J. Parallel Distrib. Comput. 89 (2016) 13–24

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Optimizing memory transactions for large-scale programs

Fernando Miguel Carvalho[a,b,*], João Cachopo[a]

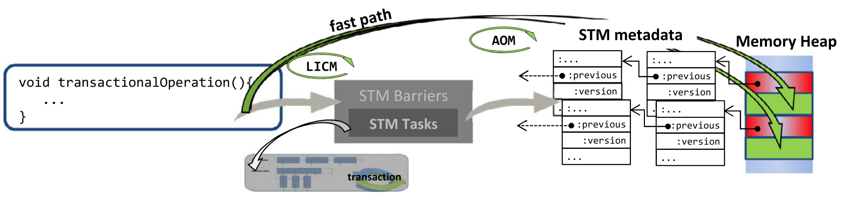
[a] INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal

[b] ADEETC, Instituto Superior de Engenharia de Lisboa, Polytechnic Institute of Lisbon, Portugal

HIGHLIGHTS

- A new technique of adaptive object metadata (AOM) that eliminates the extra STM metadata.
- AOM with LICM (lightweight identification of captured memory) provide a fast path for non-contended objects.
- Results that show performance with an STM that rivals a fine-grained lock in a large-scale benchmark.
- Integrated in Deuce STM full support for in-place metadata that is required by LICM and AOM.
- Innovative adaptation of Deuce STM: maintains original API, and enhances any existing STM.

GRAPHICAL ABSTRACT



```
void transactionalOperation(){
   ...
}
```

ARTICLE INFO

Article history:
Received 8 November 2013
Received in revised form
20 February 2015
Accepted 3 December 2015
Available online 14 December 2015

Keywords:
Software Transactional Memory
Runtime optimizations
Concurrent programming

ABSTRACT

Even though Software Transactional Memory (STM) is one of the most promising approaches to simplify concurrent programming, current STM implementations incur significant overheads that render them impractical for many real-sized programs. The key insight of this work is that we do not need to use the same costly barriers for all the memory managed by a real-sized application, if only a small fraction of the memory is under contention—lightweight barriers may be used in this case. In this work, we propose a new solution based on an approach of *adaptive object metadata* (AOM) to promote the use of a fast path to access objects that are not under contention. We show that this approach is able to make the performance of an STM competitive with the best fine-grained lock-based approaches in some of the more challenging benchmarks.

© 2015 Elsevier Inc. All rights reserved.

1. Introduction

The idea of providing hardware support for transactions was firstly introduced by Knight [18] to check the correctness of parallel executions of Lisp programs. Later, Herlihy and Moss [16] extended the concept to the notion of Transactional Memory (TM) and then Shavit and Touitou [26] proposed a software realization of the same idea, called Software Transactional Memory (STM), however, with a different interface from the original TM proposal.

One of the key selling points of TM is that it simplifies the development of concurrent programs, because programmers do not have to enforce isolation through some concurrency control mechanism, but instead, they just have to say which groups of operations should be executed *atomically*. To that end, Harris and Fraser [13]

[*] Corresponding author at: INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal.

E-mail addresses: mcarvalho@cc.isel.ipl.pt (F.M. Carvalho), joao.cachopo@ist.utl.pt (J. Cachopo).

http://dx.doi.org/10.1016/j.jpdc.2015.12.001
0743-7315/© 2015 Elsevier Inc. All rights reserved.

proposed the use of an `atomic` construct to provide a style of *conditional critical region* [17] in the Java programming language. Alternatively, and following the same approach, the Deuce STM [20] provides a simple API based on an `@Atomic` annotation to mark methods that must have a transactional behavior.

Unfortunately, one of the consequences of making an STM completely *transparent* to the programmer is that it may add large overheads to the program. A major source of overheads in STM-based programs is the use of *STM barriers* whenever a memory location is accessed within a transaction. These barriers are responsible for keeping track of what is done inside a transaction (at a minimum, STM barriers need to keep track of what is read and written), so that the STM runtime is able to ensure the intended transactional semantics.

This problem was pointed out by some authors (e.g., [6,22]), who raised questions about the practical applicability of STMs to large-scale programs. In fact, whereas STMs have shown promising results when applied to micro-benchmarks, they typically perform worse than coarse-grained lock-based approaches in larger benchmarks with a large number of shared objects.

Yet, we claim that it is possible to reduce substantially the STM-induced overheads for a large-scale program if we assume that the amount of memory under contention – that is, memory being concurrently accessed both for read and for write – is only a small fraction of the total amount of memory accessed by that program.

The key idea is that for non-contended memory (that is, memory not being concurrently accessed both for read and for write) we can avoid the full-blown STM barriers, which should be used only for accessing (the relatively rare) memory under contention. Instead, for the frequent non-contended memory accesses we use lightweight barriers that access directly the target memory, thereby significantly reducing the overheads imposed by the STM. Our approach combines two orthogonal techniques – *lightweight identification of captured memory* (LICM) and *adaptive object metadata* (AOM) – to reduce the overheads of accessing objects that are not under contention.

LICM allows the automatic identification of transaction-local objects at runtime, for which no STM barriers are needed, but its use does not eliminate all of the excessive overheads caused by an STM in many large-scale benchmarks. Yet, it complements well the second technique that we propose in this paper. AOM is an optimization technique for multi-versioning STMs, based on the JVSTM [10] general design, that is adaptive because the structure of the metadata used for each transactional object changes over time, depending on how the object is accessed.

By combining AOM with LICM, we are able to outperform the coarse-grained (and compete with the best fine-grained) lock-based approaches in some of the more challenging benchmarks, while retaining ease of programming.

In Section 2, we present some motivation for this work and present an overview of our approach. Then, in Section 3 we introduce the key aspects of the JVSTM that are relevant to understand our proposal. In Section 4, we describe in detail the AOM design and discuss the correctness of its operations. In Section 5, we present an experimental evaluation for a variety of benchmarks. In Section 6, we discuss related work on efficient support for STMs. Finally, in Section 7 we conclude and discuss some future work.

2. Motivation and solution overview

Despite the promising results of STMs for micro-benchmarks, when we apply STMs to larger benchmarks, such as STMBench7 [12] or Vacation [2], they typically perform worse than the single-threaded sequential version of the same benchmark. This effect was observed by Fernandes and Cachopo [10], who show that using Deuce with either TL2 STM [8] or LSA STM [25] in STMBench7 achieves a throughput up to 100 times lower than using a coarse-grained lock. Interestingly, however, they also show that by manually instrumenting STMBench7 with the JVSTM (rather than with Deuce), it was possible to get better performance than with the medium-grained locks, which suggests that, after all, it is possible to get good performance from STMs in large-scale applications.

The large gap in performance observed in that work may be attributed to the difference in the STMs used, to the amount of barriers that are introduced into the benchmark by each approach (Deuce vs manual),[1] or, most probably, to a combination of both.

To help us pinpoint the causes for the differences observed by Fernandes and Cachopo [10], we ran STMBench7 with three synchronization approaches: (1) using Deuce to instrument all of the code and executing it using 3 different STMs: TL2, LSA, and JVSTM[2]; (2) using the coarse-grained and medium-grained lock-based synchronization of STMBench7; and (3) using an STMBench7 that was manually instrumented to use the JVSTM.

In Fig. 1, we show the throughput of the benchmark for each of the synchronization approaches, when the number of threads varies from 1 to 48 (in Section 5 we describe the details of the experimental platform). The results show that for this workload the JVSTM performs better than the other STMs (even when all the instrumentation is made by Deuce), but the more telling aspect is the huge gap in performance between using the JVSTM with Deuce or manually: Whereas when using the JVSTM with Deuce the throughput never gets above the sequential non-instrumented execution, in the manual case we get a speedup of 3 times, outperforming even the medium-grained lock-based approach.

Given that the JVSTM used in both cases is exactly the same, this difference must result from the over-instrumentation made by Deuce. This over-instrumentation has two consequences: (1) it adds more STM barriers to the execution of the benchmark; and (2) it adds to each object extra metadata, which needs to be traversed when accessing those objects. Both affect performance negatively.

Our goal is to suppress STM barriers and avoid the metadata indirections in situations where they are not really necessary—that is, when accessing objects that are not under contention. By integrating our solution in Deuce we expect to be able to achieve results similar to those obtained with the manual use of the JVSTM, albeit without the intervention of the programmer (thereby, making the STM easier to use).

The first part of our solution is based on the use of LICM, which we introduced in [5] and further enhanced in [4]. LICM provides a new lightweight runtime technique to elide STM Barriers when accessing *captured memory*—that is, memory allocated inside a transaction that cannot escape (i.e., is captured by) its allocating transaction, as defined by Dragojevic et al. [9]. Captured memory corresponds to newly allocated objects that did not exist before the beginning of their allocating transaction and that, therefore, are held within the transaction until its successful commit. In our LICM work, we were concerned with how to perform the *runtime capture analysis* efficiently. In our solution, we label new objects with a *fingerprint* that uniquely identifies their creating transaction, and then check if the accessing transaction corresponds to that fingerprint, in which case we avoid the barriers. This check is simply an identity comparison between the fingerprint of the accessing transaction and the fingerprint of the accessed object, and, thus, a very lightweight operation. The idea of the fingerprint

[1] When using Deuce, all classes are automatically instrumented, whereas when manually using the JVSTM only the classes of the effectively shared objects are instrumented.

[2] For this purpose we integrated in Deuce the lock-free version of the JVSTM, which was not available yet.

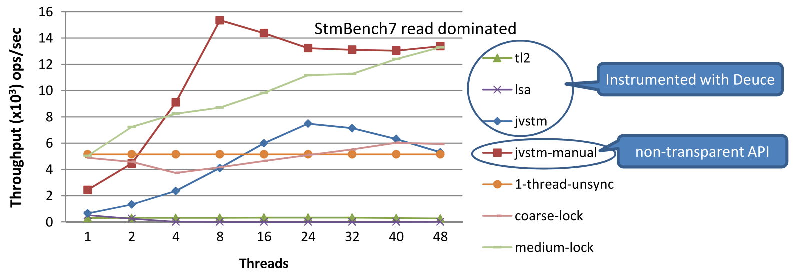


Fig. 1. The throughput of STMBench7 for the read-dominated workload without long traversal operations, using several synchronization approaches.

is inspired in the design of the Bartok STM [14] and the use of a special metadata tag to uniquely identify the allocating transaction and its allocated objects. However, this technique still needs to log these objects in the *updated-object log*, so that the objects can be closed when the transaction commits, whereas the LICM does not require it and suppresses all the overheads of the transaction bookkeeping.

The second part of our solution is based on the AOM approach, which aims to remove unnecessary metadata for non-contended objects. The key insight that will allow us to eliminate this overhead is that the STM metadata is needed only when several transactions contend for the same transactional object and at least one of those transactions writes to that object. Thus, assuming that in large-scale applications the vast majority of objects are seldom written, the number of objects that need the metadata should be residual when compared to the total number of objects in the application. This, in turn, means that if we use a compact representation for the non-contended objects, we may have a significant reduction both in the memory used and in the performance overhead. Our proposal is based on the idea of transactional objects having *adaptive object metadata* (AOM).

LICM and AOM techniques complement each other, substantially reducing the overheads of an STM in large-scale programs. We combined both techniques and integrated them in the Deuce STM framework. As we shall see, our approach can solve one of the major bottlenecks that reduces the performance in many large-scale applications and simultaneously preserves the transparency of an STM API, as shown with its implementation in Deuce.

3. JVSTM overview

The JVSTM [10] is a Java library that implements a lock-free, *multi-versioning* STM. It uses the core concept of *versioned boxes (vboxes)*, which can be seen as a replacement for transactional memory locations. Instead of keeping only a single value, a *vbox* (instance of the `VBox` class) maintains a sequence of values – the vbox's *history* – where each value is tagged with a *version* corresponding to the number of the transaction that has committed that value. Each entry in the history of a vbox is an immutable *vbox body* (instance of the `VBoxBody` class with `final` fields), and the entries are sorted by their version in descending order.

During a transaction, writes to a vbox are logged into a per-transaction *write-set* and do not affect the history of the vbox until commit time. Similarly, reads of vboxes are logged into a per-transaction *read-set*, which will be used for validating the transaction.

At commit time, read–write transactions are checked if they are *valid*[3] by verifying that all of the vboxes in their read-set are still up-to-date—that is, that none of the vboxes read by the transaction have been changed in the meanwhile by another concurrent transaction that successfully committed. Otherwise, we say that there was a *conflict* and the transaction cannot commit successfully; instead, it needs to *restart* in the new version of the program's state, corresponding to the version of the last committed transaction.

Write-only and valid read–write transactions will try to enqueue into a global queue of transactions that want to commit; these transactions are represented in the queue by instances of the class `ActiveTransactionRecord` (ATR). By successfully getting into that queue, a transaction obtains a global order for commit and is guaranteed to commit in that order, possibly with the help of other transactions waiting for their turn to commit: This results in a *lock-free* commit algorithm.

During the commit of a (successfully enqueued) transaction, there is the *write-back* phase, which is when the new values of the vboxes changed during the transaction are added to each vbox's history, as shown in Fig. 2. As a result of the helping during the commit, more than one thread may attempt to write-back to the same vbox. Thus, the `commit` method of class `VBox`, which implements the write-back of a vbox, uses a *compare-and-swap* (CAS) operation[4] to install the new `VBoxBody` at the head of the list of bodies. If the CAS fails, then another thread must have successfully completed the write-back for this vbox. In either case, the write-back of a vbox returns the vbox body that was successfully added to the vbox's history. All these bodies are collected and stored in the ATR corresponding to the committing transaction, and, after the write-back operation is completely finished this ATR passes to the *committed* state (corresponding to set a commit flag that establishes the *linearization* point of the commit algorithm), making the changes globally visible.

The history of a vbox is needed to ensure that reads will always be able to access a consistent view of the shared-memory state. At transaction begin, transactions obtain the number of the latest committed ATR and store that number as their *reading version*. This version is used in all reads to ensure that they are consistent: Reading a vbox traverses the vbox's history to obtain the value corresponding to the transaction's reading version (the value with the largest version smaller than or equal to the transaction's version). This is one of the key elements that guarantees that the JVSTM satisfies the *opacity* correctness property [11].

Thus, old entries in a vbox's history – that is, all entries except for the most recent one – may be discarded as soon as there are no active transactions that may need to access those entries. To discard no-longer accessible versions, the JVSTM implements its own garbage collection (GC) algorithm, which only needs to look at version numbers and does not test reachability via references.

The JVSTM's GC runs in its own thread and uses the information from the ATR objects to find out the inaccessible versions. An ATR is considered *inactive* when there is no transaction with that

3 In the JVSTM, read-only and write-only transactions do not need to validate as they may always commit successfully.

4 Based on the `compareAndSwap` version of the `sun.misc.Unsafe`, which receives the parameters: `Object target`, `long fieldOffset`, `Object expected`, `Object newValue`.

```
// in class VBox:
VBoxBody commit(Object newValue, int txNumber) {
  VBoxBody currHead = this.body;
  if (currHead == null || currHead.version < txNumber) {
    VBoxBody newBody = new VBoxBody(newValue, txNumber, currHead);
    return CASbody(currHead, newBody);
  } else {
    return currHead.getBody(txNumber);
  }
}

VBoxBody CASbody(VBoxBody expected, VBoxBody newBody) {
  if (compareAndSwapObject(this, bodyOffset, expected, newBody)) {
    return newBody;
  } else { // if the CAS failed the new body must already be there!
    return this.body.getBody(newBody.version);
  }
}
```

Fig. 2. Algorithm used by the JVSTM to write-back to a vbox. The `commit` method receives the new value and the transaction number corresponding to the version of the new value.

```
// in class ActiveTransactionRecord:
void clean() {
  for (Pair<VBox, VBoxBody> pair : this.allWrittenVBoxes) {
    pair.body.previous = null;
  }
}
```

Fig. 3. Algorithm to clean the VBoxBody objects committed by that record's transaction.

ATR's number and there is already a newer ATR enqueued (this ensures that new transactions will start in a more recent version and the versions of the *inactive* ATR are inaccessible). In this case, the cleaning work is triggered for the newer record, which calls the `clean` method shown in Fig. 3.

The `clean` method of an ATR iterates over all of the bodies that were written back for the commit of that ATR's transaction and trims them – that is, it sets the `previous` field to `null` – because the previous entries in the history are no-longer needed. As we shall see next, our AOM approach is plugged into this `clean` method.

4. The adaptive object metadata approach

In Fig. 4(b), we show what the structure of a transactional object (in this case, a point) looks like in the JVSTM. Compare that with a plain non-transactional point, which consists only of the instance of the `Point` class. It becomes clear that using the JVSTM adds significant overhead, not only in terms of the extra memory needed to store all of the metadata, but also in terms of the cost needed to traverse all of that metadata during accesses to the object.

Even when the application reaches a quiescent state where the GC is able to trim all of the vboxes' histories but one, there is still significant overhead due to that single entry. A transactional read still has the overhead of traversing two pointers: One from the field `body` to the entry at the head of the history and another from that entry to the snapshot containing the value to be accessed. This pointer chasing typically results in poor cache locality, further contributing to the degradation of the performance.

The novelty of AOM approach is that it uses two different layouts for transactional objects – the *compact layout* and the *extended layout* – and changes objects back and forth between these two layouts, so that, most often, objects are in the compact layout, which does not need metadata.

The extended layout is similar to the original layout used by the JVSTM. However, given that there is always one and only one vbox for each object, we coalesce the vbox with the object. So, in our approach, every transactional class must inherit from `VBox`, thereby guaranteeing that each transactional object has a field `body` that points to its history of values.

The structure shown in Fig. 4(b) corresponds to the extended layout of our AOM approach. In this layout, the values contained in the fields of the `Point` class are ignored by the STM, because the values are contained in the history stored in `body`. Each value of the history corresponds to a *snapshot* of an instance of `Point`, created whenever a transaction commits some changes to that instance. Because in these snapshots the field `body` is not used, we omit it in Fig. 4(b). We say that an object with the extended layout is an *extended object*.

The goal, however, is that most objects should use (for most of the time) the compact layout, in which case the field `body` is `null` (so, there is no history) and the fields declared in the class `Point` contain the object's current values, as shown in Fig. 4(a). An object with the compact layout is a *compact object*.

In our approach, transactional objects may be in any of these two layouts and they may swing back and forth between the two: A compact object is *extended* whenever it is changed by a transaction and, therefore, it needs to maintain more than one version of its fields; an extended object may be *reverted* (to a compact layout) whenever it has only a single version in its history.

To support the extension and reversion operations, we added two auxiliary methods to the class `VBox` – `snapshot()` and `toCompactLayout(Object)` – which are overridden by every class that inherits from `VBox`. The execution of `snapshot()` returns a clone of the object containing the current values of the object's fields. The method `toCompactLayout(Object)` is responsible for copying the fields of a given snapshot into the transactional object. The implementation for both methods is injected into every transactional class instrumented by Deuce through a specific *enhancer* that is responsible for this transformation [3].

With this approach of swinging back and forth between the two layouts, we intend to reduce both the memory and the performance overheads caused by the STM's metadata, but obviously there is a tradeoff here. Not only because extending and reverting objects has costs, but also because it may interfere with the rest of the STM operations.

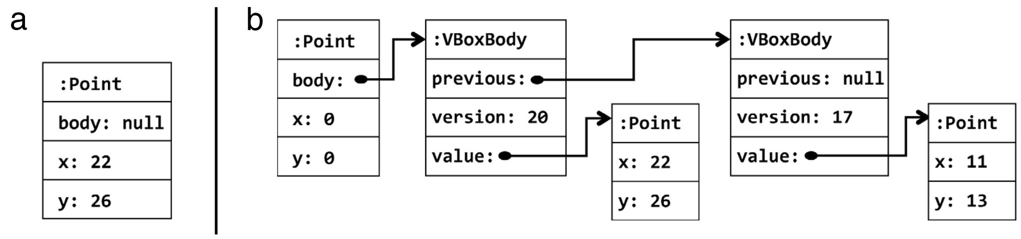


Fig. 4. An instance of the class `Point` in AOM, using either (a) the *compact layout*, or (b) the *extended layout*.

```
 1 // in class ActiveTransactionRecord:
 2 void clean() {
 3   for (Pair<VBox, VBoxBody> pair : this.allWrittenVBoxes) {
 4     pair.body.previous = null;
 5     tryRevert(pair.vbox, pair.body); // new call for AOM
 6   }
 7 }
 8
 9 boolean tryRevert(VBox vbox, VBoxBody body) {
10   synchronized (vbox) {
11     if (vbox.body == body) {                            // step 1
12       vbox.toCompactLayout(body.value);                 // step 2
13       compareAndSwapObject(vbox, bodyOffset, body, null); // step 3
14     }
15   }
16 }
```

Fig. 5. Algorithm to revert an object as part of the GC's clean task. We show in bold the line that was added to the `clean` method.

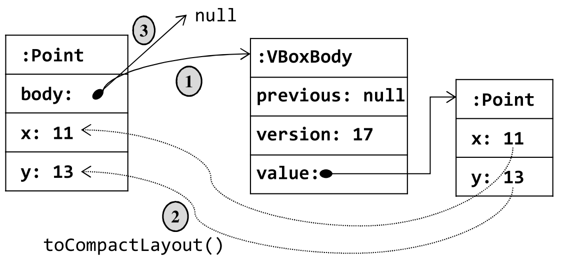


Fig. 6. Reverting an object that is in the extended layout and that stores the values 11 and 13 as the most recent, and only, committed values.

We designed our AOM operations – the extension and the reversion of an object – such that all of the JVSTM's operations preserve their progress guarantees. Namely, that reading a vbox and writing to a vbox are wait-free, and that committing a transaction is lock-free. Even though we use locks in the reversion of an object, the reversion is performed during the GC, which runs in separate threads, and, therefore, does not affect the rest of the transaction's operations.

4.1. *Reverting objects*

The process of reverting an object is plugged into the JVSTM's GC: When the GC trims an object's history and only a single version remains (the most recent, of course), it may revert the object to the compact layout.

In Fig. 5 we show both the `tryRevert` method that implements the reversion of an object and its call during the GC's `clean` method. Fig. 6 illustrates the process of reverting an object with a single version.

When the GC algorithm trims the history of an object (line 4), it calls the `tryRevert` method (line 5). Given that the JVSTM's GC may run in parallel with other threads, the `tryRevert` method may be called concurrently for the same object. Thus, it acquires the object's monitor (line 10) to prevent that more than one thread tries to revert the same object concurrently. Once the monitor is acquired, the method checks whether the head of the object's history is the `VBoxBody` that was just trimmed (line 11). If this condition is satisfied, which means that there is only one version in the history, it copies the values contained in that body to the corresponding fields in the object (line 12) and tries to CAS the `body` of the object to `null` (line 13). The CAS fails if the current value of the field `body` is no longer the body that was trimmed, which may happen only if one or more transactions commit new values for this object. So, when the CAS fails nothing else needs to be done, as the object cannot be reverted.

4.2. *Extending objects*

When an object is created, it is naturally in the compact layout. Moreover, when using LICM, the object will stay in the compact layout even when its owning transaction commits and publishes it. This is safe because the object is not shared until that transaction successfully commits, and, therefore, no other transaction may access the object before that.

So, when does a transactional object need to be extended? As mentioned before, we need to extend an object only when we need to have more than one version of the object, which happens only when a transaction writes a new value to any of the object's fields and successfully commits those changes.

Thus, the extension operation is plugged into the write-back phase of the commit: The object is extended during the write-back if it is in the compact layout. In this case, however, we cannot use locks in the extension operation if we want that the entire commit operation continues to be lock-free.

In Fig. 7 we show the new code for the write-back phase, which includes the code to extend objects if needed. When trying to add a new version to an object, the write-back operation must now check if the object is in the compact layout (line 6). If it is, then the previous version of the object's state is in the object's own fields, rather than in its (nonexistent) history. Thus, to add the new version to the previous version, the write-back operation must extend the object such that both versions are in the object's history (because after the extension no transaction will look into

```
 1 // in class VBox:
 2 VBoxBody commit(Object newValue, int txNumber) {
 3   VBoxBody currHead = this.body;
 4   if (currHead == null || currHead.version < txNumber) {
 5     VBoxBody newBody = null;
 6     if (currHead == null) {
 7       Object v0 = this.snapshot();                        // step 1
 8       VBoxBody body0 = new VBoxBody(v0, 0, null);        // step 2
 9       newBody = new VBoxBody(newValue, txNumber, body0);   // step 3
10     } else {
11       newBody = new VBoxBody(newValue, txNumber, currHead);
12     }
13     return CASbody(currHead, newBody);                 // step 4
14   } else {
15     return currHead.getBody(txNumber);
16   }
17 }

19 VBoxBody CASbody(VBoxBody expected, VBoxBody newBody) {
20   if (compareAndSwapObject(this, bodyOffset, expected, newBody)) {
21     return newBody;
22   } else {
23     // If the CAS failed then either the new value must already be there,
24     // or the object may have been reverted by a concurrent GC Task,
25     // in which case we need to retry the CAS to commit the new body.
26     // If the second CAS fails, then some other thread did the write-back.
27     VBoxBody currHead = this.body;
28     if (currHead == null) {
29       if (compareAndSwapObject(this, bodyOffset, null, newBody)) {
30         return newBody;
31       } else {
32         return this.body.getBody(newBody.version);
33       }
34     } else {
35       return currHead.getBody(newBody.version);
36     }
37   }
38 }
```

Fig. 7. New write-back algorithm of the VBox class including the extension. We show in bold the code that was added to the original code of the JVSTM.

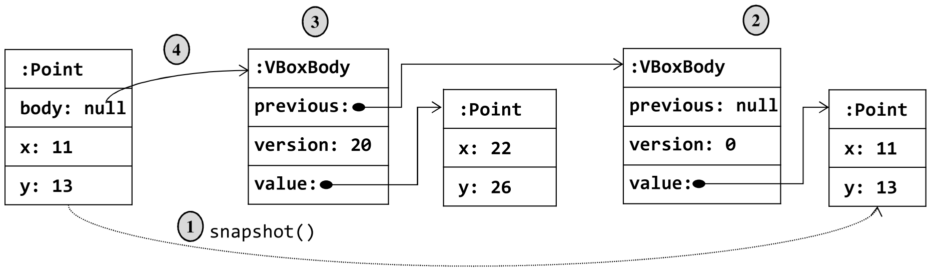


Fig. 8. An example of a transaction that commits the values 22 and 26 to the fields x and y, respectively, of a Point, which was in the compact layout and was storing the values 11 and 13 before.

the object's fields before it is made compact once again). This is accomplished by: (1) creating a snapshot of the object (line 7), (2) creating an entry for the snapshot that is marked with version 0 (line 8), and (3) creating the entry for the new value and version that points to the entry with version 0 (line 9). If the object is not in the compact layout, then the write-back works as before, by creating a new entry for the new value and version (line 11). In either case, the operation proceeds by calling the CASbody method (line 13) to update the history, which in the case of an extension corresponds to installing a new history. An example of a successful extension that goes through all these steps is shown in Fig. 8.

The method CASbody was also changed. Previously, it attempted to CAS to the new body only once and returned even if the CAS failed, because the only reason why the CAS operation could fail was if some other thread performed the write-back of an identical body, in which case there was nothing else to do. With AOM, however, there may be another reason for a failure of the CAS: A concurrent GC thread may revert the object back to its compact layout between the read of the object's body (line 3) and the attempt to CAS it to the new value (line 20); such a reversion changes the value of body to null and the CAS fails. But in this case the new value was not added to the object's history and, therefore, we need to try to add it again. This scenario is identified by reading again the object's body (line 27) and checking whether it is null (line 28), in which case the CAS is attempted once again (line 29) to swing the field body from null to the new body. If this second attempt of the CAS fails, then it is because this body was written back by some helper. We shall return to why this is so in Section 4.4, where we discuss the correctness of our AOM algorithms.

4.3. Reading objects

Given that AOM objects may be in one of two possible layouts, the read operation must take the layout into account when accessing an object. So, the first thing that a read operation does is to check whether the field body of the object is null. If it is, the read operation uses a fast path that directly accesses the other fields of the object—we refer to these as *proper fields*.

Consider the case of a thread Th_1 that is executing transaction Tx_1 with a reading version v_1. To read the proper field of an object *Ob*, Th_1 must find null at the field body. Otherwise, it goes through the history until it finds a version v, such as $v \leq v_1$ and, then reads the proper field of the corresponding snapshot. The latter case is the original algorithm of the JVSTM, which traverses only immutable data structures.

For read–write transactions we must still log the read operation in the read-set for objects in both layouts. But for read-only transactions we do not need to keep any log and, therefore, reading an object in the compact layout has almost no overhead over accessing directly an object's field without an STM barrier: We just have to check whether the body field is pointing to null or not.

4.4. Correctness of AOM operations

AOM is an extension to the JVSTM. JVSTM ensures the opacity correctness property (see Section 3), and a key goal of our work with AOM was to preserve that property. After presenting the algorithms for reverting, extending, and reading an object, we discuss now that the changes we introduce with the AOM preserve opacity by proving that each operation does not interfere with the outcome of the original JVSTM.

As in the original JVSTM, in AOM the objects representing a history entry – instances of the class VBoxBody – are immutable. They are created during the write-back phase of a transaction's commit and once created cannot be changed. Likewise for the snapshots, which are the instances of the transactional classes that are created to represent the value of an object at a particular version. Snapshots are created also during the write-back (which may include the extension of an object) and stored in the history entries before they are made available to other threads.

So, the only interesting cases that may be concurrently read and written by multiple threads are the fields of a transactional object. Whereas without AOM only the field body is used, with AOM the proper fields of a transactional object may also be read and written.

To help us reason about the correctness of AOM it is useful to make an observation about the JVSTM first. In the JVSTM, a thread Th_1 will help a transaction *Tx* to commit (e.g., by doing its write-back) only if Th_1 is itself executing a transaction Tx_1 (possibly *Tx*) and is waiting for its turn to commit. This means that Tx_1 must have started before *Tx* enqueue into the global queue of transactions that want to commit.

Observation 1. If *Tx* enqueues with the version v (i.e., its *committing version*), the version v_1 with which Tx_1 started (i.e., its *reading version*) is necessarily smaller than v (i.e., $v_1 < v$). Given this observation, we may now establish some results regarding AOM.

We start with a theorem establishing the impossibility of a late write-back.

Theorem 1. *A thread that arrives at the write-back of an object Ob (line* 3 *of Fig. 7) for some transaction Tx after at least another thread that completed the write-back for this body (line* 21 *or* 30*, of Fig. 7) will not change the object in any way.*

Proof. This result is trivially guaranteed in the original JVSTM because each helping thread reads the value of the field body before deciding whether it needs to do the write-back. Moreover, the body objects are immutable. So, if it finds an entry corresponding to the write-back that it is trying to do (i.e. an entry with the same version that it is trying to write-back), it returns without changing the object; otherwise it tries to install a new entry with a CAS. Yet, only the first attempted CAS will succeed and all others will fail because late threads will find the value of body changed since they first saw it. With AOM, however, the field body may swing back and forth between null and some other values (due to reversions). So, we need to check whether a late thread may find a null value in the field body and decide to add a new version, even though that version has been written before.

Assume that Th_1 is a late thread executing a transaction Tx_1, with the reading version v_1, in the write-back of *Ob* for *Tx* that commits with the version v (i.e. $v_1 < v$, from Observation 1). That means that some other thread Th_2 must have completed the write-back of *Ob* for the same transaction *Tx*, necessarily leaving in the field body of *Ob* a history containing at least two versions: the version v and its previous version. So, for Th_1 to find a null value in the field body of *Ob* after the write-back done by Th_2, *Ob* must have been reverted in the meanwhile. Yet, if Tx_1 is executing, *Ob* cannot be reverted, because it will need to keep at least two versions until Tx_1 finishes: the version v, which cannot be accessed by Tx_1 because $v > v_1$, and the previous version, which is accessible by Tx_1. □

Theorem 2. *If a thread Th_1 is executing the write-back of a transactional object Ob, then Ob is reverted at most once until Th_1 finishes the write-back, regardless of how long it takes.*

Proof. Let us assume that, while Th_1 is executing the write-back of *Ob*, there are two reversions for *Ob*. If that is the case, then *Ob* must have been extended in between the two reversions, while Th_1 is still running. Given that by Theorem 1 there are no late write-backs, that extension must have occurred as part of the write-back of *Ob* for some version v such that $v > v_1$ (where v_1 is the reading version of the transaction that Th_1 is still executing). So, at least until Th_1 finishes the execution of the write-back of *Ob*, *Ob* cannot be reverted again, because Tx_1 cannot access the version v of *Ob*. □

Theorem 2 justifies why the method CASbody in Fig. 7 tries the CAS only once after a failure of the CAS at line 20: The failure of the first CAS may be due to a concurrent reversion, but the failure of the second CAS cannot, because it is not possible to have two reversions for the same object while a write-back is in progress. A similar, simpler result is the following.

Theorem 3. *If a thread Th_1 executing a transaction Tx_1 reads a transactional object Ob and sees Ob in the compact layout, then Ob cannot be both extended and reverted until Th_1 finishes executing Tx_1.*

Proof. The reasoning for this proof is similar to the previous one. Assume that the reading version for Tx_1 is v_1. If *Ob* is extended after Th_1 sees it in the compact layout, then it must be because of a write-back to *Ob* that created a version v such that $v_1 < v$. So, if a reversion was to happen, then the history would be trimmed to keep only version v, which only happens if the ATR for v_1 is inactive, meaning that there is no transaction with that ATR's number. Given that Tx_1 is still running then its corresponding ATR is active and will prevent *Ob* from being reverted. □

Given these three theorems, we may now discuss the correctness of each of AOM operations.

4.4.1. Correctness of the read operation

Consider the case of a thread Th_1 that is executing transaction Tx_1 with a reading version v_1. If Th_1 finds the object *Ob* in the compact layout, then it must be because the most recent version of *Ob*, let us call it v_{ob}, is such that $v_{ob} \leq v_1$ and the proper fields of *Ob* contain the values corresponding to that version v_{ob}. Otherwise, if $v_{ob} > v_1$ then the transaction Tx_1 cannot read v_{ob} and should get a valid version v from the history of *Ob*, such as $v \leq v_1$, and thus, *Ob* cannot be reverted because it has a history with more than one version.

Moreover, by Theorem 3, we know that while Th_1 is executing Tx_1, *Ob* may be extended but it cannot be reverted again. Given that only the reversion writes into the proper fields of an object, we know that the values in the proper fields of *Ob* are the values that Th_1 needs to read and, thus, that it is safe for Th_1 to read them.

If, on the other hand, Th_1 finds *Ob* in the extended layout, it will be able to find the version that it needs in its history. To see why, consider the two possible scenarios, where the last version v written back to *Ob* by a transaction *Th* occurred: (1) before Tx_1 started, such as $v_1 \geq v$, or (2) after Tx_1 started, but before Th_1 attempted to read it, such as $v_1 < v$. In the first scenario, and because no reversion for *Ob* occurred in the meanwhile, then the head of the history contains the entry that Th_1 needs to read. In the second scenario, either *Ob* was already extended and the commit of *Th* simply added a new version to the head of the history, or *Ob* was extended by the commit of *Th*, creating not only a newer snapshot for version v but also a snapshot for version 0 with the values of *Ob*'s proper fields. In both cases, Th_1 will jump over the entries at the head of the history until it finds an entry with a version that it can read (possibly, version 0 that results from the extension). We shall see below why it is correct to create this version 0.

4.4.2. *Correctness of the reversion operation*

Only the reversion operation writes to proper fields, through the execution of the method `toCompactLayout`, which copies the values of a snapshot into the proper fields. Given that doing this copy requires acquiring the object's monitor, no two concurrent threads may be copying values back to the proper fields at the same time and, thus, after a reversion the values at the proper fields of an object will necessarily reflect a consistent set of values for some version (obtained from a given immutable snapshot).

Yet, while the copy is executing, the values in the proper fields may correspond to an inconsistent set of values, as some fields have already been copied while others have not. So, we need to ensure that while such a copy is in progress no reads of the proper fields are made.

With AOM, proper fields may be read only in two cases: (1) when reading an object during a transaction, and (2) when extending an object during the write-back (by the execution of the method `snapshot`). In both cases, the proper fields are read only if the field `body` was found to be `null`.

Moreover, the reversion operation tries to change the field `body` back to `null` only after copying all values from a snapshot into the proper fields (see lines 12 and 13 in Fig. 5). This is done with a tentative CAS of `body` from the entry containing the snapshot to `null`. This reset of the field `body` may fail if another thread concurrently commits a new value to this object, in which case it installs a new entry in `body`. In this case, the copy of the snapshot back into the proper fields was in vain because the object may no longer be reverted. Still, the values copied are consistent after the copy. But, more importantly, given that `body` was never `null`, no read of the proper fields may have occurred during the copy. So, reversions are guaranteed to occur only when they are safe with regard to potential concurrent reads.

4.4.3. *Correctness of the extension operation*

The behavior of the write-back of an object when the object is in the extended layout was not changed with AOM. So, we just need to consider the case of doing a write-back for an object that is in the compact layout.

We have two scenarios to consider: (1) when the write-back operation first sees the object extended but the object is reverted during the write-back, and (2) when the write-back operation sees the object already in the compact layout.

In the first scenario, the write-back obtains the object's current history and creates a new entry to add at the beginning, as in the normal case. Yet, when it tries to install the new history the CAS fails because the object was reverted in the meanwhile. Nevertheless, the newly computed history is still correct and may be installed in the `body`, effectively extending the object again. This is accomplished by the second CAS in the method `CASbody`. As discussed before, after this second CAS it is guaranteed that the new version has been installed in the object.

In the second scenario, the write-back operation needs to create a snapshot of the object to capture the current values of its proper fields. This snapshot is tagged with version 0 and added to the history, just after the newly created version. As we saw before, this snapshot is guaranteed to obtain a consistent view of the object's state. The version 0 is used because an object in the compact layout has no information about which version its values correspond to. Still, we know that it must be the case that no previous version would be needed by any of the running transactions (or else the object would not be in the compact layout), and, thus, we may use any version that is lower than the oldest running transaction; version 0 satisfies trivially this constraint.

5. Performance evaluation

To evaluate the performance of our approach, we compare the performance of the JVSTM in four different scenarios: (1) the original unmodified implementation of the JVSTM (*jvstm*), (2) the JVSTM enhanced with AOM (*jvstm-aom*), (3) the JVSTM enhanced with LICM (*jvstm-licm*), and (4) the JVSTM enhanced with both techniques (*jvstm-aom-licm*). For comparison, we include in our analysis other STM enhanced with LICM: the LSA [25]. The implementation of LSA is available with the original Deuce framework distribution [19], whereas the JVSTM was integrated in the modified version with LICM [3].

We evaluate the performance of the STMs in STMBench7 benchmark [12] and in four applications of the STAMP benchmark [2]: Vacation, Intruder, Kmeans and Ssca2. We also ran a sequential non-instrumented version of each application for comparison (*1-thread-unsync*). Given that STMBench7 has a medium/fine-grained locking synchronization strategy, we also compare the performance of the lock-based approach with the STM-based approaches.

All tests were executed on a single machine with a NUMA architecture containing 4 AMD Opteron^{TM} 6168 processors, each one with 12 cores, resulting in a total of 48 cores. Furthermore, the system has 128 GB of RAM. The JVM version used was the 1.6.0_33-b03, running on Ubuntu with Linux kernel version 2.6.32. The applications were executed with the JVM configured as a server-class machine with the initial, minimum heap size of 8192 MB and using the parallel GC, which uses an adaptive heap size policy to dynamically adjust the size of the heap. During the measurements there were no full garbage collections.

5.1. *STAMP benchmark*

We start our analysis with the Vacation application, which emulates a travel reservation system where *customers* concurrently make requests that affect some of the application's *items* (such as flights and cars): Each request from a customer is composed of operations – such as reserving a car, or canceling a reservation – that

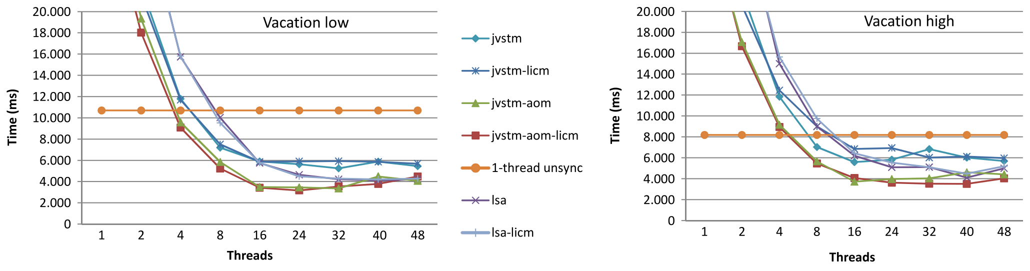


Fig. 9. The results for Vacation in the two workloads (low and high contention) with LSA, JVSTM and a sequential non-instrumented version of Vacation.

must be performed atomically. Because all the operations that take place in a request include at least some transactional writes, this application does not take advantage of read-only transactions. The application holds the global items in red–black trees, one for each type of item, including the customers themselves. On the other hand, each customer contains a list that points to each resource that it reserved.

We ran this benchmark with the configurations proposed by Cao Minh et al. [2], but because we want to emulate the behavior of large-scale programs, we used larger data sets. So, instead of the proposed value for the parameter *n* (between 2 and 4), which specifies the number of *items* operated by session and that is directly related to the transaction's length, we used a higher number of 256 items. Thus, for the low contention scenario we used the parameters "-n 256 -q 90 -u 98 -r 262144 -t 65536", whereas for the high contention scenario we used the parameters "-n 256 -q 60 -u 90 -r 262144 -t 65536".

We present in Fig. 9 the results obtained for the two workloads of the Vacation benchmark with the various STMs and the sequential non-instrumented version of Vacation. As we can see, in this benchmark, the LICM shows no benefits. This was expected, because the Vacation benchmark does not present any opportunity for elision of transaction-local barriers and, thus, we cannot observe any improvement in performance of JVSTM or LSA when they are enhanced with LICM. Yet, these results show the lightweight nature of LICM: Its penalty on the performance of the JVSTM (either with or without AOM) is just below 10% in the worst case and is negligible in most cases.

In Vacation, the *items* operated by each transaction are randomly selected by each *customer*. Thus, because we use large data sets, there is a low probability that two consecutive transactions performed by the same *customer* (or even two concurrent transactions) update the same *items*. Furthermore, all transactions need to traverse the global structures containing the application items to find the selected resources. These lookups perform many memory reads over transactional objects that are seldom changed. So, we expect to be able to improve the performance of the benchmark if we keep transactional objects in a compact layout, thereby promoting the use of lightweight STM barriers. In fact, there is an advantage in reverting the transactional objects to the compact layout whenever possible to accelerate the lookup phase of each transaction and therefore improve the overall performance of the benchmark. This is the reason why AOM improves the performance of the *jvstm* and the *jvstm-licm* in both workloads of the Vacation: As shown in Fig. 9, both *jvstm-aom* and *jvstm-aom-licm* reduce the time taken to execute the benchmark to half of the time taken by *jvstm*.

Next we analyze the behavior of JVSTM and LSA with our optimizations techniques in other 3 STAMP applications: Intruder, Kmeans and Ssca2. Intruder scans network packets for matches against a known set of intrusion signatures; K-Means implements k-means clustering; Ssca2 is comprised of four kernels that operate on a large, directed, weighted multi-graph (this benchmark just allows the execution of a number of threads with a multiple of two). We ran these applications with the configurations proposed by Cao Minh et al. [2]: for Intruder, "-a 10 -l 128 -n 65536 -s 1"; for KMeans, "-m 15 -n 15 -t 0.00001 -i random-n65536-d32-c16.txt"; for Ssca2, "-s 13 -i 1.0 -u 1.0 -l 13 -p 3". The three applications have relatively short transactions and low levels of contention. Moreover, all three applications have a low amount of total transactional execution time.

We present in Fig. 10 the results obtained with each STM and the sequential non-instrumented version. As we can see, the sequential version of each application performs almost always better than any of the others synchronization techniques. Overall, for the three applications, relatively little of the total execution time is spent in transactions. Nevertheless, the LSA enhanced with LICM performs better than the sequential version of Intruder for more than 4 threads. The speedup we observed in Intruder is consistent with the results of Dragojevic et al. [9], which provide evidence for some opportunities of elision of transaction-local barriers. We can observe the same effect of LICM on the speedup of JVSTM in the Intruder, but it is not enough to achieve the performance of the sequential version. In fact, the JVSTM is better suited for programs with a large number of transactional objects and long transactions, where the overheads of the multi-versioning approach are dissipated by the efficiency of the read operations. In the same way, the AOM has benefits in the reduction of the overheads of additional metadata, which become negligible for these three applications, and thus, we cannot observe any speedup of the JVSTM. However, we can still confirm that the AOM has almost no overhead and it does not degrade the JVSTM performance in these cases.

Neither K-Means nor Ssca2 access transaction local memory and therefore, there are no opportunities for eliding barriers with LICM. Thus, we cannot observe any speedup of LSA or JVSTM when enhanced with LICM. However, my results show that LICM has almost no overhead and that it degrades performance in 10%, in the worst case.

5.2. STMBench7 benchmark

STMBench7 is a benchmark for evaluating the performance of STMs on a model of a large scale CAD/CAM application. Its data structure consists of a large graph of different kinds of objects and its operations manipulate large parts of this data structure. Several of STMBench7's operations traverse a complex graph of objects by using iterators over the collections that represent the connections in that graph. Typically, these iterators are transaction local and, thus, accessing them using STM barriers adds unnecessary overhead to STMBench7's operations. As shown in the work of Carvalho and Cachopo [4], the majority of the barriers elided by LICM in STMBench7 access instances of classes related to the iterators of the `java.util` collections. So, it is no surprise that using capture analysis has a great influence in the performance of STMBench7 with Deuce, as we can verify in the results shown in Fig. 11, where we can see that LICM improves the performance of any of the enhanced STMs.

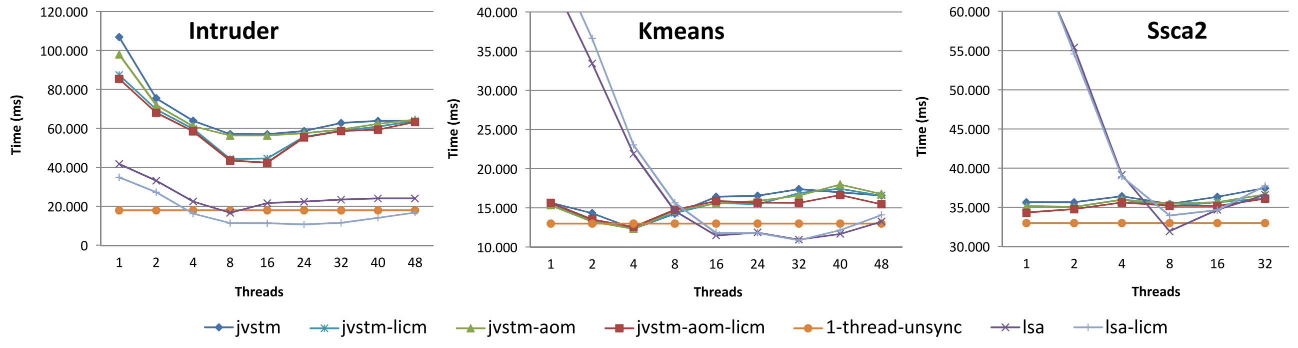


Fig. 10. The results for Intruder, Kmeans and Ssca2 (this one just allows the execution with a multiple of 2 threads), with LSA, JVSTM and a sequential non-instrumented version of each application.

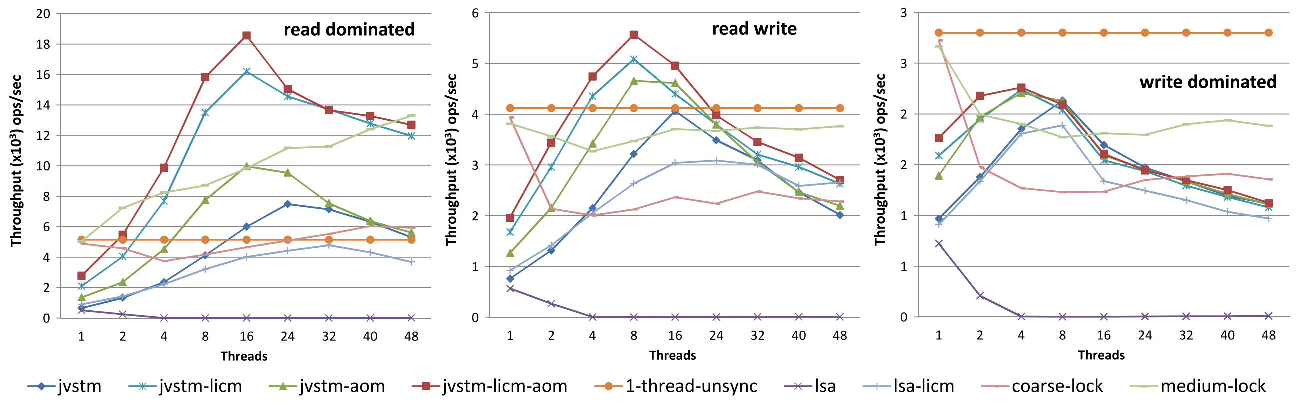


Fig. 11. The results for STMBench7 in the three available workloads without long traversal operations, for LSA, JVSTM, two lock-based strategies and a sequential non-instrumented version of STMBench7 (1-thread-unsync).

LSA and JVSTM present the best performance in STMBench7, when compared to TL2, because of their versioning approach, which allows read-only transactions to get a valid snapshot of memory. This effect is amplified by the fact that only with LICM can we have read-only transactions. Without LICM, most of the read-only transactions are forced to be executed as read–write transactions because they need to use write barriers when using iterators, which perform modifications to their own state. This problem is aggravated by the eager ownership acquisition approach followed by LSA, which acquires a lock for every written location. All this together contributes to an extremely high rate of aborts that drastically reduces the performance of the LSA for more than 4 threads. Once the useless barriers are elided with capture analysis, the LSA scales for an increasing number of threads, getting an improvement of up to 12-fold in performance [4]. In the case of the JVSTM, the LICM improves the performance of the original algorithm by 2.5 times, which does not share the same problems of the LSA due to its lock-free commit algorithm and the lazy ownership acquisition approach.

As in the case of Vacation benchmark, in STMBench7 the AOM can also improve the performance of the JVSTM when the traversal operations are dominated by read accesses. In a read dominated workload the AOM can double the throughput of the baseline JVSTM. On the other hand, when we increase the update rate, the benefits of AOM are reduced because of the overhead incurred by the extension of transactional objects. If we have too many objects being extended and reverted back consecutively between the two layouts, then the benefits of the lighter read barriers cannot overtake the overheads of the layout transitions. Yet, the results show that even in the write-dominated workload the AOM adds only a residual overhead that does not degrade the performance of the JVSTM.

Finally, when we combine LICM with AOM, we can observe that LICM can take one step further and still improve the performance, achieving a throughput up to 2 times higher than *jvstm-aom* and up to 3 times higher than *jvstm*. In fact, the *jvstm-aom-licm* is the best synchronization approach in STMBench7 (except for the write-dominated workload) and gets even better results than the medium-lock synchronization approach.

6. Related work

The *multi-versioning* design was introduced by Reed [24] in the context of distributed synchronization and was firstly used in the JVSTM [1]. The same idea was subsequently adopted by the LSA [25], which uses locators (indirection between object reference and object's content) based on the design of DSTM [15]. More recently, the SMV [23] also implements *multi-versioning*, with a design similar to TL2 [8], from which they borrow the ideas of *lazy ownership acquisition* of updated objects and a *global version clock* for consistency checking.

Perelman et al. [23] studied the SMV behavior using STMBench7 and Vacation, as we did in our evaluation, and they compared SMV to TL2. They observed that in the read-dominated workload of STMBench7, SMV's throughput was seven times higher than that of TL2. These results are consistent with our observations of the original JVSTM algorithm in the same STMBench7's workload, which gets an improvement of up to 12-fold in the throughput in comparison to TL2. Yet, adding LICM and AOM on top of the JVSTM allowed us to get a 4-fold improvement over JVSTM, which let us conclude that the *jvstm-licm-aom* is the best performing STM. In fact, if we compare TL2 with *jvstm-aom-licm*, we observe that *jvstm-aom-licm*'s throughput is 45 times higher than that of TL2.

With respect to the results presented for the Vacation by Perelman et al. [23], SMV performs always worse than TL2, whereas in our results both JVSTM and *jvstm-aom-licm* perform better than TL2.

The key insight of the AOM approach that the multiple versions are only needed under contention scenarios is inspired in the same idea of the work of Marathe and Moir [21], which claim that more expensive metadata management is only necessary in situations of conflict with unresponsive transactions. Yet, their work is for a word-based STM implementation that resorts to a table of ownership records to associate metadata to each memory location. NZTM [27] is based on the same principle, but instead of the table of ownership records, they use the object headers to attach the indispensable metadata to every object. This technique is similar to ours, but even in the compact layout they still require three additional slots for the pointers to the transaction's descriptor, the `ReaderList` data structure, and the old data copy. In contrast to this, our solution requires only a single word that contains a `null` value when the transactional object is in the compact layout.

Dalessandro et al. [7] also tackle the problems induced by the use of ownership records (*orecs*) and they propose a new STM implementation called NOrec that abolishes their use. Rather than logging the addresses of *orecs*, transactions log the addresses of the locations and the values read. Validation consists of re-reading the addresses and verifying that locations have not been committed by a concurrent transaction since they were read. Yet, this approach does not share the benefits of a multi-versioning design, which promotes lightweight barriers for read-only transactions.

7. Conclusions and future work

For small-scale applications it is, in general, easy to find a simple fine-grained, lock-based synchronization solution that is simultaneously correct and high-performant. The problem with shared data synchronization in concurrent programming arises when programs become too large and it is not trivial to design such a fine-grained locking solution. That is where STMs promise to make a difference.

Yet, even though STMs present some attractive features, in large-scale applications they often perform worse than a coarse-grained lock approach, thus limiting their usefulness for this type of applications, where they would matter most.

Our experience in testing STMs with several large benchmarks, however, is that the problem stems from having instrumentation on memory locations that are not actually shared among transactions. In this work we propose a solution that explores this fact to reduce the overheads that are typically caused by useless STM barriers and metadata. Moreover, we implemented our proposal as an extension of the JVSTM and integrated it in the Deuce STM, to allow a fair comparison with other STMs.

Even though we presented AOM as an extension of the JVSTM, it should be possible to adapt it to other multi-versioned STMs, as it requires little changes to the JVSTM's algorithms—basically, some way to plug the extension and reversion operations.

The results of our experimental evaluation show that our approach can significantly improve the performance in many large-scale applications, while preserving the transparency of an STM API, as shown with its implementation in the Deuce framework. Although our technique adds a minor overhead in memory space to all transactional objects, we still get a huge speedup in the Vacation and the STMBench7 benchmarks. In fact, for the first time in the case of STMBench7, we were able to get better performance with an STM than with the medium-grained lock strategy.

Finally, our experimental results confirm our expectations that it is feasible to use STMs for large applications, provided that the STM is not adding barriers to unnecessary memory locations. In fact, we believe that integrating LICM and AOM together in a managed runtime may further reduce the overhead of our approach and provide a significant boost in the usage of STMs.

Acknowledgment

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and under project PTDC/EIA-EIA/108240/2008 (the RuLAM project).

References

[1] J.a. Cachopo, A. Rito-Silva, Versioned boxes as the basis for memory transactions, Sci. Comput. Program. 63 (2006) 172–185. URL: http://portal.acm.org/citation.cfm?id=1228561.1228566.
[2] C. Cao Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: IISWC'08: Proceedings of The IEEE International Symposium on Workload Characterization, September 2008, pp. 35–46.
[3] F.M. Carvalho, Deuce STM with LICM and support for Enhancers, 2012. URL: http://inesc-id-esw.github.io/deucestm/.
[4] F.M. Carvalho, J. Cachopo, Lightweight identification of captured memory for software transactional memory, in: J. Kolodziej, B. Martino, D. Talia, K. Xiong (Eds.), Algorithms and Architectures for Parallel Processing, in: Lecture Notes in Computer Science, vol. 8285, Springer International Publishing, 2013, pp. 15–29.
[5] F.M. Carvalho, J. Cachopo, Runtime elision of transactional barriers for captured memory, in: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'13, ACM, Shenzhen, China, 2013, pp. 303–304. URL: http://doi.acm.org/10.1145/2442516.2442556.
[6] C. Cascaval, C. Blundell, M. Michael, H.W. Cain, P. Wu, S. Chiras, S. Chatterjee, Software transactional memory: Why is it only a research toy? Queue Concurr. Probl. 6 (5) (2008) 46–58.
[7] L. Dalessandro, M.F. Spear, M.L. Scott, NOrec: streamlining STM by abolishing ownership records, in: Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP'10, ACM, Bangalore, India, 2010, pp. 67–78. URL: http://doi.acm.org/10.1145/1693453.1693464.
[8] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: Proceedings of the 20th International Conference on Distributed Computing, DISC'06, Springer-Verlag, Stockholm, Sweden, 2006, pp. 194–208.
[9] A. Dragojevic, Y. Ni, A.-R. Adl-Tabatabai, Optimizing transactions for captured memory, in: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'09, ACM, Calgary, AB, Canada, 2009, pp. 214–222.
[10] S.M. Fernandes, J. Cachopo, Lock-free and scalable multi-version software transactional memory, in: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP'11, ACM, San Antonio, TX, USA, 2011, pp. 179–188.
[11] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'08, ACM, Salt Lake City, UT, USA, 2008, pp. 175–184. URL: http://doi.acm.org/10.1145/1345206.1345233.
[12] R. Guerraoui, M. Kapalka, J. Vitek, STMBench7: a benchmark for software transactional memory, in: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys'07, ACM, Lisbon, Portugal, 2007, pp. 315–324.
[13] T. Harris, K. Fraser, Language support for lightweight transactions, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA'03, ACM, Anaheim, California, USA, 2003, pp. 388–402. URL: http://doi.acm.org/10.1145/949305.949340.
[14] T. Harris, M. Plesko, A. Shinnar, D. Tarditi, Optimizing memory transactions, in: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'06, ACM, Ottawa, Ontario, Canada, 2006, pp. 14–25.
[15] M. Herlihy, V. Luchangco, M. Moir, A flexible framework for implementing software transactional memory, ACM SIGPLAN Not. 41 (2006) 253–262. URL: http://doi.acm.org/10.1145/1167515.1167495.
[16] M. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lock-free data structures, in: Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA'93, ACM, San Diego, California, United States, 1993, pp. 289–300. URL: http://doi.acm.org/10.1145/165123.165164.
[17] C.A.R. Hoare, The Origin of Concurrent Programming, Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 231–244. (Chapter Towards a theory of parallel programming). URL: http://dl.acm.org/citation.cfm?id=762971.762978.
[18] T. Knight, An architecture for mostly functional languages, in: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP'86, ACM, Cambridge, Massachusetts, United States, 1986, pp. 105–112. URL: http://doi.acm.org/10.1145/319838.319854.
[19] G. Korland, Deuce STM, 2010. URL: https://sites.google.com/site/deucestm/.

[20] G. Korland, N. Shavit, P. Felber, March 2010. Noninvasive concurrency with Java STM, in: Electronic Proceedings of the Workshop on Programmability Issues for Multi-Core Computers. MULTIPROG, Pisa, Italy, p. 10.
[21] V.J. Marathe, M. Moir, Toward high performance nonblocking software transactional memory, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'08, ACM, Salt Lake City, UT, USA, 2008, pp. 227–236.
[22] P. McKenney, M. Michael, J. Triplett, J. Walpole, Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory, ACM SIGOPS Oper. Syst. Rev. 44 (2010) 93–101.
[23] D. Perelman, A. Byshevsky, O. Litmanovich, I. Keidar, SMV: Selective multi-versioning STM, in: Proceedings of the 25th International Conference on Distributed Computing, DISC'11, Springer-Verlag, Rome, Italy, 2011, pp. 125–140. URL: http://dl.acm.org/citation.cfm?id=2075029.2075041.
[24] D.P. Reed, Naming and synchronization in a decentralized computer system (Ph.D. thesis), Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.
[25] T. Riegel, P. Felber, C. Fetzer, A lazy snapshot algorithm with eager validation, in: Proceedings of the 20th International Conference on Distributed Computing, DISC'06, Springer-Verlag, Stockholm, Sweden, 2006, pp. 284–298.
[26] N. Shavit, D. Touitou, Software transactional memory, in: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC'95, ACM, Ottowa, Ontario, Canada, 1995, pp. 204–213. URL: http://doi.acm.org/10.1145/224964.224987.
[27] F. Tabba, M. Moir, J.R. Goodman, A.W. Hay, C. Wang, Nztm: nonblocking zero-indirection transactional memory, in: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'09, ACM, Calgary, AB, Canada, 2009, pp. 204–213. URL: http://doi.acm.org/10.1145/1583991.1584048.

Fernando Miguel Carvalho obtained his lecture degree in Electrical and Computer Engineering, in 1997, and Msc. degree in Networks and Computers Systems, in 2005, and received his Ph.D. degree in Computer and Software Engineering in 2014, from the Technical University of Lisbon, Portugal. He got started as software architect at Altitude Software, during 1997–2001. In 2001 he has joined Quatro SI, as manager of systems integration group. Since 2007, he is Assistant Professor at Telecommunications and Computer Engineering Department, at Polytechnic Institute of Lisbon and a researcher at Centro de Calculo and INESC-ID. His research interests include virtual execution environments and parallel programming.

João Cachopo is Assistant Professor at the Department of Computer Science and Engineering of the Instituto Superior Técnico (IST), University of Lisbon. He received his Ph.D. degree in Computer and Software Engineering from IST in 2007, is the leader of the Software Engineering Group at INESC-ID, and was until 2012 the Chief Software Architect of the FénixEDU project. His current research interests include transactional memories, parallel programming, web engineering, and software architectures. He led the development of the first real-world application of a Software Transactional Memory to a production system (the FénixEDU project).