A scalable and efficient commit algorithm for the JVSTM *

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

Abstract

Most of the modern and top-performing Software Transactional Memory implementations use blocking designs, relying on locks to ensure an atomic commit operation. This approach revealed better in practice, in part due to its simplicity. Yet, this approach may have scalability problems when we move into many-core computers.

In this paper we present and discuss the implementation of a new scalable and efficient commit algorithm for the Java Versioned Software Transactional Memory (JVSTM), a multi-version STM implementation that uses a global lock to commit write transactions.

The new commit algorithm allows commits to proceed in parallel during the validation phase, and reduces the critical region of a commit to a minimum by resorting to helping from threads that would otherwise be waiting to commit. Instead they help in the write-back of previous committing transactions. We evaluate the new algorithm with three benchmarking applications, and show it to scale well up to 172 cores, the maximum number of physical cores available on the machine used for the tests. We analyze also how the new commit algorithm affects the overall performance of the applications.

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

General Terms Algorithm, transaction, commit, scalability

Keywords JVSTM, transactional memory

1. Introduction

Since the seminal work on Transactional Memory [12] and Software Transactional Memory (STM) [14], 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.

* This work was partially supported by the Pastramy project (PTDC/EIA/72405/2006).

TRANSACT 2010 April 13, Paris, France.

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 [2, 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 reduces the critical region of a commit to a minimum by resorting to helping from threads that would otherwise be waiting to commit. We evaluate the new algorithm with three benchmarking applications, and show it to scale well up to 172 cores, the maximum number of physical cores available on the machine used for the tests. We analyze also how the new commit algorithm affects the overall performance of the applications.

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, and in Section 4 we evaluate its performance using three benchmarking applications. Section 5 provides a discussion on properties of the algorithm and related work. We conclude in section 6.

2. JVSTM overview

JVSTM 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 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

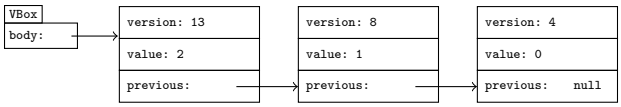

Figure 1. A transactional counter and its versions.

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 shows the abstract commit algorithm of write transactions (the commit of read-only transactions simply returns).

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

The global lock provides mutual exclusion among all committing write transactions, which is important to ensure 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 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;
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).

2.1 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 heavy-read 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 limited 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. Scalable 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.

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 by checking the write-sets of all transactions that have committed versions greater than the one this transaction started with (*incremental validation*). Suppose that T_1 started in version v_1. For any transaction T_i that committed a version greater than v_1, then if any element in the write-set of T_i is in the read-set of T_1, T_1 cannot commit, because of a conflict.

Of course that, using this lock-free validation, while T_1 is validating itself, other transactions can also be validating themselves to commit. 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, which already exists in the JVSTM. This structure was created to manage garbage collection of old versions of the transactional locations. In brief, it holds a list of transaction records deemed to be active. Each record keeps information about a write transaction that already committed, namely the commit version and the values that were modified. A record is active if that committed version of the values is still accessible by any other transaction. Inactive records can safely be garbage collected, because they represent inaccessible versions. The implementation is a singly-linked list of records, with each record holding a reference to the next most recent record. To support the incremental validation, we extend the 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 2 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 2. 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 its read-set against 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 that atomically sets the `commitTxRecord` as the next valid record after the `lastValid`. This is a tentative operation that 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 first gain a position in this list. 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 3, shows an example of a possible state of the Active Transactions Record. Transactions 9 and 10 have already committed[1], whereas transactions 11 to 13 are valid but not yet written-back. At this time, any transaction that starts, will start in version 10, because it is the most recent committed version. We also show the two transaction records in use by transaction 12 (thus, assuming that transaction 12 started before transaction 10 had committed).

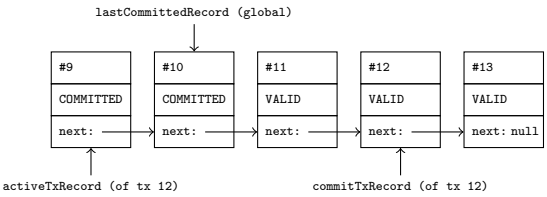

Figure 3. An example snapshot of the Active Transactions Record.

Writing-back consists of creating a new `VBoxBody` in 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 two valid transactions will write-back to the same transactional location. 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 4.

After validation, transactions run the `ensureCommitStatus` method. This method starts by accessing the globally available `lastCommittedRecord` to get the next record to commit. When the `helpCommit` method returns, it is guaranteed that the transaction corresponding to `recToCommit` has been committed.

The maximum number of transactions that can help to write-back is naturally limited by the number of processors available.

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

```
ensureCommitStatus() {
  ActiveTxRecord recToCommit = lastCommittedRecord.getNext();
  while (recToCommit.txNumber <= this.commitTxRecord.txNumber) {
    if (!recToCommit.isCommitted()) {
      helpCommit(recToCommit);
    }
    recToCommit = recToCommit.getNext();
  }
}

helpCommit(ActiveTxRecord recToCommit) {
  WriteSet writeSet = recToCommit.getWriteSet();
  boolean lastOne = writeSet.helpWriteBack(recToCommit.txNumber);
  if (lastOne)
    finishCommit(recToCommit);
  else
    waitUntilCommitted(recToCommit);
}

boolean helpWriteBack(int newTxNumber) {
  int block = nextBlock.getAndIncrement();  //atomic counter
  while (block < N_BLOCKS) {
    writeBackBlock(block, newTxNumber);
    if (blocksDone.incrementAndGet() == N_BLOCKS) { //atomic counter
      return true;
    }
    block = nextBlock.getAndIncrement();  //atomic counter
  }
  return false;
}
```

Figure 4. The method that ensures the correct execution order of all pending write-back operations together with the helping methods.

If we happen to have more transactions committing than CPUs available, it is simply better to put the extra transactions "on hold", and let the others do the work. The processors will already be occupied, writing-back as fast as possible.

To avoid contention between concurrent helping transactions we split the values to write-back into several buckets, of approximately the same size each. Any transaction can help to write-back, by picking a unique bucket and writing-back the values in that bucket. Writing to each location can proceed without synchronization, as no other thread will be writing to the same location. This is guaranteed, because the write-set does not contain duplicates, and only one write-set is written-back at any given time, even though several buckets of the same write-set can be written-back in parallel.

Ideally, we use as many buckets as the number of available processors to maximize parallelism, unless it causes the size of each bucket to be very small. Writing-back a single value is very fast and, usually, it does not make sense to have a transaction help with that. Therefore, we use a minimum value for the bucket size, which can lead to less buckets than the number of processors, but that, in practice, works better.

The write-set itself holds the buckets, so `helpCommit` delegates the write-back operation to the write-set, configuring it with the version number to use. The `helpWriteBack` method will write-back as many buckets as possible (one at a time), and return whether this helping transaction was the last to finish writing-back a bucket. Such transaction will have the responsibility of finishing the commit of the transaction being helped. Observe that any helping transaction T_j will finish the commit of another transaction T_i only if T_j happens to be the last one to finish the write-back of a bucket of T_i. Moreover, unless $i = j$, in which case the transaction is helping itself, T_i is necessarily in front of T_j, in commit order. Finishing a commit entails publicizing the changes globally, and it can be done only after all values are in place.

If the `helpWriteBack` method returns `false`, then all buckets are already taken by some other helping transaction and they are still being written-back. This transaction cannot help any further and it will have to wait until others finish their part. For a transaction that actually helped, and taking into account that bucket sizes are uniformly distributed, then the other helping transactions will, on average, finish almost at the same time, so the wait is expected to be short. If a transaction did not help at all, then it is because there are more committing transactions than buckets, and this implies that the available processors are already fully occupied, so waiting is acceptable.

The current implementation of the `waitUntilCommitted` method uses an exponential back-off algorithm. The method cyclically checks the commit status of the given record, sleeping for some random time within an increasing interval, until it sees the record committed. We have tested with other implementations, but this solution yielded the best performance for the waiting procedure, based on experimental results. We also developed alternative commit algorithms that allowed for a complete nonblocking write-back, but they all entailed some penalty to the readers of transactional locations, which would make those solutions unsuitable to keep our goal of not affecting the read operations.

The only synchronization required is, thus, between the helping transactions in two points: (1) when they pick a unique bucket to write-back; and, (2) when they signal the completion of a bucket, to identify the last writer. For this, we use two atomic counters `nextBlock` and `blocksDone`. The `helpWriteBack` method, of the `WriteSet` class, presents the use of the two counters.

Each write-set instance contains an array of `N_BLOCKS` blocks. The number of blocks may vary from one write-set instance to another, but never changes during commit. Both counters are initially set to zero. The `getAndIncrement` method atomically ensures a unique block index, for each invocation. If the index is within valid limits, that block is written-back. After writing-back a block the `blocksDone` counter is incremented. If the finished block is the last, then the method returns `true`. Otherwise, it gets another block index to write-back. When the block index exceeds the limit the method returns `false`, signaling to the invoker there is nothing else that can be done. The `writeBackBlock` method has no synchronization, and it does the actual work of copying the values from the given write-set block to their public location.

The atomic increment of the `blocksDone` counter ensures a happens-before relation between other helping transactions that have already incremented this counter, and this transaction, which, in turn, ensures that all values previously written-back are now visible by the transaction incrementing the counter. This is important, because the last helping transaction to write-back will perform the final commit of the helped transaction and, as such, we need to ensure that all values written-back are effectively seen by transactions that start after committing the helped transaction.

One way to reduce the waiting performed by a committing transaction that is no longer able to help (because all buckets are already taken) is to start writing-back the write-set of a following record. This can be safely done as long as the write-set of that record does not intersect the write-set of any previous record that is still not written-back. An example of an STM that uses this approach is RingSTM [15], which is able to intersect write-sets efficiently because it uses Bloom filters to represent the write-sets (note that false positives in the intersection operation do not affect correctness). Given that intersecting write-sets is a relatively expensive operation in our implementation and that the waiting is expected to be short, we did not experiment with this approach in our current design.

3.3 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) an incremental validation 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, incremental 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. Incremental 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, incremental 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 the incremental 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 following code snippet.

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

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, validation is performed 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 incremental 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.

3.4 Finishing a commit

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:

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

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.

4. Evaluation

To evaluate the scalability of the new commit algorithm, we tested it using three benchmarking applications. For each one, we tested the average commit time per transaction, and the overall application performance, using from 1 up to 256 concurrent transactions, with both the old and the new versions of the commit algorithm. These tests were executed on an Azul Systems' Java Appliance with 172 cores, running a custom Java HotSpot(TM) 64-Bit Tiered VM (1.6.0_07-AVM_2.5.0.5-5-product).

The first benchmark (Array) is a micro-application that we developed. It is highly configurable, and very useful to stress test the JVSTM, but it does not implement any real-world behavior. It allows us to configure several execution parameters, such as the number of transactions to execute, how many of them can run concurrently, the number of reads and writes per transaction, the number of read-only transactions, write transactions, and so forth. It creates an array of integer transactional locations, and each transaction can read and write values to random positions of the array. This benchmark is useful, because the results that it provides are unaffected by complex application logic, and as such, they are more sensitive to changes inside the JVSTM.

The other two benchmarks are often used in STM performance tests and they implement real-world workloads: Lee-TM [16] and STMBench7 [10]. The former implements Lee's routing algorithm for automatic circuit routing and it measures how long it takes to lay down tracks on a circuit board. The latter provides a shared data structure, consisting of a set of graphs and indexes, which models the object structure of complex applications, and it measures the throughput of a set of concurrent operations that change the shared data. Testing with real-world scenarios is important, because they provide a quantitative measure of the expected influence that the changes in the JVSTM can produce in real applications.

We present the results of the tests in Figures 5 and 6. For the Array benchmark we ran to completion 10^4 write transactions, each transaction executed 10^3 reads and 10 writes, randomly distributed over an array of 10^6 positions. Lee's algorithm was applied to the memory circuit board from [1]. The STMBench7 ran with structural modifications disabled, a write-dominated workload, and with long traversals disabled.

Figure 5 shows the average commit times per transaction, for each benchmark. The number of threads in the horizontal axis, indicates the number of concurrent transactions allowed to run (each transaction runs in a different thread). Note that the test machine has 172 cores, so above that value multi-processing is implied, whereas there is hardware support for real parallelism in the tests running with up to 128 threads (i.e., unless contented, transactions can run at full speed). Also, observe that the vertical axes use a logarithmic scale.

An excellent result for the new algorithm is that, within each benchmark, the commit times have almost no variation for any

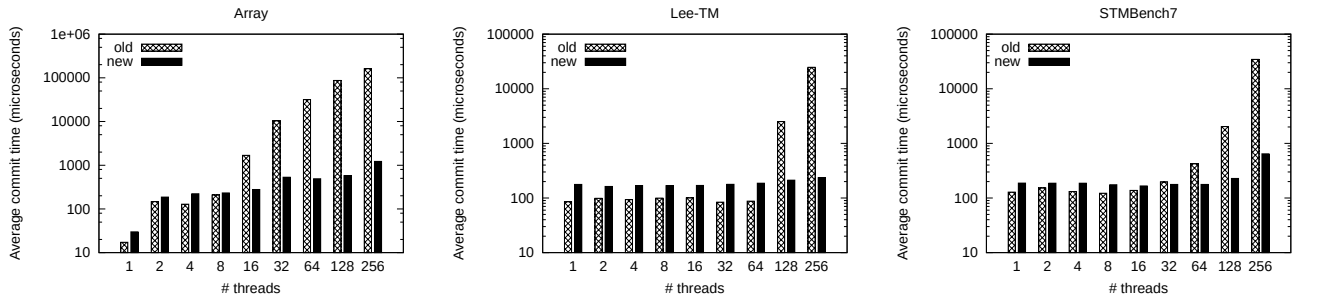

Figure 5. Average commit time per thread.

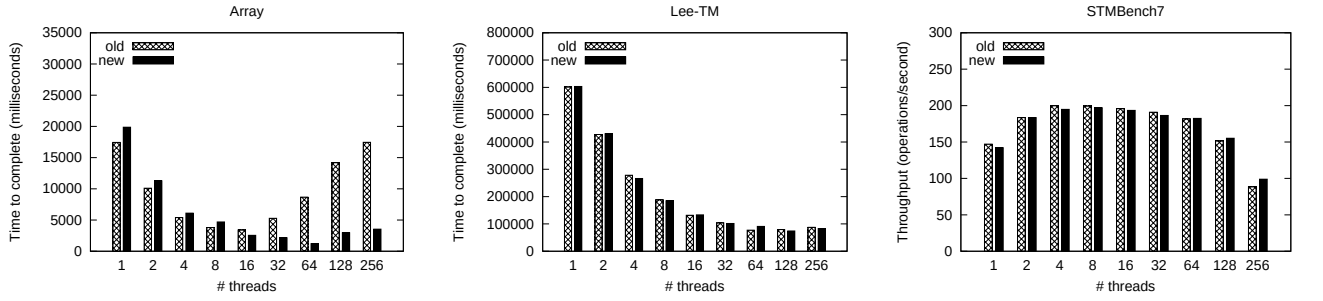

Figure 6. Overall performance of the applications.

number of threads, showing that the new commit algorithm scales well, at least up to the number of threads that we tested. With the old algorithm, performance degrades, because the higher contention for the commit lock introduces an increasing waiting time in the commit operation. The point where it starts to degrade varies with the benchmark, because the percentage of time spent committing varies when compared to the time of the whole transaction. The Array has the shortest transactions, so the commit takes a larger percentage of the time, thus increasing the probability of contention. On the downside, the new algorithm is slower for a smaller number of threads, mostly due to the more expensive mixed validation algorithm. Still, the new algorithm outperforms the old in all benchmarks for a larger number of threads. For the single-threaded executions, the difference between the old and new versions effectively shows the added cost of the new algorithm. Whereas for a reduced number of concurrent transactions the additional complexity may not be worth it, we see that for a large number (approximately above 100), it starts to pay off.

We are also interested in measuring the influence of the improved commit on the overall performance of applications. For Array and Lee-TM we measured the time to complete the tasks, and for STMBench7 we measured the number of operations per second performed over a period of two minutes. Figure 6 shows mixed results for the three benchmarks. In Array, the effects of the scalable commit are clearly visible. The old version does not scale beyond 16 threads, exactly the point where the commit time starts to increase greatly. This application is highly concurrent, its transactions are short, and the time spent committing represents a significant part of the transactions' total time. Considering Lee-TM, the differences are insignificant. The application performs many long transactions that write only a few memory locations, which cause the commit time to have little relevance on the overall result. For STMBench7, throughput actually decreases for both versions as we increase the number of threads, with the old version performing slightly better up to 32 threads. For a higher number of threads, the new algorithm performs a little better, but this application executes many conflicting operations that write to many locations, unfortunately causing a reduction in throughput for both versions.

We conclude that, even though there may exist significant improvements in the average commit-time of a transaction, they might have only little consequences in the overall performance of real-world workloads, mostly because the duration of the commit is much shorter than the remainder of the application, and for a low number of threads, the probability of two transactions being in the commit phase at the same time is reduced.

The new algorithm is best suited to situations where there is an increased probability of concurrent commits. Such situations depend on the combination of the application's workload and a large enough number of concurrent transactions. With the micro-benchmark Array we were able to simulate such situations.

We speculate that with a growing number of cores available, the degradation caused by the global commit lock will become more noticeable, and, consequently, the new commit algorithm will become more suitable. Finally, we verify that even though the new algorithm is better suited for many-core scenarios, it may also be used in place of the old one with little performance penalty for low thread counts.

5. 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 [15], TinySTM [8], NOrec [5]), 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 are not affected by this approach, 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.

Our new commit algorithm is still blocking, but allows for concurrent commits without incurring into the overheads of acquiring locks as in other designs, and reduces the critical region to a minimum.

In our design, validation is lock-free and is done outside the critical region. Moreover, a thread is blocked only when it needs to wait for the end of the write-back of a previous transaction, but given that we have helping in the write-back, where we may have all of the cores doing the write-back concurrently, the duration of the critical region is highly reduced. Note that the coordination of the helping process is done through a lock-free algorithm, also. NOrec [5] uses a similar lock-free validation mechanism, but it does not allow any concurrent commits, thus limiting scalability. RingSTM [15], 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. Only the commit of write transactions was changed. There are other proposed STM designs that include helping mechanisms [9, 13], but they incur performance penalties in the reads.

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 on the system,

6. 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 how the JVSTM can be improved with a new scalable and efficient commit algorithm.

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 algorithm. We are confident on the algorithm's 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, showing that execution patterns and domain-specific logic have a great influence on the outcome, due to the variation on the relative time that is spent in the commit operation, when compared with the entire transaction. These results, on the other hand, reveal that the simple approach used in the original JVSTM goes a long way without showing real problems, thereby reinforcing the idea of choosing simple designs whenever possible.

Acknowledgments

We wish to thank Azul Systems and Dr. Cliff Click for giving us access to the hardware on which we could run 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] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *ICA3PP '08: Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, pages 196–207, Berlin, Heidelberg, 2008. Springer-Verlag.

[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] 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.

[14] 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.

[15] 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.

[16] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.