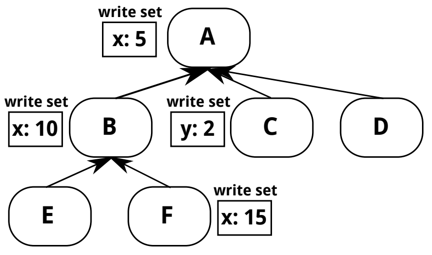


Figure 1: Nesting tree representing an execution and corresponding writes to `VBoxes`: Each node of the tree corresponds to a transaction, and the writes done by each transaction are shown next to it.

nested transactions $t_1, t_2, ..., t_n$, and $R_t(x, y)$ means that transaction t reads the `VBox` x and finds the value y.

The value read by transaction E, corresponding to the value written by B to x, is the only acceptable result for this execution. Reading the value written either by A or by F would not correspond to the expected behavior of such an execution. This means that the read lookup on the ancestors must retrieve the entry owned by the closest ancestor, if any exists, rather than a random one.

In the next Sections we detail the changes that we had to make to the various operations of the JVSTM to support the parallel nesting model that we just overviewed above.

4.1 Managing versions in Nesting Trees

The top-level property of the JVSTM that ensures that read-only transactions never abort may still be preserved in nested transactions. This allows a transaction to be decomposed in several siblings, among which some may be read-only; the read-only siblings are guaranteed to commit despite concurrent commits by their sibling writers. This property relies on the maintenance of versions for nested transactions similarly to what was described for the case of top-level transactions.

In the linear nesting model, which was previously being used by the JVSTM, a nested transaction could always overwrite previous write entries because no other transaction could ever read them. In the new model, however, each transaction may potentially spawn parallel nested transactions. Therefore, at each level of a nesting tree, sibling transactions may concurrently attempt to write to the same `VBoxes`, but these writes must be properly isolated.

To preserve these multiple versions, each transaction now contains a number representing the latest version that has been committed to 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 `ancVersions` set): When a nested transaction is spawned, it computes the union between its parent's current number and its parent's `ancVersions` set to obtain its own `ancVersions` set.

The `ancVersions` set represents the restrictions that the nested transaction has on the view of the nesting tree's write entries that it may read. Consider the previous example whose execution now includes the commit of transactions E, F and B, shown in Figure 2. When the read-only transaction D attempts to read `VBox` x, it must obtain one of the write entries present in its ancestor A's private write-set. In this event, the write entry obtained is restricted by the maximum version that D may ever read on A, given by the mapping contained in its `ancVersions`. In this example, it reads the entry corresponding to version 0. On the other hand, when the read-write transaction C tries to read `VBox` x, it will behave differently. Being a read-write transaction means that its linearization point takes place at the time of commit and not at the time of start, which is what happens for read-only transactions. But, given that C's `ancVersions` is equal to D's `ancVersions` (because they were spawned simultaneously), C can read at most

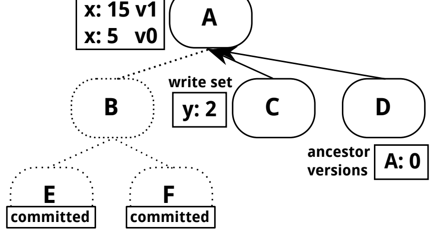


Figure 2: Nesting tree resulting from the commit of transactions B, E and F.

version 0 also. As version 0 is no longer the most recent for `VBox` x on ancestor A, that means that C will not be able to commit, and therefore C fails to read the `VBox` and aborts eagerly.

All of this is possible only if the write entries are reified to contain, not only the tentative value associated to the box, but also other data related to the nested transactions. We will refer to this reification as a `WriteEntry`, which is composed by the transaction that currently owns it, the associated commit version at that nesting tree level, and the value it is recording for write-back.

We explain in detail how the version number on a given transaction is incremented, as commits occur on that level by its children, in Section 4.4.

4.2 Writing to a VBox

As previously mentioned, putting a value in a `VBox` entailed buffering it in the private write-set. In our new design we have to perform an additional step to ensure that we can answer the read-after-write question in a single operation regardless of the nesting depth.

Besides inserting the new tentative value in the private write-set of the transaction, writing to a `VBox` also publicizes this write to all transactions that belong to the same nesting tree. The shared data structure that allows this is the `sharedWriteSet` (a `ConcurrentHashMap`), which, similarly to the private write-sets, allows associating write entries to `VBoxes`. But, whereas the private write-sets of each transaction map a `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.

Recalling the execution prior to nested commits, shown in Figure 1, we now describe the `sharedWriteSet` structure on Figure 3. In this example we have two `VBoxes` written, mapped to their respective write entries. The set of write entries applied to the same `VBox` forms 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 had to be read instead of that one.

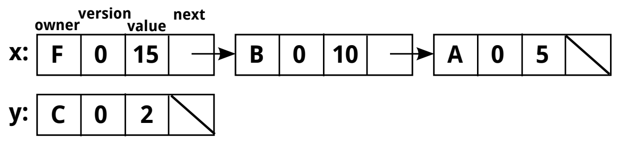


Figure 3: Representation of the `sharedWriteSet` structure for the nesting tree represented in Figure 1.

Instead of having a depth-dependent lookup, we now have a lookup that depends on how many writes contend for the same `VBox` in a given nesting tree. This means that we changed the worst-case complexity of the lookup operation from $O(d)$, where d is the depth of the nesting tree, to $O(n)$, where n is the number of transactions in a nesting tree and $d \leq n$. This worst-case corresponds to a lookup to a VBox x by a nested transaction T where all other transactions in T's nesting tree also wrote to x after T did so. Yet, we claim that in the use case for parallel nested transactions, the sub-transactions that compose it must have somewhat disjoint accesses so that the work may be effectively parallelized. Typically, a transaction does not write arbitrarily to transactional variables without performing some reads. Therefore, it is reasonable to assume that, to parallelize a transaction without having many conflicts among its parallel nested transactions, the nested transaction's read- and write-sets must not intersect that often. This relationship between the read- and write-set leads to

Listing 1: Algorithm to perform a write of a value to a VBox.

```
void setBoxValue(VBox<T> vbox, T value) {
  WriteEntry wE = privWS.get(vbox);
  if (wE != null) {
    // fast path if the transaction already
    // owns a write entry
    wE.value = value;
    return;
  }

  WriteEntry oldFirstWE = sharedWS.get(vbox);
  WriteEntry newWE = makeWE(value, oldFirstWE);
  privWS.put(vbox, newWE);

  // 1st write on this vbox on this tree?
  if (oldFirstWE == null) {
    oldFirstWE = sharedWS.putIfAbsent(vbox,
        newWE);
    if (oldFirstWE == null) { return; }
    else { newWE.next = oldFirstWE; }
  }

  // try to place the new entry on the head
  while(!sharedWS.replace(vbox, oldFirstWE,
      newWE)) {
    oldFirstWE = sharedWS.get(vbox);
    newWE.next = oldFirstWE;
  }
}
```

the conclusion that if the nested transactions do not conflict that much and are, therefore, able to run in parallel efficiently, then each VBox will have very few writes contending on it. Consequently, the length of the linked list of write entries for a given nesting tree is often very short despite the depth of the nesting tree, making the average-case $O(k)$ where $k \ll d, n$. Therefore, the traversal is fast even if there is no previous write to that VBox in this nesting tree that can be read. Moreover, given that the most recent write entry is at the top of the list, a lookup in a read-after-write scenario ends quickly and does not need to search the whole list.

At the start of the method setBoxValue in Listing 1, we test for a write-after-write situation in which the transaction overwrites a value it had previously written. That represents the fast path, which requires only an update on the value of the already existing write entry. If that is not the case, we must create a new write entry containing the aforementioned data (owner, version and value). An additional parameter is added that serves the purpose of connecting the write entries of a given VBox in a linked list as described above.

The insertion of the new write entry is performed at the head of the list of write entries. This is effectively performed by setting the next field of the new write entry to what is expected to be the first entry until the atomic insertion on the ConcurrentHashMap returns positively.

4.3 Reading from a VBox

As hinted in the beginning of Section 4, the get operation on a VBox may return a tentative write entry from the nesting tree buffered in the sharedWriteSet, or it may return a VBoxBody from the globally publicized writes of top-level commits. These two alternatives are explored in the example in Listing 2 where getAncWrite checks for the existence of a read-after-write situation and the readFromBody obtains a globally committed write.

The difference on the design is present in the getAncWrite as we make use of the sharedWriteSet. When iterating over the

Listing 2: Algorithm for retrieving a value of a VBox for a read-write transaction.

```
T getBoxValue(VBox<T> vbox) {
  T value = getAncWrite(vbox);
  if (value == null) {
    value = readFromBody(vbox);
  }
  return value;
}

T getAncWrite(VBox<T> vbox) {
  WriteEntry itWE = sharedWS.get(vbox);
  while (itWE != null) {
    // is the owner an ancestor of mine?
    int ancVersion = getAncVer(itWE.owner);
    if (ancVersion != NOT_ANCESTOR) {
      // can I read that version?
      if (itWE.version <= ancVersion) {
        bodiesRead.put(vbox, itWE);
        return itWE.value;
      } else {
        // eager W-R conflict detection
        // abort up to given ancestor
        throw new Conflict(itWE.owner);
      }
    }
    itWE = itWE.next;
  }
  return null;
}

T readFromBody(VBox<T> vbox) {
  VBoxBody<T> body = vbox.body;
  if (body.version > number) {
    // eager W-R conflict detection
    // abort up to top level
    throw new Conflict();
  }
  bodiesRead.put(vbox, body);
  return body.value;
}
```

write entries performed on the given VBox, the transaction will find entries that belong to other branches of the nesting tree. In other words, some of those write entries will not be owned by an ancestor of the nested transaction and, therefore, cannot be read.

The iteration over the write entries stops when the transaction T attempting the read finds a write entry belonging to an ancestor of T (which includes itself). If no entry is found, the readFromBody is used to fetch a globally committed write. Additionally, the version of the entry cannot be greater than the maximum version that T can read from the ancestor that owns that entry. Recall that this information is stored in the ancVersions set that is created for each transaction upon its start. We essentially collect, in a HashMap, the references to the Transaction objects that represent the ancestors. This way, we can query if a given transaction A is an ancestor of T by computing the hash of the reference of A and checking if the corresponding bucket on the ancVersions set of T contains A's reference. In practice, for small depths it does not pay off to compute the hash and maintain the map over a simple array containing equivalent information and consequent iterations over it to answer queries.

Note that, once the getAncWrite loop reaches a readable write entry, i.e., one that respects the restrictions stated earlier, the algorithm never iterates any further. This is based on the invariant presented for the shared structure representing the nesting tree shared write-sets. The example provided in Listing 2 works for read-write transactions, as it uses eager conflict detection. Conversely, for read-only transactions, we change the behavior slightly: The iteration does not stop necessarily, either successfully or with an abort, if a write entry belonging to an ancestor is found; instead, the iteration stops only when an entry respects both restrictions of ownership and versioning and never aborts because ultimately it reads a `VBoxBody` of a globally committed version.

4.4 Committing Parallel Nested Transactions

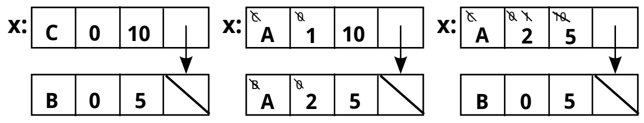
The commit procedure in nested transactions is responsible for validating the execution and propagating into the parent the sets of records, collected by the nested transaction during its execution, effectively publicizing them to the siblings of the committing transaction and serializing it.

Conceptually, the validation requires that the reads performed during the execution (read entries) are still the most recent versions of the `VBox` read. A nested transaction may collect two different types of reads: (1) top-level reads, which are obtained via the `readFromBody` method and return a value of a `VBoxBody` that has been committed by some top-level transaction; and (2) nested reads, which are obtained via the `getAncWrite` method and return a value of a `WriteEntry` that represents a tentative value of an ancestor in a read-after-write fashion. In this sense, both `VBoxBody` and `WriteEntry` represent a value that was written to a `VBox`, but whereas the former is consolidated, the latter is tentative. Therefore the `WriteEntry` extends a `VBoxBody` so that a transaction collects both in the same read-set. Note that for read-only transactions we do not have to perform any validation, neither in top-level nor in nested. But, in the case of nested read-only transactions we have to collect the read-set, if it must be propagated to some read-write ancestor. Only if the nesting tree is entirely composed of read-only transactions may we avoid collecting read-sets as in top-level read-only transactions. Also note that it makes no sense for a read-only transaction to spawn nested read-write transactions.

To validate a nested transaction T we iterate over its read-set. For each read entry r on some `VBox` X we verify that, if T's parent, A, has a write entry w for X, then it must be the r read by T attempting to commit. Otherwise, T fails its validation because r was outdated by w. If w already existed at the time of r then the invariant of the shared structure states that the obtained r would have been exactly w, in which case this validation would have succeeded. This is once again a consequence of the fact that the read lookup has to return the write entry belonging to the closest ancestor, if any exists.

After validation we still have to merge both the read- and write-set into the parent's respective sets. Among the records collected, specifically in the read-set, not all of them have to be propagated. Considering a read entry r, obtained from an ancestor A, r will only have to be validated upon every commit until the commit that merges the records into A (including). Using the execution in Figure 1, if E reads `VBox` X, obtaining the write entry owned by B, then upon E's commit the read entry associated with that access would not have to be propagated to B. On the other hand, if E reads `VBox` Z that had not been written in this nesting tree, the access would result in reading a top-level `VBoxBody` that was already consolidated by a top-level commit. Therefore, this read entry would have to be propagated to B upon E's commit.

Merging the write entries entails updating the owner of the entries as well as its version. To obtain the new version in a commit procedure, nested transactions proceed in a similar fashion to what happens at top-level, by grabbing their commit order on the parent queue of `NestedRecords` rather than on a top-level queue of `ActiveTransactionsRecord`. At the end of this procedure, the write entries that belonged to the committing transaction will now



a) Prior to commits b) Naive merging b) Naive merging

Figure 4: Representation of the `sharedWriteSet` prior to commit, as shown in Figure 1, after commit of that execution using a naive merge procedure, and after commit if using the corrected merge procedure.

be owned by its parent, and have a new version. This version corresponds to the order obtained in the enqueue on the parent.

Now, recall that the structure presented to manage the write entries of a given `VBox` in the nesting tree was associated with the aforementioned invariant: Whenever a transaction iterates over the write entries of a `VBox` looking for a potential read-after-write, as soon as it finds a readable entry, it is guaranteed not to have to look down any further in the list. Consider now the following execution, where $C_T(ok|fail)$ means that transaction T attempts to commit and either succeeds (ok) or aborts $(fail)$:

$S_A(B, C)\ W_B(X, 5)\ W_C(X, 10)\ C_C(ok)\ C_B(ok)$

If we assume the simple merge of write entries in which the committing transaction updates the owner and version of each entry, the result of the execution will break the invariant as shown in Figure 4. Note that the resulting shared structure for that `VBox` X ends up with version 1 on top of version 2. Therefore, if transaction A now attempted to read X, the algorithm would retrieve the value 10 rather than 5.

The underlying issue in the scenario described is that the order in which the concurrent writes are performed is not necessarily the same order in which the transactions commit. Therefore we have to consider the merge in a more careful way. Suppose that T is committing into its parent A. The actual merge of a write entry w for `VBox` X, belonging to the committing transaction T, iterates the list of write entries of X in that nesting tree. The iteration stops when the it finds w, in which case it updates the entry version and owner. However, if the iteration finds an entry w' owned by A, into which the merge will happen, then w' will be conceptually overwritten by w: Any sibling of T that starts after T's commit will always read w instead of w'. If the iteration during the merge of w finds w' before w, we are in the case in which the invariant would be broken. We avoid that by updating the contents of w' to correspond to the merge of w by changing its value to the value of w and the version to the commit version obtained. Note that we need not change the owner as it is already A. In practice, we compare-and-swap (CAS) changes to the version and value held in an indirection object, for reasons that are explained in the next Section.

In the meantime, one could argue that the merging strategy is erasing some versions from the nesting tree, as we are physically overwriting entries that are logically overwritten by commits. If that happens, we may be breaking the semantics of nested read-only transactions which resort to the logically overwritten values (older versions) to ensure that they always commit. However, we can demonstrate that the versions that we physically overwrite in the merge procedure are guaranteed to no longer be read.

Consider once again the execution that led to Figure 4, but now using our corrected merging strategy. Let us assume that some other read-only nested transaction of A, named D, needed the write entry committed by C with value 10 and version 1 on A and could never be satisfied with the write entry committed by B with value 5 and version 2. Then, D must necessarily have version 1 on A. This means that D was spawned after the commit of B but before the commit of C. However, this is a contradiction, because between both commits, A was never in execution, which renders impossible the chance of D being spawned at that time and acquiring start version 1 on A.

4.5 Lock-free commit

So far we have omitted how we actually synchronize the commit of sibling parallel nested transactions to their parent. As hinted before, we have adopted the top-level commit algorithm to nested commits: Each transaction contains a queue of `NestedCommitRecord` which provides a committing order to concurrent siblings. Each committing nested transaction T performs the validation described above and proceeds to obtain its order in the queue by using a CAS to set a new entry on it. If that fails, then some other sibling of T, let us name it S, did so, and therefore T must check that S's write-set does not intersect with T's read-set (in which case T aborts by having failed the incremental validation). Note that by the moment a transaction obtains its place in the parent's queue it is guaranteed to be valid to commit.

The lock-free helping mechanism was also adapted 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, there will always be one committing at a time, even if the underlying threads are allowed to fail silently. This happens because a sibling will always ensure that other siblings that are first in the commit order have already committed successfully before performing its own commit.

Multiple commits (with multiple sibling helpers each) may take place in the same nesting tree concurrently manipulating the same shared structure in a lock-free manner. Consequently, it is relevant that all possible inter-leavings of those executions produce the same result that respects the commit orders obtained: When a helper changes a write entry, a different helper (even if helping another commit) will only change that write entry if it performs the same changes or other changes corresponding to a commit taking place after the first one. This means that our algorithm ensures that the contents of a write entry are guaranteed to never go back to what had resulted from a previous commit. Such situation could be the result of a delayed helper, whose commit had been finished by other helpers, which changed a write entry that had also been changed in the meantime by a newer commit. It is for this reason that we have to CAS changes on the version and/or value of a write entry: A helper only attempts an atomic change if the version of the entry is smaller than the version that it is attempting to commit to.

Note that the writes performed by nested transactions never contend with the commit procedure that changes only already existing write entries. Consequently, the helping procedure of the commit is wait-free: Each helper is guaranteed to finish in a finite number of steps regardless of concurrent actions in the same nesting tree because the operations performed during the helping never have to retry and are bounded by the size of the read- and write-sets to merge.

4.6 Abort procedure

When a transaction aborts, every write that it performed will still be in the `sharedWriteSet`. One could be led to think that this could cause all sorts of problems with transactions reading write entries of a transaction that was no longer active and had actually aborted. However, we guarantee that no transaction T can ever read the write entries of an aborted transaction A, unless T itself is doomed to abort. This happens because for the ownership comparison of write entries, used for the ancestor question raised earlier, we use references to instances of `Transaction`. Consider the following execution: $W_A(X, 1)\ S_A(B)\ R_B(Y, 5)\ C_B(ok)\ C_A(fail)$

As A aborts, its write entries remain in the structure. Suppose now that, when re-executing, A does not write to `VBox` X because its control flow was different. Yet, in practice, the write of A to X is still in the nesting tree structure. However, our point is that when B attempts to read it, it will fail to do so, because the owner of that entry is some other A that is not the A that spawned B now. Moreover, we are guaranteed that no `ABA` problem can ever happen in which the address of the old A would be released and reused in some different transaction which would get this ghostly write entry out of the blue. The reason we have such guarantee is due to the underlying garbage collector provided by the Java runtime: As long as one of those aborted write entries exists, the aborted transaction is still referenced and therefore its reference address can never be used by some other transaction.

Due to memory usage concerns, we actually clean the aborted transactions to allow the garbage collector to clear most of the data, such as the read-set. Regarding the write entries, we attempt a single physical delete from the lock-free linked list that is associated with the `VBox` of the write entry w in the shared structure of the nesting tree. To do so, we find the entry b in the list that is placed before w and change its next pointer to point to the next of w. There is no need for a compare and swap as the only transaction that is ever going to change the next b, is the transaction that owns w. It may happen that b now points to another w' (that was the next of w) that was concurrently deleted by another aborting transaction. Consequently this action actually reverts that delete and, instead of having 2 successful concurrent deletes, we end up with only 1 actual physical delete. However, as shown, having aborted entries in the list is guaranteed to never harm correctness. Moreover, these structures are private to each top-level transaction (and its nested transactions), meaning that upon its completion, they are all discarded. This results in a best-effort strategy of saving memory while remaining safely correct.

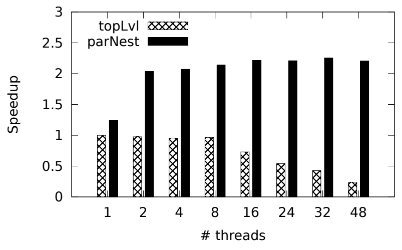
5. Evaluation

To assess the behavior of the new parallel nesting algorithm, we conducted a series of tests on the `Vacation` application of the STAMP benchmark [12]. In `Vacation` a set of customers concurrently makes requests that affect some of the application's resources (such as flights and cars): Each request from a customer is composed of operations—such as reserving a car for the customer, or canceling some reservation—that must be performed atomically.

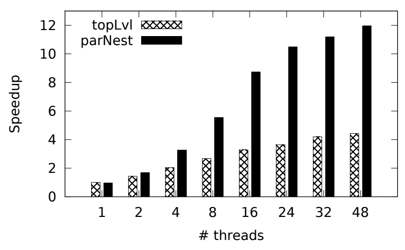
In practice, the `Vacation` application consists of a series of top-level transactions that correspond to each of the requests. Because all the operations that take place in a request include at least some transactional writes, this application does not take advantage of read-only transactions that always commit in the JVSTM. On the other hand, in scenarios of high contention, we expect that parallelizing top-level transactions internally and running less top-level transactions at a time delivers better performance. As we shall see, this is possible in this application (but not in all) because these requests have some latent parallelism.

To control the level of contention in the benchmark, `Vacation` has parameters that allow us to configure the number of customer atomic requests and inner operations to execute, how many threads should be performing the requests concurrently, and how much they should be contending for the same data. Additionally, we may parameterize the application so that the executed requests respect a given probability distribution. To explore the latent parallelism in the application, we changed its three requests as follows:

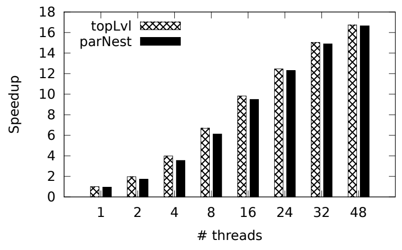
- `MakeReservation`: This is the most dominant request in the application. It starts with a bulk of traversals across the underlying data performing only reads. It is where the operation spends most of its time. According to the data that it collects, it may perform some writes in the end (which often does). We added



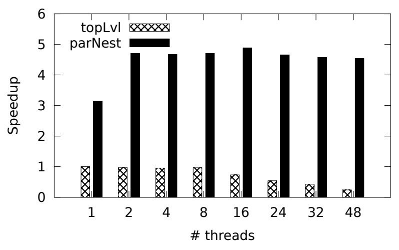
(a) Only 5% of reservations.



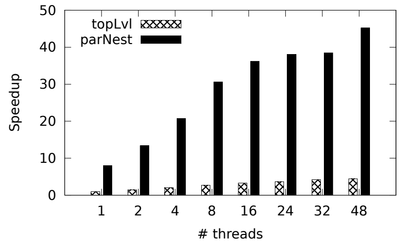
(b) 98% of reservations and high contention.



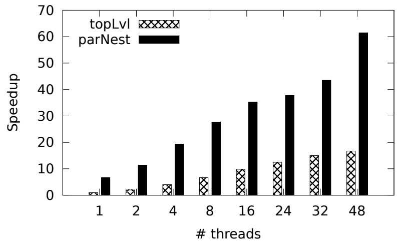
(c) 98% reservations and low contention.



(d) Only 5% of reservations.



(e) 98% of reservations and high contention.



(f) 98% reservations and low contention.

Figure 5: Comparison between the use of only top-level transactions (`topLvl`) against parallel nested transactions (`parNested`) with an increasing number of threads. The speedup is relative to the sequential run of the top-level version. In Figures 5a, 5b, and 5c, `parNested` uses a `VBox` on each node of the red-black trees, whereas in Figures 5d, 5e, and 5f, it uses a single `VBox` to hold each red-black tree.

parallel nested transactions that performed the traversals concurrently (split among them).

- `DeleteCustomer`: This request deletes a given customer and all its reservations. The cancellation of his reservations is split among parallel nested transactions.
- `UpdateTables`: This request either adds or removes resources from the underlying data structures that maintain them available for the application. The updates are distributed among parallel nested transactions that perform the changes concurrently.

The application holds the resources in red-black trees, one for each type of resource, including the customers themselves. On the other hand, each customer contains a list that points to each resource that it reserved. To be able to use the JVSTM, we adapted the benchmark by encapsulating the mutable shared fields in `VBoxes`. We refer to this process as transactification.

We present the results of our tests in Figure 5, where we vary the number of available threads and plot the speedup of each configuration relative to the sequential execution using top-level transactions. In these tests we are comparing the execution when using only top-level transactions against our new approach of parallel nesting described above (the underlying STM is the same design of the JVSTM in both cases). Each result is the average of three executions. These tests were executed on a machine with four AMD Opteron 6168 processors (48 cores total) and 128GB of RAM (using at most 4GB), running Red Hat Enterprise 6.1 and Oracle's JVM 1.6.0_24. We required 96 requests to be performed with 480k operations each.

In Figure 5a, we used a workload with very few reservations, therefore mostly inserting and removing resources and clients. The top-level transactions contend significantly because of the structural modifications on the trees, which results in slowdown as the thread count increases, due to the overhead of the JVSTM aborting and restarting most of the transactions. On the other hand, using parallel nested transactions and a single top-level transaction at a time, we see an increase in performance, but only from one to two threads for two reasons. First, because there are only three tables, concurrent operations conflict whenever we have more than three siblings. Second, because deleting a customer is a very lightweight request that does not benefit from its parallelization, as each customer has typically very few reservations. For these reasons, the speedup slightly decreases as the thread count increases. Note, however, that the top-level approach decreases its performance at a much greater speed.

Figures 5b and 5c both depict the normal case of a high amount of reservations. The former with high contention and the latter with low contention. In the low contention scenario both alternatives behave similarly: The overheads of parallel nesting are visible averaging a 4% loss in performance. In the high contention scenario, however, top-level transactions barely scale as the thread count increases; we are in the case described in the Introduction, where top-level transactions conflict with high probability and are unable to scale properly. Replacing the top-level transactions with the parallel nested approach yields much better results: At 48 threads we have a speedup of 12x and an increase in performance of 270% with regard to the top-level approach. Therefore, as we initially pointed out, this approach is more appropriate for the case in which there is high contention and the parallelization of the operations is effective.

In these experiments we used a single top-level transaction in the parallel nesting approach as the siblings being spawned were enough to occupy the available processors. But, if that was not the case, we could run more top-level transactions concurrently: There is no restriction imposing the parallel nesting approach to run only a single top-level transaction at a time.

Given this context, in which we are able to use all the available resources with a single top-level transaction parallelized, we noticed that we could increase the performance of the parallel nesting version in this benchmark even further. In the results already described we used a `VBox` on each node of the red-black trees (finely-grained transactified). This allowed top-level transactions to perform requests whose operations contended on the same type of resources without false conflicts arising. This has an inherent cost, because finding a node in the finely-grained transactified red-black tree requires opening $O(log(n))$ `VBoxes` for a tree containing n nodes. Both approaches incurred in this cost in the results shown in Figures 5a, 5b, and 5c. The alternative of encapsulating the whole red-black tree in a `VBox` causes all concurrent top-level transactions to conflict with each other, as they typically read and write some items in all types of resources. Yet, this is not the case if we use our approach with parallel nesting, because we run only one top-level transaction at a time. Thus, by using parallel nesting, we have the added benefit of having more lightweight red-black trees encapsulated in `VBoxes` rather than fully transactified red-black trees. The experiments with parallel nesting using a coarse-grained transactified red-black tree are shown in Figures 5d, 5e, and 5f. The benefit of using a more lightweight red-black tree in the parallel nesting approach is visible in the fact that it outperforms the top-level approach already at one thread only. Replacing the top-level transactions with the parallel nested approach in this case is, on average, approximately 10 times faster with high contention and 5 times faster with low contention.

6. Related Work

The CWSTM [8], 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 [9] 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 [6], which was built on top of OpenMP and Intel's STM to integrate parallel and atomic blocks. NePalTM uses an abstract lock [7] 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 whereas 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 shallow nested transactions.

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 [5] 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 and are intensively used for conflict detection. Finally, the PNSTM [1] 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 is lock-based and single-version, whereas the implementation that we describe in this paper is a lock-free algorithm with multi-version support, thus allowing read-only transactions to always commit.

7. Conclusion and Future Work

In this paper we presented an extension to the JVSTM lock-free algorithm that adds support for parallel nested transactions, while maintaining the properties of read-only transactions. Using this new algorithm on a well-known benchmark, we showed that parallel nesting may allow us to uncover latent parallelism in some applications, leading to significant speedups of up to 10 times. A key observation from these tests 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.

Despite the positive results, we are aware of some overheads inherent to our extension to the JVSTM core algorithms, which in the results presented were overwhelmed by the parallelization in the benchmark. As a result, in this ongoing work, we will attempt to make the spawn and commit of a parallel nested transaction more lightweight, so that we may apply it to a wider variety of applications.

References

[1] J. Barreto, A. Dragojevic, P. Ferreira, R. Guerraoui and M. Kapalka. Leveraging parallel nesting in transactional memory. In *Proc. of the* 15^{th} *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '10), 91–100*, 2010.

[2] S. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proc. of the* 16^{th} *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11), 179–188*, 2011.

[3] J. Moss and A. Hosking. Nested Transactional Memory: Model and Architecture Sketches. In *Science of Computer Programming, 63(2), 186–201*, 2006.

[4] W. Baek and C. Kozyrakis. NesTM: Implementing and Evaluating Nested Parallelism in Software Transactional Memory. In *Proc. of the* 9^{th} *Conference on Parallel Architectures and Compilation Techniques (PACT '10), 253–262*, 2010.

[5] R. Kumar and K. Vidyasankar. HParSTM: A Hierarchy-based STM Protocol for Supporting Nested Parallelism. In *the* 6^{th} *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '11)*, 2011.

[6] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian and R. Narayanaswamy. NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *Proc. of the* 23^{rd} *European Conference on Object-Oriented Programming (ECOOP '09), 123–148*, 2009.

[7] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. Hosking, R. Hudson, J. Moss, B. Saha and T. Shpeisman. Open nesting in software transactional memory. In *Proc. of the* 12^{th} *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '07), 68–78*, 2007.

[8] K. Agrawal, J. T. Fineman and J. Sukha. Nested parallelism in transactional memory. In *Proc. of the* 13^{th} *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '08), 163–174*, 2008.

[9] H. Ramadan and E. Witchel. The xfork in the road to coordinated sibling transactions. In *the* 4^{th} *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '09)*, 2009.

[10] I. Anjo and J. Cachopo. Jaspex: Speculative parallel execution of Java applications. In *1st INFORUM*, Faculdade de Ciências da Universidade de Lisboa, 2009.

[11] M. Herlihy, J. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the* 20^{th} *Annual International Symposium on Computer Architecture (ISCA '93), 289–300*, 1993.

[12] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of The IEEE International Symposium on Workload Characterization (IISWC '08), 35–46*, 2008.