Adaptive object metadata to reduce the overheads of a multi-versioning STM*

Fernando Miguel Carvalho[1,2] and Joao Cachopo[2]

[1] DEETC, ISEL/Polytechnic Institute of Lisbon, Portugal
mcarvalho@cc.isel.ipl.pt
[2] INESC-ID/Technical University of Lisbon, Portugal
joao.cachopo@ist.utl.pt

Abstract. Current Software Transactional Memory (STM) implementations typically incur both into memory overheads that are proportional to the number of transactional locations used by a program, and into performance overheads when accessing any of those transactional locations. Such overheads render the use of STMs for the entire heap of a real-sized application impractical.
In this paper we propose an STM implementation that reduces substantially both the memory and the performance overheads associated with transactional locations that are not under contention. Therefore, assuming that the number of transactional locations that may be under contention is only a small fraction of the total memory managed by a real-sized application, the overheads introduced by the STM become residual.
Our proposal is based on a multi-versioning, object-based STM that swings objects back and forth between two different object layouts: a compact layout, where no memory overheads exist, and an extended layout, used when the object may be under contention. We describe the architecture and memory model of the *adaptive object metadata*—AOM—and we show results that evidence that the AOM can improve the performance of the WormBench in read-dominated workloads and Lee-TM benchmark.

Keywords: Software Transactional Memory, Optimization

1 Introduction

In the past few years, the intense research on Software Transactional Memories (STM) advanced considerably not only the design and implementation of STMs but also the understanding of their shortcomings. Yet, the problems originated by the runtime overheads of STMs and the causes behind those problems are still far from solved.

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

One of the techniques that aims to improve the STM performance is the *multi-versioning* approach, which is used in some STMs such as the JVSTM [2], the LSA-STM [13], and the SMV-STM [12]. A *multi-versioning* STM has the benefit that read-only transactions never conflict with other transactions, but the drawback of typically requiring both more memory to store the multiple versions and extra indirections to access the value of each transactional location.

In this paper we propose a new *multi-versioning* STM design that aims to reduce these overheads for programs where the number of transactional objects that may be under contention is only a small fraction of the total number of transactional objects. The key insight that will allow us to eliminate most of the memory overheads is that the multiple versions are 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 real-sized applications the vast majority of objects are seldom written, the number of objects that need to have more than one version 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 in the memory overheads.

Our proposal is based on the idea of transactional objects having *adaptive object metadata*—henceforth AOM for short. AOM is an object-based design that follows the JVSTM general design, but it is adaptive because the metadata used for each transactional object changes over time, depending on how the object is accessed.

In its most compact form the metadata adds an overhead of a single word per object, but when an object is changed and several versions of it may be needed, the object is extended with extra metadata to represent the various versions of the object. This extension is not permanent, however, and the object may revert back to its most compact form. With this AOM approach we expect to be able to reduce not only the memory overheads imposed by the STM, but also its performance overheads.

We implemented a prototype of this approach using Java bytecode manipulation to enhance a transactional class with the AOM approach. We tested it with the JWormBench [17] and Lee-TM [1] benchmarks and the results obtained so far confirm our expectations: Under a read-dominated scenario, the performance of the AOM approach improves as much as 36% over the original JVSTM design.

The remainder of this paper is organized as follows. In the following section we give an overview of the key aspects of the JVSTM. Then, in Section 3 we describe the new AOM design and its memory model. Next, in Section 4, we describe our performance evaluation of the AOM approach. In Section 5 we discuss related work on efficient support for STMs, namely identifying some of the runtime overheads imposed by many implementations and also some solutions to mitigate those overheads. Finally, in Section 6 we conclude and discuss some future work.

2 JVSTM overview

The JVSTM is a Java library that implements a word-based and *multi-versioning* STM. It uses the core concept of *versioned boxes*, which can be seen as a replacement for transactional memory locations. Instead of keeping only a single value, a *versioned box* (instance of the `VBox` class) provides a container that keeps a sequence of values—the box's history—where each value is tagged with a version, as exemplified in Figure 1. The version associated with each value in the history of a *versioned box* corresponds to the number of the transaction that has committed that value. Each versioned value from the history of a *versioned box* is a *box body* (instance of the `VBoxBody` class).

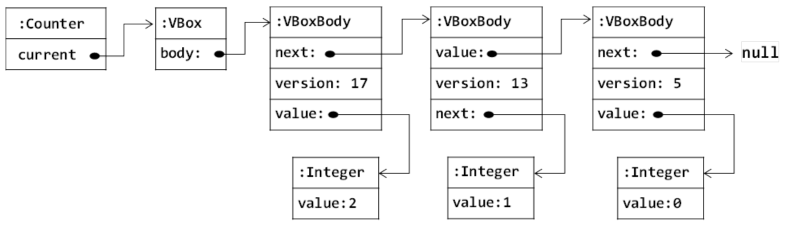

Fig. 1. Structure that represents a transactional counter and its versions, with a *versioned box* and three values in its history.

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. Read-write transactions, on the other hand, require validation at commit-time to ensure that values read during the transaction are still consistent with the current commit-time, i.e., that values have not been changed in the meanwhile by another concurrent transaction.

The JVSTM follows a *redo-log* approachand at commit-time all read-write transactions synchronize themselves on an exclusive global lock, thereby ensuring that the validation of the read-set and the writing-back of the write-set occur atomically. Also, the version number is provided by a unique global counter—`lastCommitted`—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.

3 The adaptive object metadata approach

The additional metadata that is associated with every transactional location is one of the causes behind the runtime overheads of a *multi-versioning* STM. The novelty of the AOM design 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.

In our current prototype of the AOM approach, we add to each transactional object a field that, in the compact layout, contains the value `null`: So, in AOM the per-object overhead of the compact layout is a single word. Yet, the idea is that on a JVM-level implementation of the AOM approach, that overhead may be further reduced by using some unused bits of the objects' header, effectively reducing the memory overhead of the compact layout to zero.

The extended layout is used when we need to have additional metadata associated with an object, such as multiple versions of the object. In that case, the previously mentioned field points to that metadata as shown in Figure 2: The transactional object's header points to the head of the versions' history. We say that an object in this state is an extended object.

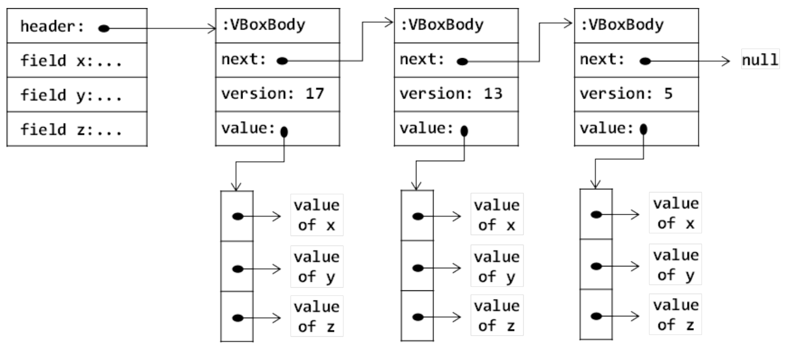

Fig. 2. Object's *extended layout* storage. The values of the object's fields lay out on `VBoxBody` array by the same order as instance fields on object's *compact layout*.

Considering only the extended layout, this design is identical to the design of the JVSTM with only two major differences: (1) the granularity of the conflict detection is at the object-level rather than at the word-level; (2) there is no need for the `VBox` object, because the extra field plays that role.

The major difference of AOM, however, is that after an object has been extended, and if no other running transaction needs the previous versions of the object's history, the object will be reverted back to the compact layout, which, naturally, involves copying the values of `VBoxBody` to the object's fields before setting the metadata field to `null` again.

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. So, we need to be careful about when and how to extend and to revert objects.

In the following two subsections, we describe when does AOM extend and revert objects. Then, we discuss informally the correctness of our approach by looking into the various scenarios of concurrent interleavings of the STM operations on a transactional object.

3.1 Extending objects

When an object is created, it is naturally in the compact layout because the extra metadata field contains `null`. Moreover, if no write is made to any of the object's fields, it will remain in this compact layout. This is fine because, assuming that the object is created within a transaction and, thus, that it is not shared until that transaction successfully commits, there is no possibility of another transaction accessing the object.

So, when does a transactional object need to be extended? As described 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.

So, AOM extends objects only during the commit of a write transaction, after validating the transaction, when it finds out that the object is still in the compact layout. During the write-back phase of the commit, when adding a new version to a transactional object, AOM must first check if the object is in the extended layout. If it is, then it just creates a new version and appends it at the head of the history, as JVSTM did. If the object is in the compact layout, however, it must extend the object and add the new version. This is done by executing the following three steps: (1) create a new `VBoxBody` instance tagged with version zero, containing the current values of the object's fields; (2) create a second `VBoxBody` instance containing the new values produced during the transaction and pointing to the previous `VBoxBody` instance; and (3) update the object's header so that it points to the previous `VBoxBody` instance, as depicted in Figure 3.

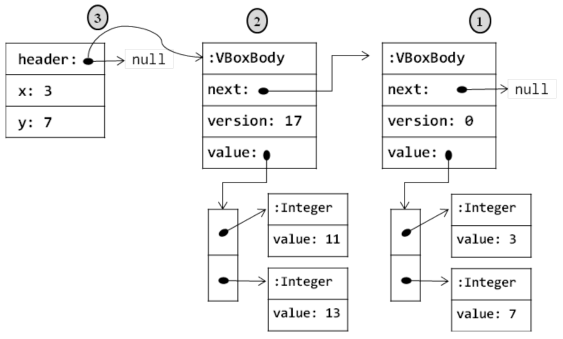

Fig. 3. An example of a transaction that commits the values 11 and 13 to the fields `x` and `y` of a transactional object that was in the compact layout and was storing the values 3 and 7 before.

3.2 Reverting objects

The process of reverting an object is plugged into the JVSTM's algorithm for garbage collecting (GC) old values from the objects' histories.

The JVSTM's GC algoritm ensures that a version in the history of an object is not removed as long as there may be some running transaction that may need to access it. So, once the GC decides to trim an object's history, it may happen that only a single version remains (the most recent, of course), in which case we may revert the object to the compact layout. To do this, the GC algorithm that trims an object's history after a certain `VBoxBody` first acquires that object's monitor and then executes the following steps: (1) read the head of the object's history and check that it is the `VBoxBody` that was just trimmed, executing the following two steps only if it is; (2) copy the values of the last committed body to the corresponding fields in the object; (3) do an atomic *compare and swap* to change the value of the object's header from its previously read value to `null`. Figure 4 shows an example of reverting an object.

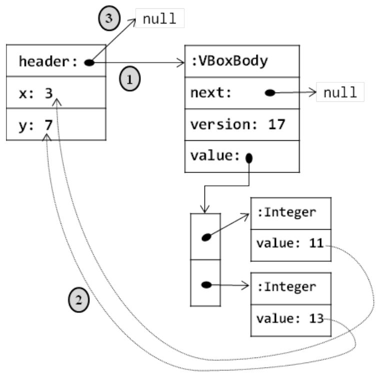

Fig. 4. Reverting an object that is in the extended layout and that stores the values 11 an 13 as the most recent, and only, committed values.

3.3 Correctness of the algorithm

After presenting the algorithms for extending and reverting an object, we discuss now, informally, the correctness of those algorithms while giving the rationale for our design along that discussion. To assess the correctness of our design, we need to reason about all of the possible interleavings of the concurrent execution of operations that may change or read a transactional object. Those operations are limited to the following: (1) extending the object, (2) reverting the object, and (3) reading a field of the object.

Concurrently extending an object

Starting with the operation of extending an object, it is easy to see that it is not possible to have two such operations executing concurrently, because they occur only during the write-back phase of a commit, which in the JVSTM is protected with a single global lock: So, only one at a time may execute.

Concurrently reverting an object

Similarly for the operation of extending an object, because that operation acquires a lock over the object to revert. Note, however, that in this case we may have concurrent operations of reverting an object, provided that they are reverting different objects. The GC algorithm of the JVSTM, on which the revert operations occur, is parallel: It may be executed as part of the commit of any transaction, even of read-only transactions. In fact, it may happen that two threads decide to revert the same object concurrently, and that is why we need to acquire the lock over the object to revert before reverting it. With the lock, only one thread will be able to revert the object, because when the first thread releases the lock and the second thread acquires it, this latter thread will not find its `VBoxBody` as the head of the object's history.

Concurrently extending and reverting an object

A more interesting case occurs when one thread is trying to revert an object and another thread is trying to extend that same object, which may happen in our design. Actually, we could prevent this scenario by acquiring the lock over the object when extending it, but we preferred to avoid that costly synchronization mechanism, which would increase the duration of the critical region within the commit of a transaction. Instead, in our approach, no costly synchronization is needed to extend an object.

To understand why our current approach is correct, note that when we revert an object, we do not change the instances of `VBoxBody` that represent the object's history; we simply set the metadata field of the reverted object to `null`. Moreover, the old object's history is still valid, in the sense that it represents correct information about the committed values for the object; it just happens that it is not needed anymore.

So, imagine that one thread T_e is extending an object concurrently with another thread T_r trying to revert it. If T_e finishes all its steps before T_r reads the head of the object's history, then T_r will refuse to revert the object because it will see in the head of the history a different body. If, instead, T_r is able to run to completion before T_e reads the history's head, then T_e will see the object in the compact layout and will extend it as usual. Finally, the potentially problematic case is when the two threads move in parallel through their first steps: T_r sees the expected value at the history's head, and T_e sees that same value, because it reads it before T_r sets it to `null`. In this case, T_e decides that it does not need to extend the object (because the object is already in the extended layout) and simply adds another version at the head of the current history and sets the corresponding metadata field in the object; this will be done independently of what T_r does. If T_e finishes first, the CAS of T_r will fail, leaving the object in the correct state (in the extended layout with the new value at the beginning of its history). If, on the other hand, T_r finishes first, T_e overwrites the metadata field with the new history, effectively turning the object into the extended layout again.

Concurrently reading an object while it is being extended or reverted

To conclude this discussion of the correctness of our AOM approach, we just have to discuss what happens when a thread reads an object that is being concurrently changed by any of the previous operations. Such a read should always return a value that is consistent with the executing transaction, to preserve the opacity of the JVSTM design [8].

To read an object's field, a reading thread T reads first the metadata field. If the value read by T is `null`, then T reads the field directly from the object; otherwise, it goes through the object's history until it finds the correct version and then reads the corresponding value. All of this goes on potentially in parallel with other operations over the same object. So, lets see each possible case in detail.

First, assume that T finds `null` in the metadata field. This means that, at this time, there is only one single version of the object and the object's values are stored in its fields. And that is why T may safely proceed to read the object's fields directly. But what if, in the meanwhile, another thread T_e writes to this object? As we saw before, T_e will extend the object at commit time, but it will not change the object's fields; in fact, the object's fields are only written by a reverting operation, but that cannot happen because after the extension of the object by T_e, the object has now two versions that cannot be GCed while T is running. So, T will still be able to safely read the object's fields, no matter how long it takes to get there.

As a second scenario, assume that T finds the object in the extended layout, meaning that it found the object's metadata field pointing to an history of versions. In this case, T will go through this history to find the appropriate version. Because the objects in this history are immutable, T will be able to succeed in its read independently of what else is done to the object. Note that new versions of the object are appended at the beginning of this linked list of versions, and thus does not change the existing nodes in the list.

4 Performance evaluation

To evaluate the performance benefits of our approach, we used the JWormBench [17] and Lee-TM [1] benchmarks. All the tests were performed on a machine with two quad-core Intel Xeon CPUs E5520 with hyperthreading, resulting in 8 cores and 16 hardware threads, running Java HotSpot(TM) 64-Bit Server VM (1.6.0 22-b02) with up to 16 concurrent transactions.

We tested JWormBench with three different kinds of workloads according to the following characteristics:

1. $O(n)$ and *N-reads-1-write*: This workload excludes the operations that are more computational intensive with $O(n^2)$ complexity. The length of the read-set is equals to the size of the worm's head and the length of the write set is equals to one.
2. $O(n)$, $O(n^2)$ and *N-reads-1-write*: Equals to the previous configuration, but including operations with complexity $O(n^2)$.
3. $O(n)$, $O(n^2)$ and *N-reads-N-write*: Equals to the previous configuration, but the length of the write-set is equals to the size of the worm's head, inducing a higher contention to the transactional locations and a consequently higher abort rate.

In all workloads we used the following environment configurations: a *world* consisting of 1024 nodes and 48 *worms* with a head's size varying between 2 and 16 nodes, corresponding to a number of nodes under the worm's head between 4 and 256. In all configurations we maintain the proportion between read-write and read-only *worm operations* of 20-80%. The charts of Figure 5 depict the speedup of each STM—JVSTM and AOM—over sequential, non-instrumented code for the JWormBench and Lee-Tm (later we will talk about Lee-TM).

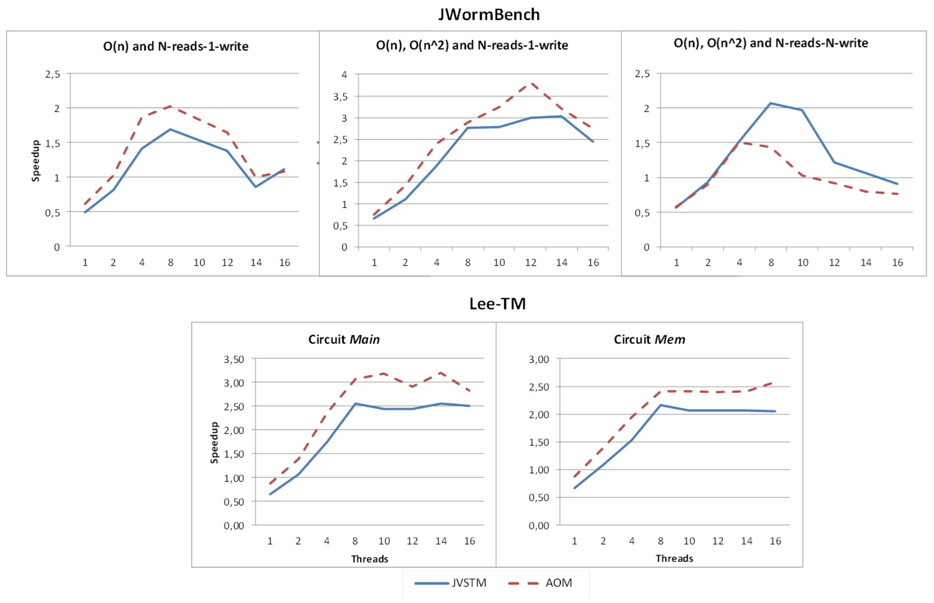

Fig. 5. Speedup of each STM—JVSTM and AOM—over sequential, non-instrumented code execution of the JWormBench and Lee-TM benchmarks.

In the first two scenarios of the JWormBench the AOM has always a better performance than the JVSTM and can improve the performance by as much as 36%. The less favorable scenario for AOM is when read-write transactions manipulate write-sets with the same length of the read-sets and equals to the number of nodes under the worm's head—between 4 and 256. In fact, these conditions do not favor a *multi-versioning* STM, because of the high number of updates per read-write transaction, which will substantially increase the time of committing a top-level write transaction, due to the creation of new `VBoxBody` instances. Adding to this, in AOM the cost of committing a new value into a transactional object that is still in the *compact layout* is bigger than in the JVSTM, due to the process of extending the layout of that object. That is the reason why the AOM performs worse than the JVSTM in such a workload.

The Lee-TM is a realistic and non-trivial benchmark suite for TM systems that uses Lee routing algorithm to automatically produce interconnections between electronic components. We tested Lee-TM with two different workloads based on two complex circuits called *main* and *mem* that are provided with this benchmark. In the charts of Figure 5 we can see that the AOM performs always better than the JVSTM in both workloads and increases the speedup between 13% and 35%.

Finally, we also analyzed the evolution of the allocated heap size during the execution of the JWormBench for the JVSTM and the AOM, as depicted in the figure 6. Again, in the third scenario we see no significant reduction of the memory overheads, because of the high number of write operations per read-write transaction. Yet, in the first two workloads we can reduce the allocated heap memory size in almost 75%.

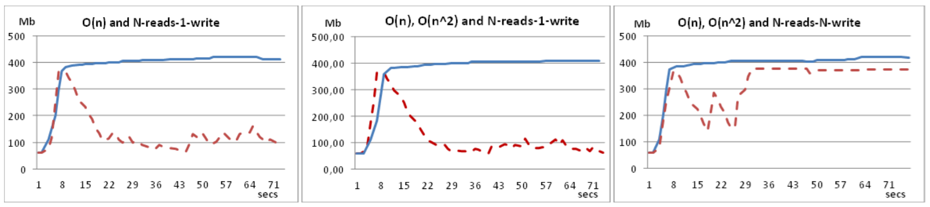

Fig. 6. Size of the heap used by the JVSTM (in solid blue) and the AOM (in dashed red) during the execution of the JWormBench.

5 Related Work

Among the proposed solutions to mitigate the runtime overheads of an STM, some follow an *undo log* approach (e.g., the *direct access* STM of Harris et al. [9] and the McRT-STM proposed by Saha et al. [14]), whereas others follow a *redo log* approach (e.g., those proposed in [6,15]). In fact, the better solution is closely dependent of the characteristics of the workload where it is evaluated, leading to different analysis and conclusions, as Dice and Shavit have described in their work [6].

The *global version clock*—GVC—is a technique that is adopted by some STMs [5,11] and mitigates the overhead of lookup the entire read-set. The values of the GVC establish a serial order between transactions. When a transaction commits it increments the GVC and attaches its value with each modified memory location. The location's version enables a transaction to detect a conflict in the moment it accesses a location.

The key insight of our work that the multiple versions are only needed under contention scenarios is inspired in the same idea of the work of Marathe and Moir [11], which claims 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.

The NZTM [16] is based in the same principle, but instead of the table of ownership records, they employ 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. Unlike this one, our solution requires only a single word that contains a `null` value when the transactional objects are in the compact layout.

Dalessandro et al [4] also tackle the problems induced by the use of ownership records—*orecs*—and they propose a new STM implementation — NOrec — that abolishes their usage. 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.

6 Conclusions and Future Work

We introduced an adaptive object metadata approach into an object-based design for a multi-versioning STM, and provided preliminary results that show an improvement in the performance of workloads where the number of objects written is much lower than the total number of transactional objects.

This work offers promising directions for a future implementation of an enhanced version of the JVSTM that reduces both the memory and the performance overheads due to the maintenance of the versioning histories. Yet, in any future implementation of the AOM, we plan to build on the most recent designs of the JVSTM, which are not yet included in the current version of AOM described in this paper. These new versions of the JVSTM, such as the new lock-free commit algorithm [7], show better performance, but the use of a lock-free commit algorithm requires a more in-depth analysis of the possible inter-leavings of concurrent layout transitions.

Moreover, our current AOM implementation still incurs into overheads due to the occurrence of some runtime verifications such as checking the objects' headers on every access to transactional objects and checking the kind of a transaction (read-only or read-write) in a transactional context. Our goal is to keep the contention-free execution path as simple as possible (and consequently fast), ideally with the same overhead of accessing non-transactional objects. So, we would like to be able to eliminate all the overheads of accessing objects in the compact layout in the scope of read-only transactions. This means that, in this scenario, the execution path should be clear of any STM barrier. To accomplish that, we plan to create four transactional twins of every function corresponding to different combinations of the previous conditions: Accessing objects in compact versus extended layout, and in the context of read-only versus read-write transactions.

Despite the lack of all these optimizations in the current implementation of AOM, the results collected thus far in JWormBench and Lee-TM are promising and give us confidence that we can reach even better performance through the integration of the previously referred optimizations.

References

1. Mohammad Ansari, Christos Kotselidis, Ian Watson, Chris Kirkham, Mikel Luján, and Kim Jarvis, *Lee-tm: A non-trivial benchmark suite for transactional memory*, ICA3PP '08, Springer-Verlag, 2008, pp. 196–207.
2. João Cachopo, *Development of rich domain models with atomic actions*, Ph.D. thesis, Instituto Superior Técnico, Technical University of Lisbon, 2007.
3. Fernando Miguel Carvalho and João Cachopo, *STM with transparent API considered harmful*, ICA3PP '11, Springer, 2011.
4. Luke Dalessandro, Michael F. Spear, and Michael L. Scott, *NOrec: streamlining STM by abolishing ownership records*, PPoPP '10, ACM, 2010, pp. 67–78.
5. D. Dice, O. Shalev, and N. Shavit, *Transactional locking II*, DISC, Proceedings of the 20th International Symposium on Distributed Computing, 2006, pp. 194–208.
6. Dave Dice and Nir Shavit, *What really makes transactions faster?*, 2006, Transact, Ottawa, Canada, 06.11.2006.
7. Sérgio Miguel Fernandes and João Cachopo, *Lock-free and scalable multi-version software transactional memory*, PPoPP '11, ACM, 2011, pp. 179–188.
8. Rachid Guerraoui and Michal Kapalka, *On the correctness of transactional memory*, PPoPP '08 (Salt Lake City, UT, USA), ACM, 2008, pp. 175–184.
9. Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi, *Optimizing memory transactions*, PLDI '06, ACM, 2006, pp. 14–25.
10. Jeremy Manson, William Pugh, and Sarita V. Adve, *The Java memory model*, POPL '05, ACM, 2005, pp. 378–391.
11. Virendra Jayant Marathe and Mark Moir, *Toward high performance nonblocking software transactional memory*, PPoPP '08, ACM, 2008, pp. 227–236.
12. Dmitri Perelman and Idit Keidar, *SMV: Selective Multi-Versioning STM*, TRANSACT '10: 5th Workshop on Transactional Computing, 2010.
13. Torvald Riegel, Pascal Felber, and Christof Fetzer, *A lazy snapshot algorithm with eager validation*, 2006, pp. 284–298.
14. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg, *McRT-STM: a high performance software transactional memory system for a multi-core runtime*, PPoPP '06, ACM, 2006, pp. 187–197.
15. Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott, *A comprehensive strategy for contention management in software transactional memory*, PPoPP '09, ACM, 2009, pp. 141–150.
16. Fuad Tabba, Mark Moir, James R. Goodman, Andrew W. Hay, and Cong Wang, *Nztm: nonblocking zero-indirection transactional memory*, Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (New York, NY, USA), SPAA '09, ACM, 2009, pp. 204–213.
17. Ferad Zyulkyarov and et al., *WormBench: a configurable workload for evaluating transactional memory systems*, MEDEA '08, ACM, 2008, pp. 61–68.