Lock-free and Scalable Multi-Version Software Transactional Memory *

Sérgio Miguel Fernandes João Cachopo
IST / INESC-ID
{Sergio.Fernandes,Joao.Cachopo}@ist.utl.pt

Abstract

Software Transactional Memory (STM) was initially proposed as a lock-free mechanism for concurrency control. Early implementations had efficiency limitations, and soon obstruction-free proposals appeared, to tackle this problem, often simplifying STM implementation. Today, most of the modern and top-performing STMs use blocking designs, relying on locks to ensure an atomic commit operation. This approach has revealed better in practice, in part due to its simplicity. Yet, it may have scalability problems when we move into many-core computers, requiring fine-tuning and careful programming to avoid contention.

In this paper we present and discuss the modifications we made to a lock-based multi-version STM in Java, to turn it into a lock-free implementation that we have tested to scale at least up to 192 cores, and which provides results that compete with, and sometimes exceed, some of today's top-performing lock-based implementations.

The new lock-free commit algorithm allows write transactions to proceed in parallel, by allowing them to run their validation phase independently of each other, and by resorting to helping from threads that would otherwise be waiting to commit, during the write-back phase. We also present a new garbage collection algorithm to dispose of old unused object versions that allows for asynchronous identification of unnecessary versions, which minimizes its interference with the rest of the transactional system.

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

General Terms Algorithms, Transactions, Scalability

Keywords Transactional Memory, Multi-Version Concurrency Control, Lock-Free Synchronization, Garbage Collection

1. Introduction

Since the seminal work on Transactional Memory [12] and Software Transactional Memory (STM) [19], there has been a boom in related research, which has led to several proposals for STM implementations, each with their own set of characteristics. One such implementation is the Java Versioned Software Transactional Memory (JVSTM) [3], which was specifically designed to optimize the execution of read-only transactions. In the JVSTM, read-only transactions have very low overheads, and they never contend against any other transaction. In fact, read-only transactions are wait-free [11] in the JVSTM.

JVSTM's design goals stem from the observation and development of real-world domain-intensive web applications. These applications have rich and complex domains, both structurally and behaviorally, leading on one hand to very large transactions, but having also a very high read/write ratio. These characteristics have been observed over several years of use of the JVSTM in the FénixEDU project in a production environment [4]. This web application executes concurrent transactions to process the user requests, and the number of read-only transactions represents, on average, over 95% of the total number of transactions executed.

The design of the JVSTM allows it to excel in read-intensive workloads [2], but raises doubts on its applicability to other types of workloads, where writes dominate. Even though during the execution of a write-transaction reads and writes of transactional locations are wait-free, the commit of these transactions serialize on a global lock, thereby having the potential of impairing severely the scalability of a write-intensive application.

In this paper we address this problem by presenting and discussing the implementation of a new scalable and efficient commit algorithm for the JVSTM that is able to maintain the exact same properties for reads, as before. This new commit algorithm allows commits to proceed in parallel during the validation phase, and resorts to helping from threads that would otherwise be waiting to commit, during the write-back phase. We evaluate the new algorithm with four benchmarking applications, and show it to scale well up to 192 parallel transactions, when tested on a machine with a little over 200 cores.

In the following section we give an overview of the key aspects of the JVSTM that are relevant to understand the new commit algorithm. Then, in Section 3 we describe the new algorithm, followed by a description, in Section 4, of the asynchronous garbage collection mechanism for old versions, which we developed to address a scalability problem in the existing code. In Section 5 we present a performance evaluation. Section 6 provides a discussion on properties of the algorithms and related work. We conclude in section 7.

2. JVSTM overview

JVSTM is a word-based STM that supports transactions using a Multi-Version Concurrency Control (MVCC) method. Each transactional location uses a Versioned Box (`VBox`) [3] to hold the history of values for that location, as exemplified in Figure 1. A `VBox` instance represents the identity of the transactional location, and

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

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

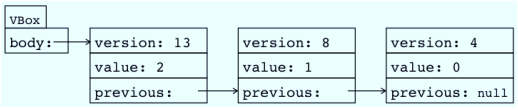

Figure 1. A transactional counter and its versions.

contains a `body` with the list of versions. Each element (`VBoxBody`) in the history contains a version number, the value for that version, and a reference to the previous state of that versioned box.

A transaction reads values in a version that corresponds to the most recent version that existed when the transaction began. Thus, reads are always consistent and read-only transactions never conflict with any other, being serialized in the instant they begin, i.e., it is as if they had atomically executed in that instant. Conversely, write-only transactions are serialized at commit-time (when the changes are atomically made visible) and, thus, they never conflict as well. Read-write transactions (write transactions for short) require validation at commit-time to ensure that values read during the transaction are still consistent with the current commit-time, i.e., values have not been changed in the meanwhile by another concurrent transaction.

During a transaction, the JVSTM records each transactional access (either a read or a write) in the transaction's local log (in the read-set or in the write-set, respectively). During commit, if the transaction's read-set is valid, then its write-set is written-back, producing a new version of the values, which will be available to other transactions that begin afterwards. The following pseudo-code in Figure 2 summarizes the commit algorithm of write transactions (the commit of read-only transactions simply returns).

The global lock provides mutual exclusion among all committing write transactions, which ensures atomicity between validating the read-set and writing-back the write-set. Also, the version number is provided by a unique global counter that changes only inside the critical region. The commit operation sets a linearization point when it updates the global counter. After that point, the changes made by the transaction are visible to other transactions that start. When a new transaction starts, it reads that number to know which version it will use to read values.

Also, observe that a read from a transactional location is usually very fast, for the following reasons: (1) No synchronization mechanism is required to read from a `VBox`. The only (lock-free) synchronization point occurs at the start of the transaction, when it reads the most recent committed version number; (2) the list of versions is always ordered by decreasing version number. The commit operation keeps the list ordered, because it always writes to the head of the list a version number that is higher than the previous head version; and (3) the required version tends to be at the head of the list, or very near to it. If it is not at the head, then it is because after this transaction started, another transaction has already committed and written a new value to that same location (recall the expected low number of write transactions, and that a transaction always starts in the most recent version available).

```
commit() {
  GLOBAL_LOCK.lock();
  try {
    if (validate()) {
      int newTxnumber = globalCounter + 1;
      writeBack(newTxNumber);
      updateGlobalCounter(newTxNumber);
    }
  } finally { GLOBAL_LOCK.unlock(); }
}
```

Figure 2. The global lock-based commit algorithm.

2.1 Deletion of old versions

To keep the list of versions in each `VBox` from growing indefinitely, versions are kept only for as long as any transaction may require them. JVSTM ensures forward progress and transactions always start from the most recent version available. As older transactions finish (or restart due to conflict), older versions may become unused, because they become logically inaccessible, and should be removed. When a write transaction commits, all values that it writes replace logically the older values, from the point of view of transactions that start henceforth. Thus, these older values can be removed, if there is no transaction running in a version older than the version just committed.

To identify which versions are no longer accessible and to manage garbage collection of old versions the JVSTM uses the *Active Transactions Record* [2]. This structure holds a list of transaction records deemed to be *active*, in a singly-linked list of records, with each record holding a reference to the next most recent record. Each record keeps information about a write transaction that already committed, namely the commit version and the values that were written (as references to the `VBoxBody` instances created during the commit). A record also has a counter of how many transactions are running in the record's version. When a new transaction starts, it atomically increments the counter on the most recent record. When a transaction finishes, it decrements the same counter and checks if it has reached zero. A record is considered active if the counter is positive or there is an older record that is active.

When a record becomes inactive and there is already a newer record[1], versions produced by the newer record make older versions inaccessible. At this point, older versions are removed by setting the field `previous` in each `VBoxBody` in the newer record to `null`. This process is dubbed "cleaning the record". Notice that a version v_1 is never lost, unless there is a transaction that writes a newer version v_2 to the same `VBox`. In such case, v_1 may be removed, only if the record referring to it becomes inactive.

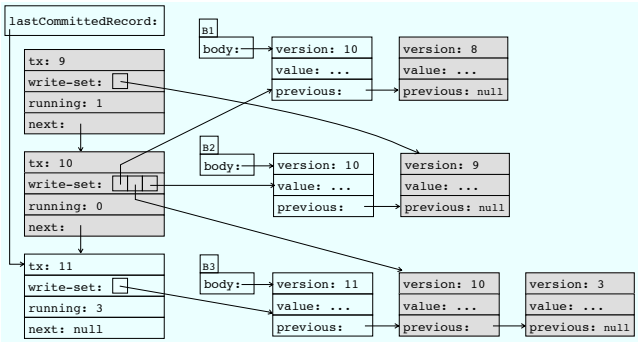

Figure 3. An example snapshot of the Active Transactions Record.

Figure 3 provides an example scenario, in which there are three active records representing the commits of versions 9, 10, and 11. There are also three transactional locations managed by `VBoxes` B1, B2, and B3. Transaction 9 wrote to B1, transaction 10 wrote to all three locations, and transaction 11 (the most recent write transaction to commit) wrote to B3. Currently, there are four running transactions: One that started in version 9, and three that started in version 11. When the transaction running in version 9 finishes, it will decrement the `running` counter to zero. After this occurs, record 9 is identified as inactive, and record 10 is cleaned. This is accomplished by setting to `null` the `previous` field in all `VBoxBodies` with

[1] A newer record ensures that new transactions will start in a more recent version and not re-activate this record.

version number 10. Now, record 10 also is identified as inactive, and record 11 is cleaned. After both records are cleaned, the objects shown in gray can be garbage collected.

2.2 The need for a new commit algorithm

In many cases, the time spent by a write transaction in the critical region at commit time is very short when compared to the time it takes to actually execute an entire transaction. Moreover, the JVSTM was initially developed to support read-heavy applications, in which the number of read-only transactions corresponds to more than 95% of the total number of transactions executed. The other 5% seldom overlap in time, even more so, if we only consider the time of the commit operation. These facts put together justified the use of the global-lock approach, with good performance results. Nevertheless, there are reasons to implement an alternate solution.

The rate of execution of write transactions affects performance. Locks do not scale. Different applications have different workloads regarding read and write operations and may execute many write transactions in a short period of time, especially in the case of CPU-intensive applications. This may highly increase the probability of having more than one write transaction trying to commit concurrently. Whereas this may not be an issue for machines with a small number of cores, many-core machines are an emerging reality and the number of cores available at an affordable price is growing. Thus, it is reasonable to assume that in a many-core machine the high contention on the global lock will degrade performance significantly.

Contention in multi-processing increases the probability of restarts due to conflicts, and reduces the overall throughput. It is easy to understand that when more transactions run concurrently, the probability of conflicts increase. But it is not so obvious that this situation may be aggravated when running N transactions in less that N processors.

As we saw, a write transaction has a conflict if some value that it read was already changed when it tries to commit. So, intuitively, the probability of conflict increases with the length of the transaction and the number of concurrent transactions, but it should not depend on the number of available processors, because having less processors should slow down all transactions proportionally and, thus, maintain the probability of having transactions committing during the execution of another transaction. Yet, this intuition is valid only if all transactions proceed without interference. Once a transaction may have to wait for a lock, things change.

The JVSTM's commit operation uses a fair locking policy. This means that each transaction will obtain the lock in the order that it was requested. When a write transaction (T_1) commits, it tries to get the lock. If another transaction (T_0) already has it, then T_1 has to wait. The more transactions there are, then the less processor time T_0 will have to finish and release the lock. Thus, there is a higher probability that other write transactions ($T_2, \ldots, T_n$) will queue up for the lock as well, because they will be given processor time to execute, whereas T_1 is still waiting. Suppose that when T_1 finally gets the lock, it fails validation, releases the lock, and restarts. Releasing the lock and restarting is generally much faster than committing T_2, thus T_1 will typically restart with the version number of T_0's commit (the most recent at the time of restart). So, in practice, the length of T_1, during which other transactions may commit and therefore conflict with T_1, has effectively increased from its normal executing length by the amount of time that it had to wait for the lock.

This occurs because the re-execution of T_1, on average, will see more transactions committing than its first execution. It is not certain that using an unfair lock would reduce the probability of restarts, simply because then $T_2, \ldots, T_n$ would still be able to get ahead of T_1 in the race to commit.

3. Lock-free commit algorithm

In this section we describe a new commit algorithm that we developed to replace the existing lock-based implementation. Conceptually, the algorithm contains the same steps as before: (1) validate the read-set; (2) write-back the write-set; and (3) make the commit visible to other transactions. In the following sections we explain how each step is accomplished in the new algorithm.[2]

3.1 Validation

In the lock-based commit algorithm, validation explicitly checks that the most recent version of each value in the read-set is still the same version that was read during the transaction (*snapshot validation*). The insight is that validation can also be performed incrementally by checking the write-sets of all transactions that have committed versions greater than the one this transaction started with (*delta validation*). Suppose that transaction T started in version v_1. For any transaction T_i that committed a version greater than v_1, if any element in the write-set of T_i is in the read-set of T, then T cannot commit, because of a conflict.

Of course that, without exclusion in the validation step, while T is validating itself, other transactions can also be validating themselves to commit as well. We need to order the commits in such a way that each transaction can finish validation and be sure that it is valid to commit.

To do so, we use and extend the functionality of the Active Transactions Record that already exists in the JVSTM. The Active Transactions Record holds a list of transaction records, each containing information about a write transaction that has already committed. To support the delta validation, we extend this list to also include valid, but not yet fully committed records. Thus, a transaction that can get an entry in this list effectively establishes its commit order (and, as such, its commit version). The code depicted in Figure 4 shows how a transaction concurrently validates its read-set in a lock-free way.

```
validateCommitAndEnqueue() {
  ActiveTxRecord lastValid = this.activeTxRecord;
  do {
    lastValid = validate(lastValid);
    this.commitTxRecord = new ActiveTxRecord(lastValid.txNumber+1,
                                             this.writeSet);
  } while (!lastValid.trySetNext(this.commitTxRecord));
}
```

Figure 4. The lock-free validation algorithm.

Each write transaction has two transaction records: The `activeTxRecord`, which points to the record that represents the version in which the transaction started; and, the `commitTxRecord`, which is the record that is created to represent the transaction's own commit. To be valid, a transaction needs to check for an empty intersection between its read-set and the write-set of each record, from the `activeTxRecord` onward. This is what the `validate` method does: It checks all records from the `lastValid` onward and it returns the last successfully validated record (or throws a `CommitException` if validation fails at any point). Next, the new `commitTxRecord` is created with an incremented commit version number, and the transaction's write-set. The `trySetNext` is a *compare-and-set* operation (CAS) that atomically sets the `commitTxRecord` as the next valid record after the `lastValid`. This is a tentative operation that

[2] For clarity, we omit minor code details, and simplify syntax. The full code is available at http://groups.ist.utl.pt/esw-inesc-id/git/jvstm.git.

only succeeds if the next record is still unset. Otherwise, this means that another transaction has won the race for that position, in which case validation resumes from the last known valid record. Thus, commit order is defined by the order in which transaction records enter the Active Transactions Record list. All transactions that are in the process of committing must obtain a position in this list, after validating themselves. Even if not all transactions are already written-back, being in this list enables future committing transactions to check their validity against all those that are already queued for a sure commit.

Notice that this validation algorithm is lock-free, because even though a single transaction may continuously fail to get a commit position, this implies that the transactional system as a whole must be making progress, i.e., other transactions are in fact validating and queueing their commit records. Additionally, if no commits occur between the start and commit of a transaction, then validation is not necessary, because that transaction's `activeTxRecord` is the most recent one and is already valid, so the committing transaction only needs to queue up its commit record.

3.2 Write-back

After a successful validation, a transaction obtained a commit version, which is represented by the commit record that was placed in the commit queue. Figure 5, shows an example of a possible state of the Active Transactions Record. Transactions 9 and 10 have already committed[3], whereas transactions 11 to 13 are valid but not yet written-back. At this time, any new transaction, will start in version 10, because it is the last committed version. We also show the two transaction records in use by transaction 12 (thus, assuming that transaction 12 started before transaction 10 committed).

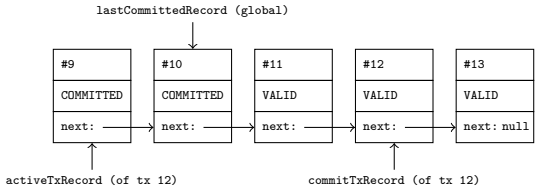

Figure 5. The Active Transactions Record is extended to hold write transactions that are valid but not yet written-back.

Writing-back consists in adding a new `VBoxBody` to the corresponding `VBox`, for each value in the write-set of the transaction, with the version obtained during validation. Taking into account that each value is written to a different location, each write can be done in parallel. One could even consider the concurrency between any two valid transactions that need to write-back values to different locations, but this is not so simple: Within each `VBox` we need to maintain the list of versions ordered, to keep the reads fast. If two valid transactions try to commit different values to the same location, then they need to be ordered according to their commit version. This poses a difficulty, because it is expensive to compute whether the write-sets of valid transactions intersect. This problem did not exist in the lock-based version, simply because only one transaction could be in the write-back phase, at any given time.

The easiest way to ensure that write-backs to the same location are performed in order is to write-back only one transaction at a time, beginning with the oldest valid transaction that is yet uncommitted. Each valid transaction will help the oldest valid transaction to write-back, up to, and including, itself. This way, instead of blocking, transactions waiting to commit can be helpful and increase the overall write-back throughput. This behavior is shown in the code depicted in Figure 6.

[3] The commit status of a write transaction is given by its commit record.

```
// in class Transaction:
ensureCommitStatus() {
  ActiveTxRecord recToCommit = lastCommittedRecord.getNext();
  while (recToCommit.txNumber <= this.commitTxRecord.txNumber) {
    if (!recToCommit.isCommitted()) {
      WriteSet writeSet = recToCommit.getWriteSet();
      writeSet.helpWriteBack(recToCommit.txNumber);
      finishCommit(recToCommit);
    }
    recToCommit = recToCommit.getNext();
  }
}

// in class WriteSet:
helpWriteBack(int newTxNumber) {
  int finalBucket = random.nextInt(this.nBuckets);
  int currBucket = finalBucket;
  do {
    if (!this.bucketsDone[currBucket].get()) {
      this.bodiesPerBucket[currBucket] =
                    writeBackBucket(currBucket, newTxNumber);
      this.bucketsDone[currBucket].set(true);
    }
    currBucket = (currBucket + 1) % this.nBuckets;
  } while (currBucket != finalBucket);
}

List writeBackBucket(int bucket, int newTxNumber) {
  List newBodies = new LinkedList();
  for (BucketEntry be : this.buckets[bucket]) {
    VBoxBody newBody = be.vbox.commit(be.value, newTxNumber);
    newBodies.add(newBody);
  }
  return newBodies;
}

// in class VBox:
VBoxBody commit(Object newValue, int txNumber) {
  VBoxBody currHead = this.body;
  VBoxBody existingBody = currHead.getBody(txNumber);

  if (existingBody == null) {
    VBoxBody newBody = new VBoxBody(newValue, txNumber, currHead);
    existingBody = CASbody(currHead, newBody);
  }
  return existingBody;
}

VBoxBody CASbody(VBoxBody expected, VBoxBody newBody) {
  if (compareAndSwapObject(this, VBox.bodyOffset, expected,
                           newBody)) {
    return newBody;
  } else {
    return this.body.getBody(newBody.version);
  }
}
```

Figure 6. Algorithm for concurrent write-back with support for helping from other committing transactions.

After validation, transactions run the `ensureCommitStatus` method. This method starts by accessing the globally available `lastCommittedRecord` to get the next record to commit. After each iteration, it is guaranteed that the transaction corresponding to `recToCommit` has been committed. To share the work among concurrent helping transactions we split the values to write-back into several buckets, of approximately the same size each. The write-set itself holds the buckets, so `ensureCommitStatus` delegates the write-back operation to the write-set, configuring it with the version number to use.

The `helpWriteBack` method picks a random[4] bucket to start, and then iterates through each bucket once. For each bucket, it checks whether that bucket is already written-back, by checking

[4] There is one instance of the randomizer class per thread, to eliminate contention on the random number generator.

the bucketsDone array. This array contains an AtomicBoolean for each bucket that indicates whether the values in the corresponding bucket are already written back. Any helping transaction may find that some buckets are already written-back, if some other transaction has helped. For those buckets that are not yet written-back, this helping transaction will attempt to write each value to its respective VBox, and mark the bucket as done. The writeBackBucket method, iterates through the elements of the bucket invoking commit on each VBox with the value and version to commit.

The commit method handles the creation of the new VBoxBody and inserts it at the head of the list of bodies. Due to concurrency in the write-back operation, more than one thread can attempt to write-back to the same VBox. As such, the commit operation starts by searching (via the getBody method) whether there is already a VBoxBody with the required version. If it does not exist, then the new VBoxBody is created and CASBody attempts to install it. This operation tries to replace the VBox's current body with the new one with a single CAS attempt. If it succeeds, it returns the new head of the list. It if fails, then it is because a concurrent helping transaction succeeded, in which case it looks up the required VBoxBody to return. So, commit always returns the written-back VBoxBody. These bodies are collected in the newBodies list that is stored in the bodiesPerBucket array in the position corresponding to the bucket being written-back. In Section 4 we will explain how these lists are used by the garbage collection algorithm to eliminate old versions.

Finishing a commit requires publicizing the changes globally, and it can be done only after all values are in place. According to the Java Memory Model [14], any write to memory performed before a volatile write of a given location x will necessarily be seen by other threads, as long as they first perform a volatile read of x. The setting of each AtomicBoolean in the bucketsDone array entails volatile write semantics, thus when another thread sees that a bucket is done (issuing a volatile read), it will surely be able to see the values of that bucket written-back.

At the end of helpWriteBack the helper transaction can be sure that all buckets of this write-set are written-back, because either this transaction did it or checked that another transaction did it. So, the helper is sure that the helped transaction can be safely committed.

The write-back phase is wait-free, because any transaction that enters this phase is certain to finish it in a bounded number of steps, in spite of other threads executing any other code concurrently. The number of iterations performed in ensureCommitStatus is clearly bounded, because there is a finite number of records between the lastCommittedRecord and the transaction's own commitTxRecord, which was already enqueued in the validation phase. The number of buckets is fixed per write-set and the helpWriteBack method only iterates them once. The CAS to set the new VBoxBody may fail, but in that scenario the written-back value is already there, so the operation does not have to be retried. Also, when the CAS fails, finding the written-back value to return is a limited search, because the list of versions only grows at the head, and the search is performed in the opposite direction.

3.3 Finishing a commit

All transactions that help to write-back another have to try to finish that transaction, because no transaction really knows whether it was the first one to fully complete the write-back. Nevertheless, after the write-back operation is completely finished, making the changes globally visible is a trivial operation with two steps: (1) set the commit flag in the commit record; and, (2) set the global lastCommittedRecord to be the record just committed. This is done in the finishCommit method, as shown in Figure 7.

In this new version of the commit algorithm, the linearization point is step (1), which sets a volatile boolean variable to true. Updating the global reference to the last committed record is just a helping operation to reduce the search effort of new transactions. The finishCommit operation has no synchronization, so it is possible that thread interleaving causes some late thread to actually set the lastCommittedRecord reference back to a previously committed record, when there is already a more recent committed record. For this reason, whenever a new transaction begins, it obtains the most recent version from the lastCommittedRecord, and then it iterates forward, looking for the most recent committed record. Recall that a record enters the list immediately after validation succeeds, whereas in the lock-based algorithm, the list was only updated in mutual exclusion, after each successful commit.

```
finishCommit(ActiveTxRecord recToCommit) {
  recToCommit.setCommitted();
  lastCommittedRecord = recToCommit;
}
```

Figure 7. Finishing a commit and publishing changes.

3.4 Validation revisited

As mentioned in Section 3.1, validation of a transaction T can be performed in two different ways: (1) A snapshot validation atomically checks if T's read-set is still up to date with the current global state (this corresponds to the original validation used in the lock-based commit); and (2) a delta validation incrementally checks each write-set committed since T started and looks for an intersection with T's read-set (the validation that we introduced in the new commit algorithm).

Either validation technique has advantages and shortcomings, performance-wise. Snapshot validation depends solely on the size of the read-set. The time it takes to validate a read set is proportional to its size. Conversely, delta validation is independent of the size of the read-set. This holds true as long as the lookup function that searches whether any given element is contained in the read-set can be executed in constant time. Delta validation depends on the size and number of all the other write-sets committed while the transaction executed.

From our experience, read-sets are, on average, several times larger than write-sets, although this difference is highly dependent on each application's read and write patterns. So, delta validation will tend to perform better when the average size of a read-set to validate is greater than the average size of the list of write-sets to iterate multiplied by their average write-set size. Whereas the size of the write-set is application dependent, the length of the write-set list to iterate grows with the number of write transactions. As the number of concurrent transactions increases, so does the cost of delta validation. To tackle this problem we have made a modification to the validation procedure, in which we mix the two techniques. This is presented in the code in Figure 8.

```
validate() {
  ActiveTxRecord lastSeenCommitted = helpWriteBackAll();
  snapshotValidation();
  validateCommitAndEnqueue(lastSeenCommitted);
}
```

Figure 8. The mixed validation.

Before any actual validation takes place, a transaction helps to write-back all pending commits already queued. The helpWriteBackAll method is similar to the ensureCommitStatus method, except that it helps to commit all queued records and it returns the last one that it helped to commit. Then a snapshot validation is performed. Notice that there is no synchronization and, therefore, it is possible that concurrent commits occur. However, the list of committed versions for each transactional location surely contains

at least all the commits up to the version of the `lastSeenCommitted` record. Thus, snapshot validation, at least, validates up to that point.

Still, if validation succeeds, we must check for any newer commits and ensure a valid commit position. So, we run the delta validation, but this time only from the `lastSeenCommitted` record onwards. The `validateCommitAndEnqueue` method is adjusted to receive the starting record, instead of defaulting to `activeTxRecord`.

Even though this algorithm may be slower for a low number of transactions, it will scale much better, because it keeps the list of write-sets to check small. Moreover, the initial step in which the transaction helps to write-back, is not wasted work, as this would have to be done by some transaction anyway.

Taking into account the progress guarantees of each phase of the commit algorithm, we can conclude that this algorithm is lock-free.

4. Asynchronous elimination of old versions

The existing garbage collection (GC) mechanism for old versions that we describe in Section 2.1 is already lock-free, so we could use it and still claim that the JVSTM with our new commit algorithm was lock-free. However, during our tests we have found that the existing algorithm contains a bottleneck in the atomic counters of the Active Transactions Record. In this section we will present the problem with the original GC and describe the replacement that we developed.

The problem is that the shared counters used in the Active Transactions Record, simply do not scale when used intensively. The following factors contribute to the bottleneck: (1) very short transactions, (2) read-dominated workloads, and (3) increased number of concurrent transactions in many-core machines. Given that every transaction increments/decrements these counters at the beginning/end, the more transactions there are, the more concurrent updates there will be to the counters, which is worsened when all transactions can run in parallel. Also, if the transactions are short, the percentage of time spent beginning and finishing is larger, when compared to the rest of the time. Most importantly, in a scenario where read-only transactions prevail, most transactions will be using the same version, thus trying to modify the same counter. We verified by experience that the current GC of old versions, causes a performance degradation in read-dominated scenarios, when there are above 20 parallel short transactions running.

Taking this into account, we developed a new algorithm to eliminate unused versions, which runs in a separate thread, still uses the Active Transactions Record, but does not require the records to have any counters at all.

```
class TxContext {
  volatile ActiveTxRecord oldestRequiredRecord;
  WeakReference<Thread> ownerThread;
}
```

Figure 9. The `TxContext` data structure keeps a per-thread registry of the oldest record required.

Instead, there is, as shown in Figure 9, a per-thread transaction context (`TxContext`) that holds a reference to the transaction record currently in use by the thread's running transaction. Each context is stored in a thread-local variable. Then there is a globally accessible list of all existing contexts and the GC task repeatedly iterates through this list, identifying which is the oldest record in use and cleaning all records up to the oldest in use. Each context also contains a `WeakReference` to the thread that created it (`ownerThread`). This is to allow the GC to detect when a thread has died, so that it can remove unnecessary contexts from the global list. In the rest of this section we present the details for the lock-free GC implementation.

Adding a new `TxContext` to the global list is a simple add operation on a non-blocking list. This only needs to be done once for every new thread that starts a transaction for the first time. Afterwards, each thread reuses its own context. Whenever a new transaction starts, it determines its version, and sets the corresponding `ActiveTxRecord` in the context's `oldestRequiredRecord`. When a transaction finishes (either with a commit or abort) it sets the record to `null`.

Due to the concurrency between the GC algorithm and the start of a new transaction, after setting the `oldestRequiredRecord` a transaction must check again whether there is a newer record, and, if so, upgrade to it, simply by setting it as the `oldestRequiredRecord`. This double-check sequence is required to ensure that the GC task never removes a record that may be in use by a transaction. Notice that a write transaction that commits will mark its `commitTxRecord` as committed, before setting the `oldestRequiredRecord` to `null`. This ensures that, if the GC task decides that a record is inaccessible, then new transactions will have to see the newer record, and use it. The GC's main algorithm is shown in Figure 10.

```
run() {
  while(true) {
    ActiveTxRecord rec = findOldestRecordInUse();
    cleanUnusedRecordsUpTo(rec);
  }
}

ActiveTxRecord findOldestRecordInUse() {
  ActiveTxRecord alreadyCommitted = lastCommittedRecord;

  ActiveTxRecord minRequiredRec_1 = getOldestRecord();
  if (minRequiredRec_1 == null) {
    return alreadyCommitted;
  }

  ActiveTxRecord minRequiredRec_2 = getOldestRecord();
  return (minRequiredRec_2 != null) ?
          minRequiredRec_2 : minRequiredRec_1;
}
```

Figure 10. The new GC algorithm.

The GC is launched in a *daemon* thread that runs an infinite loop: It finds the oldest record in use and then cleans all records up to that record. The algorithm to find the oldest record in use leverages on the following property: Apart from changing to `null`, the `oldestRequiredRecord` for each context never changes backwards, i.e., after being set to a record R_a it cannot be changed later in time to another record R_b, such that R_b`.txNumber` $< R_a$`.txNumber`. This property ensures the GC that once a transaction has set a record R in its context it will never access another record older than R, which is guaranteed, because a transaction always starts on the latest committed record. With this in mind, the `findOldestRecordInUse` method starts by reading the current `lastCommittedRecord`. Should the search fail to find any other record, it is safe to clean up to the `alreadyCommitted` record.

Then it performs two passes over the list of contexts to get the oldest record of each pass. The `getOldestRecord` method (not shown) returns the record with the lowest `txNumber` from the list, or it returns `null` if it does not find any record in the list of contexts. If the first pass returns `null`, then the `alreadyCommitted` record is the safest point to clean up to, and the method returns. Otherwise, a second pass is executed to ensure that, due to concurrency, it did not miss a thread that was starting a transaction with a record older than `minRequiredRec_1`, which could occur for any context C_i read before reading the context C_{min} that contained the identi-

fied `minRequiredRec_1`[5]. For this reason, the second invocation of `getOldestRecord` only needs to check the contexts up to C_{min}. For clarity, this optimization is not shown in the code in Figure 10. After the second pass, if a record older than `minRequiredRec_1` was found then that record is returned; otherwise `minRequiredRec_1` is the oldest.

Additionally, the `getOldestRecord` method also removes from the global list of contexts any context that is no longer associated with a thread. It does so when the `WeakReference` in the `ownerThread` becomes `null`.

After identifying the oldest record in use, the GC task cleans every record from the last one cleaned to the oldest in use. Cleaning a record requires an iteration over all the `VBoxBody` instances created by the commit of that record's transaction, and setting the field `previous` in each `VBoxBody` to `null`. The list of `VBoxBody` instances to iterate is stored in the `bodiesPerBucket` array, which was produced during the write-back phase of the commit, and is accessible through the record. The `cleanUnusedRecordsUpTo` method uses a pool of worker threads to do the actual cleaning work, handing off one record to clean to each thread in the pool.

The algorithm that we presented here does not use counters to keep information about which records are still required. Conversely, it may not eliminate old versions as fast as the previous version and cause memory to be kept for a bit longer than necessary. Overall, it requires more memory to maintain the data structures for the GC.

The biggest improvement provided is that contention is virtually eliminated when starting and finishing a transaction. The only cost that a transaction incurs is related to the cache coherence mechanisms, due to possible invalidations and cache misses that can occur when reading and writing to the shared variables (namely `lastCommittedRecord` and `oldestRequiredRecord`), which is negligible when compared to the previous overhead of atomically updating the counters.

As a side effect, we benefit from transactions that run faster, because now the transaction's thread does not have to spend time doing maintenance cleaning before returning after a commit.

5. Performance evaluation

Our goal is to evaluate the scalability of our lock-free implementation. Additionally, we intend to compare it with some of the top-performing STM implementations, which are blocking. Direct comparison is usually difficult for several reasons, such as the use of different programming languages, customized execution environments, lack of common benchmarks, and others. Fortunately, thanks to the DeuceSTM project [13], there are Java implementations of both the TL2 [6] and the LSA [17] algorithms that can be used out-of-the-box. Therefore, in this section, we present a comparison between TL2 and LSA running in DeuceSTM version 1.3.0, and the lock-free JVSTM.

JVSTM requires each transactional location to have an indirection handler (`VBox`) to the actual data. DeuceSTM does not assume a versioned model, and expects each transactional location to contain the data itself: It does not support the required indirection. As such, we chose not to implement JVSTM in DeuceSTM. Although technically feasible, it would require us to subvert design choices of either DeuceSTM or JVSTM, which we believe would make the comparison less fair. This decision can have significant effects on performance depending on application-specific logic that executes within a transaction, as we will see ahead.

For this evaluation, we considered four benchmarks that are often used to evaluate STM implementations. Three of them are micro-benchmarks that correspond to three different implementations of integer sets, namely using a linked list, a skip list, and a red-black tree. The common `IntSet` interface supports three operations: insertion, deletion, and search. The fourth benchmark is STMBench7 [10], which simulates the workload of a real-world object-oriented application. This benchmark implements a shared data structure, consisting of a set of graphs and indexes, which models the object structure of complex applications. It supports many different operations, varying from simple to complex. All four benchmarks measure the throughput as the number of operations executed per second. All tests used the runtime optimization for read-only transactions that allows them to skip read-set validation.

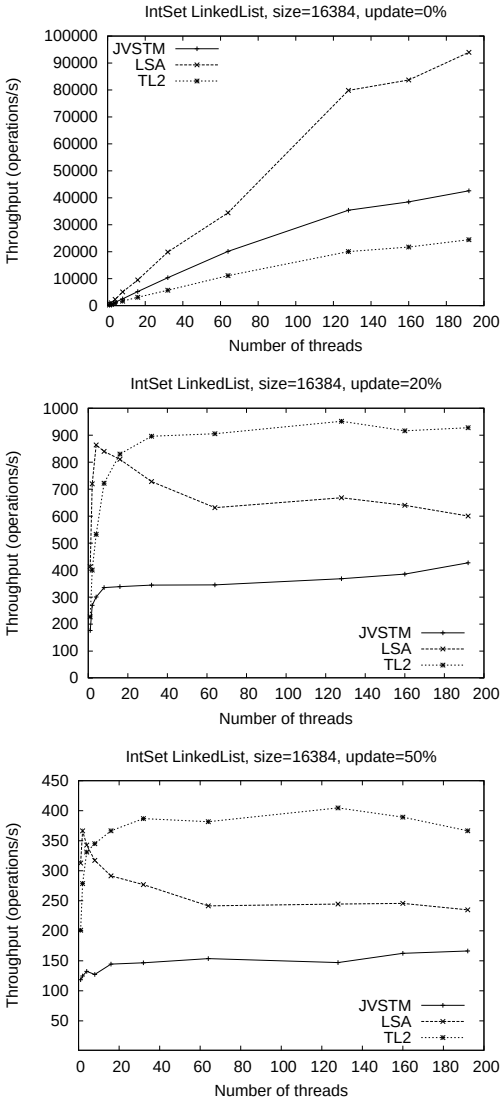

Figure 11. The linked list benchmark.

The micro-benchmarks were executed on an Azul Systems' Java Appliance with 208 cores, running a custom Java HotSpot(TM) 64-Bit Tiered VM (1.6.0_07-AVM_2.5.0.5-5-product). We run tests up to 192 concurrent transactions. Sadly, we were not able to use the same infrastructure to run the STMBench7. The reason for this is related to the way DeuceSTM works, and the requirements to run it with the STMBench7. In brief, DeuceSTM works by

[5] In this case the context C_i must have had `null` in its `oldestRequiredRecord`.

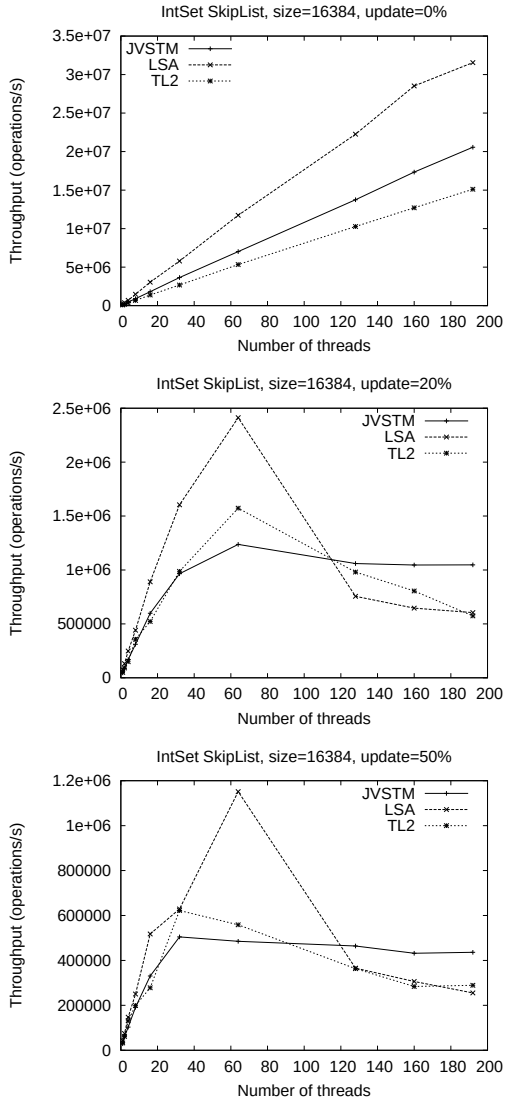

Figure 12. The skip list benchmark.

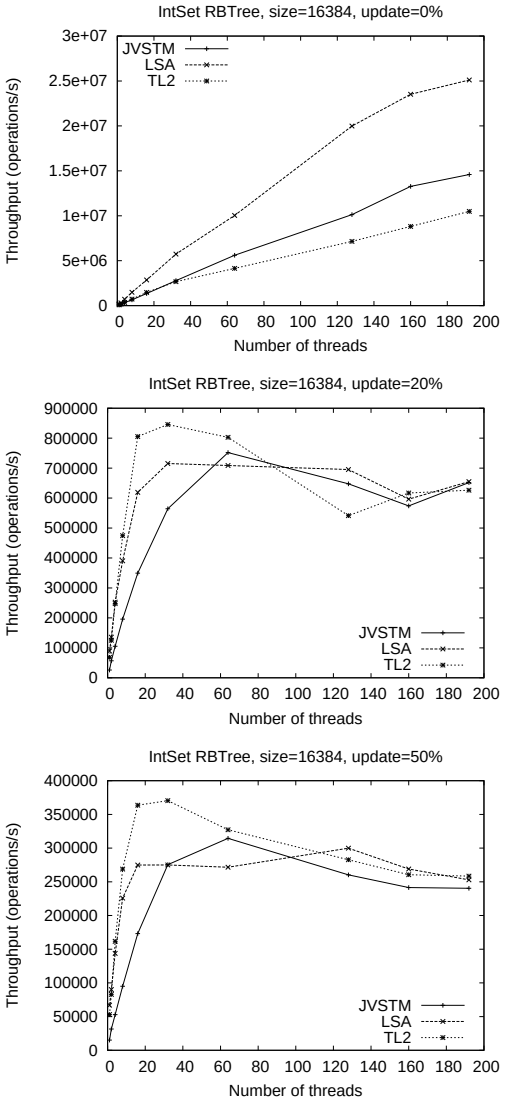

Figure 13. The red-black tree benchmark.

instrumenting the Java bytecode that runs within a transaction. In STMBench7's case, this meant that we had to pre-instrument the Java Runtime classes and start the JVM with the modified runtime (with `-Xbootclasspath/p:deuce_rt.jar` or similar). However, this did not work properly on Azul's Appliance, failing during the execution of the benchmark. For this reason we resorted to another machine with two quad-core Intel Xeon CPUs E5520 with hyper-threading, resulting in 8 cores and 16 hardware threads. We tested using Java HotSpot(TM) 64-Bit Server VM (1.6.0_20-b02) with up to 16 concurrent transactions.

5.1 IntSet micro-benchmarks

For the micro-benchmarks we used a fixed size set with 16K elements in all tests, and changed the ratio of updates. We tested with read-only transactions (only searches, 0% updates), then 20% updates, and, finally 50% updates. The write operations alternated between insertions and deletions, so that the total size of the set was constant over time.

Figures 11, 12, and 13 show the results for the three implementations when running from 1 up to 192 parallel transactions. In all read-only tests our implementation outperforms TL2, and does worse than LSA. As expected, given the absence of conflicts, all implementations steadily increase the throughput as the number of threads increases, albeit at different rates.

Considering updates, changing the ratio of write-transactions from 20% to 50% reduces the overall throughput in all cases, but does not influence the performance of each STM relative to the others. In general, memory accesses within write transactions are more expensive for JVSTM than DeuceSTM, because, whenever reading any location, JVSTM needs to access the transaction's local write-set first, to search for a transaction-local write, whereas DeuceSTM just gets the value from memory after cheaply ensuring that it is still valid. To a large extent this accounts for the reason why JVSTM does not achieve a higher maximum throughput in write transactions. JVSTM performs the worst in the linked list benchmark, showing approximately 30% of the throughput of LSA and 50% of the throughput of TL2. In the skip list benchmark, JVSTM starts off from behind, but as it keeps a sustained throughput it overtakes the other implementations beyond approximately 120 threads. The throughput of LSA and TL2 spikes at around 60 threads (more no-

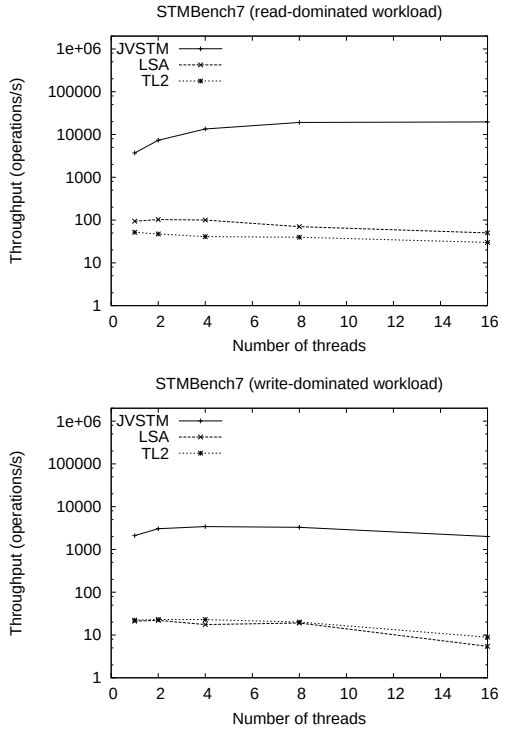

Figure 14. The STMBench7 benchmark.

ticeable in LSA), but then degrades quickly, most probably due to lock contention. In the red-black tree benchmark, all STMs perform similarly, tending to stabilize approximately on the same throughput from 60 threads onward. Overall, our implementation maintains a sustained performance over a growing number of parallel transactions, whereas the lock-based versions have higher throughput but then break for higher thread counts.

For these three micro-benchmarks the amount of code executed within each transaction is very small, and the transactions only need to manipulate the set's data structure. This causes the transactional locations managed by either STM to be the same. The next benchmark will provide a different setting.

5.2 STMBench7 benchmark

The transactions in STMBench7 vary, spanning from simple to complex manipulations of the objects, with this benchmark performing much more application logic than the micro-benchmarks. We tested with three workloads: read-dominated, read-write and write-dominated, which correspond to an increasing number of write operations. In all three cases the structural modifications and long traversals were disabled. We present the results for this benchmark in Figure 14. Here, notice that the vertical axes use a logarithmic scale. Given the space restrictions, we omit the results for the read-write workload, which are identical to the write-dominated workload. The JVSTM completely dominates the results, showing a throughput hundredfold that of LSA and TL2. As before, throughput decreases when increasing the percentage of writes.

Over-instrumentation is most likely the main reason why DeuceSTM looses performance in STMBench7. Each transaction of STMBench7 contains much more application logic that does computation with the data read than the micro-benchmarks do. Over-instrumentation results from DeuceSTM not being able to tell what is important for the transaction and what is not (e.g. access to values local to a transaction). In JVSTM, it is up to the programmer to identify the transactional memory locations (by wrapping them in `VBoxes`), which allows for a much finer control of what needs to be transactional.

There are additional reasons that contribute for this huge gap in performance. For one, the fact that with JVSTM we can provide transactional implementations for the containers required by the benchmark, whereas DeuceSTM needs to instrument the entire Java Runtime. Another reason it that STMBench7 executes many conflicting operations that manipulate the same shared objects. This necessarily causes contention when validating object accesses, either during a transaction or at commit time, and transactions may have to spend time waiting for a lock until they can decide whether they performed valid accesses. Conversely, our lock-free implementation does not prevent forward progress of the system as a whole, and allows for fast restarts in case of conflict, because validation is performed completely in parallel.

6. Discussion and related work

Even though the initial proposals for STMs used nonblocking designs (from lock-free to obstruction-free), their implementation presented large overheads, making them less performing in practice than simpler blocking designs [7]. So, most, if not all, of the most recent and current top-performing STMs block some threads to ensure exclusive access to some critical region during certain operations (e.g., TL2 [6], RingSTM [20], TinySTM [8], NOrec [5], McRT-STM [18]), typically the commit operation.

The original JVSTM followed a similar approach, using a global lock to ensure exclusive access to the commit operation of write transactions. Unlike other approaches, however, in the JVSTM read-only transactions do not use any locks, and they are wait-free. Using a single global lock has the advantage of being simple to implement and very fast when the lock is uncontended, but has the disadvantage, as we discussed already, of preventing concurrent commits that could proceed in parallel because the committing transactions accessed unrelated data.

STMs such as TL2 allow such concurrent commits by acquiring locks for all the transactional locations accessed by the transaction (and later releasing them once the commit is concluded), but this approach has overheads that are typically proportional to the size of the read-set (R) plus the size of the write-set (W). Another problem of these blocking designs is the duration of the critical region, which depends on what needs to be done within that region. The original JVSTM validates the transaction and does the write-back of the write-set with the lock held, thereby spending time that is proportional to $R + W$ in the critical region. Likewise for TL2 that needs to validate the transaction after acquiring the locks. The LSA algorithm proposed in [17] can be applied to either blocking or non-blocking STMs. We found an implementation that is obstruction-free [1], whereas the one available in DeuceSTM is actually blocking.

Our new commit algorithm allows for concurrent commits without incurring into the overheads of acquiring locks. In our design, validation is lock-free and is performed by each thread individually, racing for a position in the commit queue. The write-back phase imposes ordered writes among the committing transactions, with transactions that have a lower version number having to be committed first, but a transaction does not block waiting for other transactions to finish. Instead, transactions help to write-back other transactions that are ahead of them, thus accelerating the commit of older transactions. The write-back phase follows a wait-free algorithm.

NOrec [5] uses a lock-free validation mechanism similar to ours, but it does not allow any concurrent commits, thus limiting scalability. RingSTM [20], on the other hand, supports concurrent commits, but only insofar as the write-sets do not overlap. If this

occurs, the committing transactions have to write-back serially, and there is no helping mechanism installed.

The design of our helping mechanism allows us to dispense with more expensive locking behavior, because it maintains the sequential ordering of the write-back of all valid transactions. Besides, it maintains intact the properties of the original JVSTM design in what regards read-only transactions, as well as the behavior of reads and writes to transactional locations during the execution of a transaction. There are other proposed STM designs that include helping mechanisms, but they incur performance penalties in the reads. In [9] a transaction that reads may need to help other transactions that are writing to that location, before it can know the value. In [15] the entire read-set is validated every time a new record is accessed.

To the best of our knowledge, our commit algorithm is unique in this regard, being the first to use helping to reduce the time of the commit of a transaction, by doing the commit of a single transaction concurrently by as many cores as those available to commit on the system.

The problem of garbage collecting unused versions is seldom addressed in the context of STMs, perhaps because the most common STMs do not support multi-versioning. In [16] the authors show that no STM can be optimal in the number of previous versions kept. In fact, JVSTM may keep more versions than are actually required. Such can occur, e.g., when a long-running transaction is still running with an old version, and there have already been many updates to the same object. This will cause JVSTM to keep many versions of the same location, until the old transaction finishes and allows for record cleaning, whereas, in fact, the only versions required could simply be the oldest and the newest versions of the object, because no other transaction was running at the moment. However, and contrary to what is stated in [16], JVSTM is MV-permissive, because it keeps all versions that any running transaction may require. This holds true for both the previous garbage collection algorithm and the new one.

7. Conclusion

In face of the widespread growth of parallel computational power, the concern for the development of scalable applications is gaining relevance. In this paper we have presented a new scalable and efficient lock-free commit algorithm for the JVSTM, which shows performance comparable to, and sometimes better than, other top-performing blocking STMs. We have also presented a new lock-free algorithm for the garbage collection of unused versions, which eliminates the contention that existed in the shared counters used by the previous garbage collector.

Because we had access to a test machine with a high number of real cores, we were able to obtain results that provide high levels of confidence about the real scalability of the algorithms. We are confident on their ability to scale to an even higher number of cores and we expect, in the future, to be able to experimentally confirm this assumption. Regarding the effects on the overall performance of applications, there were mixed results especially among the micro-benchmarks, showing that execution patterns and domain-specific logic can have a great influence on the outcome.

Acknowledgments

We wish to thank Azul Systems and Dr. Cliff Click for giving us access to the hardware on which we could run some of the tests presented in this paper. We also acknowledge the reviewers' insightful comments, which have helped us to improve the final version of the paper.

References

[1] LSA-STM project home page. `http://tmware.org/lsastm`.

[2] J. Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, 2007.

[3] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[4] N. Carvalho, J. Cachopo, L. Rodrigues, and A. Rito-Silva. Versioned transactional shared memory for the FénixEDU web application. In *WDDDM '08: Proceedings of the 2nd workshop on Dependable distributed data management*, pages 15–18, New York, NY, USA, 2008. ACM.

[5] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP '10: Proc. 15th ACM Symp. on Principles and Practice of Parallel Programming*, jan 2010.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, pages 194–208, sep 2006. Springer-Verlag Lecture Notes in Computer Science volume 4167.

[7] R. Ennals. Efficient software transactional memory. Technical Report IRC-TR-05-051, Intel Research Cambridge Tech Report, Jan 2005.

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

[9] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[10] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: A benchmark for software transactional memory. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 315–324, New York, NY, USA, 2007. ACM.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[13] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MultiProg 2010: Third Workshop on Programmability Issues for Multi-Core Computers*, 2010.

[14] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[15] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, New York, NY, USA, 2008. ACM.

[16] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC '10: Proceedings of the 29th ACM symposium on Principles of Distributed Computing*, July 2010.

[17] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, Sep 2006. ISBN 3-540-44624-9.

[18] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 187–197. ACM Press, 2006.

[19] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

[20] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08: Proc. twentieth annual symposium on Parallelism in algorithms and architectures*, pages 275–284, jun 2008.