Available online at www.sciencedirect.com

ScienceDirect

Science of Computer Programming 63 (2006) 172–185

Science of Computer Programming

www.elsevier.com/locate/scico

Versioned boxes as the basis for memory transactions

João Cachopo*, António Rito-Silva

INESC-ID/Technical University of Lisbon, Rua Alves Redol n. 9, 1000–029 Lisboa, Portugal

Received 31 December 2005; received in revised form 1 May 2006; accepted 18 May 2006
Available online 4 August 2006

Abstract

In this paper, we propose the use of Versioned Boxes, which keep a history of values, as the basis for language-level memory transactions. Unlike previous work on software transactional memory, in our proposal read-only transactions never conflict with any other concurrent transaction. This may improve significantly the concurrency on applications which have longer transactions and a high read/write ratio.

Furthermore, we discuss how we can reduce transaction conflicts by delaying computations and re-executing only parts of a transaction in case of a conflict. We propose two language-level abstractions to support these strategies: the *per-transaction boxes* and the *restartable transactions*.

Finally, we lay out the basis for a more generic model, which better supports fine-grained restartable transactions. The goal of this new model is to generalize the previous two abstractions to reduce conflicts.

Keywords: Software transactional memory; Transactions; Conflict reduction; Multi-version concurrency control

1. Introduction

With the increased availability of multiprocessor computers, concurrent programming entered the realms of mainstream programming. Yet, most programming languages lack adequate abstractions for concurrent programming.

Mechanisms such as Java's `synchronized` keyword are useful to develop thread-safe single objects, but are of little help when various objects are involved. In these cases, to ensure consistency, we want that all operations execute atomically. However, ensuring atomicity for complex domains with lock-based mechanisms is difficult and highly error-prone.

Much of the recent work on *software transactional memories* (STM) [17,9,7,8] deals with this problem by introducing into the programming language abstractions such as transactions, which correspond to the atomic execution of a group of operations. With transactional memories, several transactions can run in parallel using no locks. Instead, either during the transaction or when the transaction commits, a check is made to ensure that the transaction has seen a consistent view of the shared memory. If that is not the case, then the transaction is restarted.

Obviously, STMs are most effective when the number of restarts is low compared to the overall number of transactions. Therefore, the number of transaction conflicts should be minimized. In this paper we concentrate on this problem and make the following contributions:

* Corresponding author.
E-mail addresses: Joao.Cachopo@inesc-id.pt (J. Cachopo), Rito.Silva@inesc-id.pt (A. Rito-Silva).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.05.009

- We propose the use of software transactional memories based on **versioned boxes,** which, unlike current approaches, avoids the restart of read-only transactions. A problem with current nonblocking approaches to STMs is that longer-running transactions have a higher probability of being restarted, leading eventually to the transaction's starvation. Using our approach, long-running transactions that only read never conflict with any other transaction.
- We propose the use of **speculative read-only transactions** as a means to minimize the overheads of software transactional memories in the common case of applications where read-only transactions largely outnumber write transactions. In our approach, it is beneficial to distinguish between **read-only transactions** and **write transactions**, because the former may have much less implementation overheads than the latter. Therefore, when there is a high probability of a transaction being read-only, we may optimize its execution by assuming that it is read-only and restarting it later if we find out that the assumption was wrong.
- To further reduce conflicts, we propose two strategies: (1) delaying computations, to avoid reading high-contention boxes, and (2) re-executing only the parts of a transaction that caused the conflicts. Using these two strategies it is possible to develop concurrency-friendly objects, which are designed specifically to remove high-contention points from a concurrent program.
- To simplify the implementation of the two strategies to reduce conflicts, we extend our STM model with two new abstractions: (1) the **per-transaction boxes**, which hold values that are private to each transaction, and (2) the restartable transaction, which allows methods to be re-executed at commit time if a conflict occurs because of them.
- Finally, we lay out the basis for a more generic STM model based on versioned boxes. The goal of this model is to support restartable transactions that generalize the two abstractions proposed to reduce conflicts.

In the following section we describe our proposal for STM based on *versioned boxes* and discuss its implementation as a Java library. Then, in Section 3 we discuss some scenarios of conflicting transactions and propose the two abstractions that help in the development of concurrency-friendly objects. Those two abstractions are compared and generalized in Section 4. In Section 5 we discuss some related work and in Section 6 we present the conclusions of the paper.

2. Memory transactions based on versioned boxes

The distinctive element of our approach to STM is the use of *versioned boxes* to hold the mutable state of a concurrent program. Versioned boxes can be seen as a replacement for memory locations [7] or transactional variables [8].

2.1. Model specification

Unlike conventional locations, which keep only a single value, a **versioned box** is a container that keeps a tagged sequence of values—the **history** of the versioned box. Each of the history's values corresponds to a change made to the box by a successfully committed transaction, and is tagged with the number of the corresponding transaction. For instance, if a box `B` is changed three times, in transactions numbered `5`, `23` and `87`, the history of `B` will have three values, each tagged with one of the previous numbers.

There are two operations that a thread can execute on a versioned box: (1) the **read operation,** `BoxRead(B)`, which returns the current value of box `B`, and (2) the **write operation,** `BoxWrite(B, v)`, which sets the current value of box `B` to `v`. The behavior of these operations and how they interact with transactions will be explained below. For now, it is important to note that each of these operations must execute in the context of some transaction. If the executing thread has no current transaction, then a new transaction is created immediately before and is committed immediately after the operation.

Transactions, as usual, serve to delimit blocks of operations that should execute atomically. A transaction is associated with exactly one thread – the thread that started it – and lasts until that thread either aborts or commits the transaction. A transaction `Tc` that starts in the context of another transaction `Tp` is a **nested transaction.** Moreover, we say that `Tp` is the **parent** of `Tc` and that `Tc` is a **child** of `Tp`. Transactions that have no parent are called **top-level transactions.**

When a transaction starts, it becomes the thread's **current transaction** until it finishes or another (child) transaction starts. When a transaction finishes, its parent, if any, becomes the thread's current transaction.

We say that a box `B` is read/written in the context of a transaction `T` when `T` is the current transaction of the thread executing a read/write operation on `B`.

Besides a reference for its parent, each transaction keeps the following information: (1) a transaction number, (2) a set of boxes that were read in the context of the transaction, and (3) a **local-values map** that maps boxes to values. To simplify the presentation below, given a transaction `T`, we will use the notation `T-number`, `T-readSet` and `T-localValues`, to refer to each of these values, respectively.

The transaction number represents a version number, which is used both to tag the values on histories and to read the appropriate values from versioned boxes. When a top-level transaction starts, its number is set to the number of the last successfully committed top-level write transaction. So, all the top-level transactions that start between two commits of write transactions have the same transaction number. When a nested transaction starts, its number is set to the number of its parent.

2.1.1. Reading and writing boxes

A write operation, `BoxWrite(B, v)`, executed in the context of a transaction `T` does not change `B`. Instead, it adds a mapping from `B` to `v` to the `T-localValues` map. This map corresponds to the private storage of transaction `T` and is used to keep track of the changes performed during `T`.

A read operation, `BoxRead(B)`, executed in the context of a transaction `T` should return the current value of `B`. Obviously, the expected behavior for this operation is that it returns the last value wrote to `B` in the context of `T`. To accomplish this behavior, the read operation searches for an existing mapping for `B` in `T-localValues`. If such a mapping exists, the corresponding value is returned. Otherwise, the search continues recursively on `T`'s parent, until either one mapping is found or no parent exists. If no mapping was found, then the value to return is obtained from the history of `B` and `B` is added to `T-readSet`. The value obtained from the history of `B` by the read operation executed in `T` is the value tagged with the highest number which is less than or equal to the number of `T`. As we shall see below when we describe the commit of a transaction, this was the last value wrote to `B` when `T` started.

2.1.2. Committing and aborting transactions

As transactions execute concurrently, accessing shared resources, they may conflict with each other. However, conflicts are detected only at commit time. When a conflict occurs, the commit of some transaction fails. The commit of a top-level transaction `T` may fail only if `T` is a read-write transaction. If `T` is either a read-only or a write-only transaction, then the commit will always succeed.

If `T` is read-only, then no boxes were changed in the context of `T`. Thus, we can safely assume that the transaction executed instantaneously when it started, because all the boxes read by `T` are read in the state they were when the transaction started. Even if several transactions committed during the execution of `T`, the changes made by those transactions are not visible to `T`. Therefore, `T` executed in a consistent view of the program state and so it can safely commit. Moreover, as nothing was changed, there is nothing to be done in the commit of a read-only transaction.

On the other hand, if `T` is a write transaction, then at least one box was changed during `T`, and the changes performed by `T` should become visible after its commit. Thus, to ensure that transactions are serializable, `T` can commit successfully if and only if none of the boxes in `T-readSet` changed after `T` started — in this case, we say that `T` is **valid.** Using this definition, a write-only transaction[1] is always valid, because it never obtains values from the history of a versioned box. So, its results will always be the same, no matter when it is executed, and we can assume that it executes instantaneously when it commits.

The commit of a valid top-level write transaction `T` first renumbers the transaction, so that `T`'s number is one greater than the last successfully committed write transaction. Then, each of the values in `T-localValues` is added to the corresponding history tagged with this new number. Of course, the check for transaction validity and these changes should be executed atomically.

The commit of a nested transaction is simpler, as it just propagates the changes made during the nested transaction to the parent's context. Specifically, when a nested transaction `Tc` with parent `Tp` commits, the elements of

[1] We say that `T` is a write-only transaction if for each box `B` read in `T` there is a previous write for `B` in `T`.

```
public class Transaction {
  public static void start();
  public static void abort();
  public static void commit();
}

public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

Fig. 1. The `Transaction` and `VBox` classes.

`Tc-readSet` are added to `Tp-readSet`, and, also, the mappings in `Tc-localValues` are added to `Tp-localValues`, overriding any existing mapping in the parent's map. The commit of a nested transaction always succeeds.

Finally, aborting either a top-level transaction or a nested transaction simply ends the transaction and does not have any effect on boxes. All the transaction's local-values, if any, are lost. Aborting a transaction always succeeds.

2.1.3. Garbage collecting old values

With versioned boxes, old values are not lost; they are kept in the history of the box. This is necessary so that running transactions can read the correct values even after later transactions committed new values to a box. However, as old transactions finish, older values become unreachable. So, unless we want to keep the whole history of changes, the unreachable history values can be safely garbage collected.

To capture this concept, we say that a transaction is **active** if and only if it is either the current transaction of some thread or the parent of an active transaction. Moreover, we say that a value `v` of a history `h` is **reachable** if and only if: (1) `v` is the most recent value in the history `h`; or (2) there is an active transaction that started before a value newer than `v` was added to history `h`.

2.2. Model implementation

We implemented our approach to STMs as a Java library [10], although the facilities we need from Java are available in most other object-oriented programming languages.

There are two core classes in the library – the `Transaction` class and the `VBox` class – as shown in Fig. 1. However, to simplify transaction demarcation, in our current implementation we provide also a method annotation: the `@Atomic` annotation. Classes that use this annotation are then post-processed to wrap the annotated methods' bodies with the boilerplate transaction calls.

Using this implementation, it is now easy to develop transactional objects. For instance, a simple transactional counter is shown in Fig. 2.

Note the use of a `VBox` to hold the counter state and the `@Atomic` annotation on the `inc()` method. Moreover, this transactional object can be used in larger scale transactions. If `inc()` is called in the context of another transaction, its body starts a nested transaction and, in the end, commits its changes back to the parent transaction.

In the following, we highlight some of the design decisions we made in our implementation.

2.2.1. Implementing versioned boxes

We implement versioned boxes using a variation of the Handle/Body idiom [2]. Each versioned box is separated into a handle and a series of bodies. The handle instance represents the versioned box's identity and its single field is a reference to its most recent committed body. Each body represents a particular version of the box's state. It has fields to represent the box's value, a version number and a reference to the body that maintains the previous version of the box's state. The version number in each body is the number of the transaction that committed the body. For instance, Fig. 3 shows the versioned box of a `Counter` object that was created during transaction number `4`, and that was incremented subsequently, in transactions numbered `8` and `13`.

```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  public long getCount() {
    return count.get();
  }

  public @Atomic void inc() {
    count.put(getCount() + 1);
  }
}
```

Fig. 2. A transactional `Counter` object.

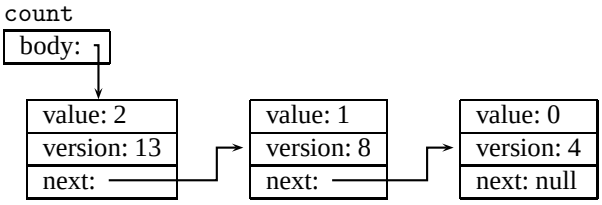

Fig. 3. The `count` box of a `Counter` instance.

Although a `VBox` keeps a reference to its body, none of the `VBox`'s methods, including the `get` and `put` operations, access the body directly. Instead, all accesses occur through a `Transaction` object, so that the correct body for the current transaction is used.

As we shall see, when a new value is written to a box, a new body is created by the transaction object, but it is not put into the box's history of bodies until the transaction commits.

As a final remark regarding the implementation of the versioned boxes, we note that both the `VBox` reference to its body, as well as all the slots in the body class are `volatile`. This is essential to ensure that assignments to these fields by one thread are written in the same order that the fields are assigned to in the thread. These fields are assigned only when a transaction commits a new body for an object. The last assigned field should be the `body` field in class `VBox`, so that if a concurrent thread reads the new value of this field, it obtains a body instance consistently assigned.[2] This way, threads that access the `body` field do not need to synchronize with any other.

2.2.2. Implementing transactions

In Fig. 4 we show part of the `Transaction` class's fields. The static field `lastCommitted` keeps the number of the last committed transaction, which is needed to give an initial number to each new top-level transaction. The value of this static field changes only when a top-level write transaction successfully commits, in which case it is incremented by one.

The `localValues` map corresponds to the transaction's private storage. The transaction uses this map to record the new bodies for each box written during the transaction. When, during a transaction, a new value is first written to a `VBox`, a new body is created with the new value and this new body is inserted into the `localValues` map. If, during the same transaction another value is written to the same box, then the previously created body is changed to reflect the new value. This is safe, because no concurrent thread can access this new body.

Naturally, when a box is read, the `localValues` map is consulted first to see whether a new body exists for the box. This search is performed recursively in the transaction's `parent` until a body is found or until no parent transaction exists. Only if no new body is found in any of the `localValues` map, will the transaction look for the appropriate

[2] This restriction is a consequence of the Java Memory Model [11].

```
class Transaction {
  static int lastCommitted = 0;

  int number;
  Transaction parent;
  Map<VBox,VBoxBody> bodiesRead = ...;
  Map<VBox,VBoxBody> localValues = ...;
}
```

Fig. 4. The member fields of the `Transaction` class.

body in the box's history. In this case, the transaction records in the `bodiesRead` map that the box public history was read. This will be needed at commit time to detect conflicts.

The search for the correct body in the public history of a box is a linear search in the sequence of bodies of that box. The search stops once a body with a version number less than or equal to the transaction's number is found. Given that the history is sorted in descending order of the bodies' version numbers, the search usually returns the first element in the sequence. Only if a concurrent transaction committed a new value for the box after the current transaction started, will the search need to go further in the sequence of bodies. Note that, even though the box may have already a new value, committed by a concurrent transaction, the current transaction may safely proceed if it is a read-only transaction. Only if we know already that the current transaction is a write transaction can we abort it when a read is performed on a concurrently changed box.

Finally, when the transaction commits, if the `localValues` map is empty, the transaction is read-only and the commit succeeds with no further actions. On the other hand, if the `localValues` map is not empty, the newly created bodies in it should be installed in the corresponding boxes histories, but only if the transaction is valid. The check for validity verifies that all the entries in the `bodiesRead` map correspond, still, to the most recent values.

According to our model, the check for the transaction validity and the installment of the new bodies should be performed atomically with respect to other concurrent commits. In our current implementation, this is accomplished by forcing all the write transactions' commits to synchronize on a single object, thereby ensuring the sequential execution of all commits. Note, though, that neither the execution of concurrent transactions nor the commit of read-only transactions are affected by this.

2.2.3. Speculative read-only transactions

The implementation we presented thus far registers all the boxes read during a transaction, so that the transaction can be validated in the end. This introduces a significant overhead, both in memory, to store all the map's entries, and in time, to update that map. Moreover, reading a box forces a search in the `localValues` map, again introducing a significant overhead. Fortunately, in some cases we can reduce these overheads.

Consider the case of a top-level read-only transaction `T`. As we saw, the commit of `T` will always succeed, and the `bodiesRead` map is not used for anything. Likewise, the `localValues` map is always empty, so the read operation on any box `B` obtains the result value from the history of `B`. Therefore, if we know beforehand that a transaction `T` is read-only, we can eliminate both of the overheads mentioned above. The problem is that, in general, it is not possible to know, until the end of the transaction, that it is read-only.

However, we can speculatively assume that a top-level transaction is read-only when it starts. If a write operation is attempted during the execution of the transaction, then we have to abort and restart the transaction as a generic read-write transaction. In programs where read-only transactions largely outnumber read-write transactions, the cost of restarting the erroneously assumed read-only transactions may be much less than the cost of unnecessarily using read-write transactions. One way to improve this strategy is to give the programmer the possibility to give hints about the nature of a transaction. Another, is to adaptively change the nature of a transaction in runtime based on prior executions of the same transactional block.

This speculative strategy can be applied to nested transactions as well, but requires some special considerations. If the parent transaction is not read-only, the nested transaction still has to register all the boxes read, so that it can add them to the parent at commit time. Therefore, we can use a nested read-only transaction only when the parent

is a read-only transaction. Moreover, when a write operation is executed in a nested read-only transaction we should restart the top-level transaction, rather than the nested transaction itself.

2.2.4. Implementing garbage collection

To conclude the discussion on the implementation of the versioned boxes model, we describe how we deal with the garbage collection of old values from the boxes' histories. As we saw in Section 2.1.3, old values are needed just as long as there are active transactions that may need to access them. Otherwise, when a new value is committed to a box, we could discard the box's previous value.

Our strategy to deal with this problem is to make the transaction that commits a new value for a box be responsible for the cleanup of old values. A transaction that commits new values already has, in its `localValues` map, the collection of boxes that need cleanup. We just have to know at what time can the cleanup be performed.

For that, we use a priority queue of active transactions, sorted by the transaction number, and a cleanup thread that removes inactive transactions from this queue.

Whenever a new top-level transaction starts, it is put into the queue, sorted by its number. When a transaction finishes, either by committing or aborting, it is marked as finished and the cleanup thread is signaled.

The cleanup thread waits until the transaction which is in the head of the queue (the oldest transaction) is finished. When that happens, the cleanup process removes all the transactions that are already finished from the head of the queue, calling the transactions' cleanup code as it proceeds. The cleanup of a write transaction consists in going through each of the bodies created by the transaction and discarding their previous versions (by setting the bodies' `next` field to `null`). Because the queue is sorted, when a write transaction is removed from the queue no older transaction is running, so the old values are unreachable and may be safely discarded.

3. Transaction conflicts

Now that we presented our proposal for STM, we will discuss some scenarios where transactions can conflict with each other. We start with a simple example that illustrates the benefits of our approach compared to non-versioned approaches. Then, we look into some examples which are hard to solve with current approaches (including ours). In these harder-to-solve examples, many conflicts occur because transactions access some high-contention objects. To solve this problem, we propose that these high-contention objects be reimplemented as concurrency-friendly objects, and show how that can be accomplished through the use of two new transactional abstractions.

We say that an object is concurrency-friendly when it is designed to be aware of and to cooperate with transactions, so that it reduces the number of conflicts on the transactions that use it.

3.1. Long-running transactions

Consider the typical example of a bank with several bank accounts, where we have two transactional methods: (1) a `transfer` method that transfers some amount between two accounts, and (2) a `totalBalance` method that calculates the total balance of all bank accounts.

Typically, different `transfer` transactions involve different accounts, so the probability of conflict between two such transactions should be rather low. However, the trivial implementation of the `totalBalance` method iterates over all the bank accounts to sum their balance. Thus, without a versioned model, any transfer during the execution of the `totalBalance` method would force this latter method to abort. Alternatively, all transfers (or any other change to bank accounts) should be suspended until the `totalBalance` method finishes. Neither option is satisfactory. Using versioned boxes as the basis for STM eliminates this problem, because the `totalBalance` transaction is a read-only transaction and therefore never conflicts.

3.2. Avoiding conflicts by delaying computations

Consider now that we want to keep a record of the total number of transfers made by the bank. An obvious solution would be to use a `Counter` object (as shown in Section 2.2) and increment the counter during each transfer transaction. Unfortunately, now all transfer transactions will conflict with each other, because all of them read and write the `Counter` state. The `Counter` object became a high-contention point in our system because it is shared by most transactions. Can we avoid this problem?

The increment of the shared `Counter` object on two transfer transactions causes them to conflict because the first transaction to commit changes the publicly available state of the `Counter` object, which is read by the other transaction in a previous state. We cannot prevent the first transaction from changing the counter, but we can prevent the second transaction from reading it in a previous state. To see why, note that the behavior of the transfer transaction is not influenced by the value of the counter. Indeed, the value of the counter is read in the `inc()` method, and neither that value nor the new value is returned to the calling method. Likewise, the execution flow of the `inc()` method does not depend on the value of the counter. That value is relevant only to determine the new value. So, if we delay the calculation of the new counter value, we can avoid the read of the counter value, thereby preventing the conflict of the second transaction with the already committed transaction. But then, when will the delayed calculation be done? To avoid the conflict, we propose to delay the read of the counter value until commit time, after the transaction has been validated. By this time, if the transaction is valid, we know that all the boxes read during the transaction were not changed meanwhile — thus, all the values read are still the most recent versions. Moreover, because commits of write transactions execute in mutual exclusion, we know that during the commit, no box can be changed. So, it is safe to update the transaction number to the most recent one and to access any box during this time, because the most recent version of every box is consistent with the currently committing transaction. The key insight here is that, at this time, the current value of the counter is the value written by the last committed transaction, rather than the value that the counter had when the current transaction started. So, we can calculate the new counter value and commit this new value. In order to do this we just have to know how many times the `inc()` method was called during the transaction (and the current value of the counter, of course).

Summarizing, to reimplement the `Counter` object in such a way that it is no longer a high-contention point (in the scenario presented above), we need to: (1) remember how many times the `inc()` method was called during a transaction, and (2) delay the calculation of the new counter value until commit time (if possible). We shall now see how to implement this.

3.3. Delaying computations with per-transaction boxes

To implement the strategy of delaying some computations until commit time, we propose a new abstraction in our model — the **per-transaction boxes**. The key idea behind per-transaction boxes is that they serve to hold values private to each transaction. These private values may then be merged with shared values, available on versioned boxes, when the transaction commits.

To accomplish this, each transaction keeps track of the value of each per-transaction box accessed during the transaction. When the top-level transaction commits,[3] the transaction calls the `commit` method on each of the per-transaction boxes, with the current per-transaction box value. Typically, the `commit` method will merge the per-transaction value received with the values of some versioned boxes. However, the responsibility of writing this method is left to the programmer that uses the per-transaction box.

The additional power given by this new abstraction stems from the fact that, when the per-transaction box `commit` method executes, the transaction has been validated and its number is set to the appropriate final number. Therefore, if the method reads some versioned box `B`, it will use the last committed value on `B`, rather than the value that `B` had when the transaction started. This last value of `B`, as we saw on the previous section, is necessarily consistent with the transaction.

Using this new abstraction, we can implement a new version of the `Counter` class that delays the increment, as shown in Fig. 5. In this new version, the `inc()` method no longer reads the value of `count`. Instead, it updates a per-transaction box which holds the number of times that the `inc()` method was called. When the transaction commits, the value of the per-transaction box is added to the `count` value. Obviously, the new `getCount()` method needs to take the `toAdd` value into account. The good thing about this new counter is that it causes no conflicts in the `transfer` transactions anymore. Moreover, all the changes were made in the counter class and the `transfer` method remains the same.

The `Counter` class is the simplest example where this approach of delaying the computations eliminates conflicts. More interesting cases include the addition and removal of elements to collections, for instance. In this case, we would

[3] The commit of a nested transaction simply propagates the per-transaction values to the transaction's parent.

```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  private PerTxBox<Long> toAdd =
    new PerTxBox<Long>(0L) {
      public void commit(Long value) {
        count.put(count.get() + value);
      }
    };

  public long getCount() {
    return count.get() + toAdd.get();
  }

  public @Atomic void inc() {
    toAdd.put(toAdd.get() + 1);
  }
}
```

Fig. 5. A `Counter` object that delays its increment.

use two per-transaction boxes: one to hold the elements to add, and another to hold the elements to remove. Both of these boxes would be changed by the addition and removal operations, and the commit of the boxes would have to perform the delayed additions and removals. Like in the `Counter` case, a collection using this approach can prevent the conflict between two concurrent transactions that change the same collection, because the access to the underlying structure of the collection is delayed until commit time. Naturally, the conflict might still exist if the concurrent transactions need to read the collection also. Likewise, if a transaction using a `Counter` object needs to read the value of the counter, then the delay of the `inc()` operation is of no use to prevent the conflict with a concurrent transaction that behaves the same. To read the counter's value, the `getCount()` method has to read the `count` box. We shall address these cases in Section 4.

To conclude the presentation of the per-transaction boxes, we note that the drawback of this approach is that the execution of the per-transaction boxes' `commit` methods increases the time of the commit operation, which executes in mutual exclusion with all the other commits of write transactions. However, neither the execution of other transactions nor the commit of read-only transactions is influenced in any way by the execution of the per-transaction boxes' `commit` methods.

3.4. Removing conflicts by restarting computations

The strategy used to reimplement the `Counter` class works whenever we can eliminate read operations. That happens when the values returned by those read operations are not needed to decide on the execution flow of the transaction. We shall now discuss some cases where this is not possible.

Consider an implementation of a singly-linked list that uses versioned boxes to keep the list forward links, and which contains the following methods: (1) the `remove(Object)` method that removes an object from the list, and (2) the `contains(Object)` method that returns a boolean value indicating whether the given object exists in the list.

Furthermore, suppose that we have a list `L`, with two objects `O1` and `O2`, that is used by two concurrent read-write transactions `T1` and `T2`. If `T1` removes `O1` from `L` and commits before `T2`, which calls `contains(O2)`, then, on a typical implementation of these methods, `T2` will have to abort because `L`'s structure changed. In particular, the head of the list now points to the node with `O2`, rather than the node with `O1`. However, after the changes made by `T1`, the result of `contains(O2)` is the same, because the list still contains `O2`. Therefore, if `T2` started after `T1` committed, it would produce the same results. So, `T2` can be serialized after `T1` and there should be no conflict.

Unfortunately, in this case we cannot delay the computation of `contains(O2)`, because the whole point of using this method inside a transaction is to know whether the list `L` contains `O2` or not — presumably, to decide on what

```
class List {
  ...
  @Restartable boolean contains(Object obj) {
    ...contains body...
  }
}
```

Fig. 6. Using restartable transactions.

to do next. We expect this to be true for every method that returns some value and does not have side-effects. Thus, per-transaction boxes do not help us here.

The rationale presented above for the conclusion that `T2` should not conflict with `T1`, was that the result of the operation that caused the conflict would be the same if that operation executed after `T1` committed. So, it seems that we can remove the conflict, if we re-execute the `contains(O2)` call at commit time, and the re-execution produces the same result. If the result is not the same, then the transaction execution might have been different, and therefore we must restart the transaction.

The advantage of this strategy is that it avoids the restart of the whole transaction, by re-executing just a smaller part of the transaction. However, unlike the restart of the whole transaction, the re-execution occurs at commit time, which means that it blocks the remaining commits during its execution.

Obviously, there is a tradeoff here that needs to be evaluated carefully. Nevertheless, we expect that for longer-running transactions, with small blocks of restartable operations, this strategy pays off. More so, if those blocks are high-contention points, because, in that case, they would force the transactions to execute almost sequentially anyway.

3.5. Restartable transactions

To deal with the problem identified in the previous section we propose a new abstraction: the **restartable transaction.**

From the programmer's point of view, this new abstraction can be introduced as a new annotation, `@Restartable`, to delimit this new kind of transactions. Fig. 6 illustrates its use in the example of the `List` class.

The semantics of this construction is an extension of the `@Atomic` annotation's semantics, introduced in Section 2.2. Like the previous one, it serves to demarcate a transaction. However, it also serves to give a hint to the transaction system that if a conflict occurs because of boxes read exclusively within this transaction, then the transaction can be re-executed at commit time to see whether it produces the same value as in the first execution, thereby resolving the conflict.

However, not all transactions can be restartable. If the transaction is not restartable, the `@Restartable` annotation behaves like the `@Atomic` annotation. Here, we restrict restartable transactions to nested read-only transactions implemented by a method. In this case, the only influence of the restartable transaction on the execution of its parent is the method's return value. In the following section, we will address the case of restartable transactions with side-effects.

To implement restartable transactions, we need to capture the following information:

- The method body with all its arguments, so that we can re-execute it later.
- The method return value, so that we can compare it with the result of the re-execution.
- The current state of the enclosing transaction, including all of its ancestors. This state consists of a copy of all the local-values maps of the restartable transaction's ancestors at the time that it started executing for the first time. This is needed, so that the re-execution can access the same values of the boxes written by the enclosing transaction.

To capture the method body with all its arguments (including the `this`), the `@Restartable` annotation can be expanded as shown in Fig. 7.

The `RestartableTransaction` object should obtain the set of boxes written in the current transaction, create a new nested transaction (having the current transaction as its parent) and execute the `run` method within the context of this newly created nested transaction. If the `run` method terminates normally and it did not write to any box,

```
class List {
  ...
  boolean contains(final Object obj) {
    return
      new RestartableTransaction<Boolean>() {
        public boolean run() {
          return restartable$contains(obj);
        }
      }.execute();
  }

  boolean restartable$contains(Object obj) {
    ...contains body...
  }
}
```

Fig. 7. Implementing restartable transactions.

then a **restartable transaction record** is created, with all the relevant data, and this record is added to the current transaction. Otherwise, the nested transaction is committed or aborted as usual. Naturally, the restartable transaction records should be propagated through the transaction hierarchy, as transactions commit. They will be used only at commit time by the top-level transaction.

Restartable transactions hide the read operations performed during their execution from their parents. Therefore, when the top-level transaction commits, the validity check will not take into account the boxes read during restartable transactions. However, after the top-level validity check passes, each of the restartable transaction records will be validated in turn.

A restartable transaction record is valid if none of the boxes read during the restartable transaction have changed. This is similar to the validity check of a top-level transaction. If a restartable transaction record is not valid, then it is re-executed in a newly created nested transaction. If the re-execution terminates normally, produces the same result as before, and does not write to any box, then the changes that occurred in the boxes do not affect the top-level transaction and it can continue. Otherwise, the commit of the top-level transaction should fail.

The key point here is, again, that the re-execution of each restartable transaction occurs during the commit of the top-level transaction, after it has been renumbered. This way, each re-execution will see the most recent values for the shared resources, much in the same way as the `commit` methods of the per-transaction boxes. Moreover, because the top-level transaction is valid, all the values read during the re-execution of the restartable transaction are consistent with the values read previously by the restartable transaction's ancestors.

Finally, the drawback of this approach to resolve conflicts is the same that we referred to at the end of Section 3.3: The re-execution of restartable transactions at commit time may introduce huge overheads in the execution time of the commit operation.

4. Towards a model for fine-grained restarts

The restartable transaction abstraction proposed in Section 3.5 allows fine-grained control over which parts of a transaction can be re-executed, provided that those parts do not change any boxes. In this section, we begin by discussing the implications of expanding this abstraction to include write operations. Then, we explore an extension to the model presented in Section 2 that is better suited to implement fine-grained restartable transactions.

4.1. Delaying computations versus restarting transactions

The first strategy we used to eliminate conflicts was to delay some computations until commit time. Then, we introduced the notion of restartable transaction, which, again, works by making some computation at commit time.

This apparent similarity raises the question of whether there is any relation between the two abstractions. More specifically, we would like to know whether restartable transactions can replace the use of per-transaction boxes.

We used the per-transaction boxes to implement a new counter because the counter's `inc` operation was causing many conflicts. Similar reasons led to the use of restartable transactions to implement the list's `contains` operation. So, it is tempting to use this later abstraction to implement the counter's `inc` operation, because it is simpler to use. Unfortunately, if we try this, we face the problem that the `inc` operation writes to a box. Therefore, to unify the two approaches, we need to extend the restartable transaction's semantics to accommodate write operations.

4.2. Restartable transactions that write

The generic necessary condition for having a restartable transaction is that its re-execution in a different context (at commit time) does not change the execution of the top-level transaction. A conservative approach to ensure this condition is to require that the restartable transaction returns the same result as before and does not perform any side-effects. This last restriction eliminates all the changes to shared resources. However, this approach is overly restrictive.

A straightforward change in the restartable transactions' semantics is to accept transactions that write new values to versioned boxes, provided that those boxes are never read in the remaining of the enclosing top-level transaction. This restriction applies both during the execution and the re-execution of the transaction, because, when a restartable transaction re-executes, different boxes may be written.

This change is sufficient to allow us to replace the per-transaction box by a restartable transaction in the simple case presented in Section 3.2. But if we consider a case where the counter is incremented twice in the same transaction, then the implementation based on a restartable transaction fails to eliminate the conflict, although the per-transaction box's solution still works. The restartable transaction approach fails in this case, because the value of the `count` box written by the first `inc` operation is used later (in the second `inc` operation). Therefore, a different value for that box can change the behavior of the top-level transaction. And it does, indeed, because the second increment would produce a different result. However, the second `inc` operation is a restartable transaction, also. So, if it re-executes after the re-execution of the first restartable transaction, then the correct value would be produced.

This later observation leads to another change in the semantics of restartable transactions: A restartable transaction may write a versioned box if, during the execution of the enclosing top-level transaction, that box is read only by restartable transactions, in which case, the later restartable transactions should be re-executed whenever the value of the box is changed. Although informally specified, this new semantics seems to be expressive enough to replace the use of the per-transaction box in the `Counter` class.

The problem with this extended semantics is that it is much harder to implement in our current model than the relatively simple per-transaction boxes. The difficulty stems from the fact that we need to keep track of which transaction produced a particular value for a box, together with the need to capture the currently written boxes whenever a restartable transaction starts.

4.3. Extending the versioned model

In our current model, if several nested transactions (of the same transaction) write to a box `B`, they do not produce different versions for `B`. Instead, they replace the previous value of `B` with the new value. This can be done, because nested transactions execute sequentially, so there is no need for older versions.

However, if we take into account restartable transactions, which may need older values when they re-execute, then it makes sense to use the same approach of keeping a different version for each committed transaction, independently of its nesting. Moreover, this approach extends naturally to allow concurrent nested transactions.

The fundamental change we propose here is to treat nested transactions in the same way as top-level transactions. The only difference between the transactions becomes the context where they execute. The execution of transactions inside other transactions forms a transaction tree, where each transaction in the tree keeps the information about what was read and what was written by itself, rather than merging this information with its parent.

The advantage, for restartable transactions, of this new approach, is that they become much more easy to implement. First, because all the execution contexts needed to re-execute the transaction are kept by the transaction tree. Second, because the re-execution of the transaction needs to concern only with updating its local values, rather than

with propagating the changes to its parent. Finally, because it becomes trivial to record the dependencies between different transactions.

With this approach it is feasible to consider that all transactions are restartable. Therefore, when a transaction conflicts with another, it is possible to determine the smallest portions of code that need to be restarted to remove the conflict. Naturally, any restart may fail to produce the same results, which would recursively cause a higher-level transaction to restart, until we eventually restart the top-level transaction.

One interesting aspect of this approach is that the transaction granularity greatly influences the ability to recover from a conflict. Furthermore, we believe that the use of fine-grained restartable transactions allows a graceful degradation from an optimistic concurrency control policy to a pessimistic policy, but only when needed.

5. Related work

Compared to other approaches to STM, e.g. [17,9,7,8], the distinctive feature of our model is the use of multiple versions for each memory location. Therefore, our approach is better suited for long-running read-only transactions than the rest. This comes at the expense of more memory overheads.

The use of multiple versions to increase the concurrency of transactional systems is well known in the area of database management systems. Since the seminal work of Bernstein and Goodman [1] on multi-version concurrency control, the technique of using multiple versions as the basis for optimistic concurrency control was applied in several contexts. One such application was made by Graham and Parker [4]. They presented a formalism for describing multi-version object base systems which is similar, at the model level, to our proposal. However, their work is in the context of object databases, which have different concerns compared to a programming language level software transactional memory. As far as we know, our STM proposal is the first to incorporate the ideas of multi-version concurrency control in this context.

The idea of keeping a history of values for each object whenever it is changed, rather than replacing the old value, was used by Reed in the context of the distributed execution of atomic actions [13,14]. However, many of his concerns regarding the difficulty of synchronization on a distributed system do not apply in the context of STM. Moreover, in Reed's approach, when an object is read, the history of the object may be updated by that read operation, thereby introducing a point of synchronization among concurrent read-only transactions, which defeats somehow the benefits of this approach.

The semantics of nesting in our model corresponds to what Moss and Hosking [12] describe as *linear nesting*. This simplified model allows a simpler implementation of nesting and is sufficient to support the `retry` and the `orElse` operations from [8]. Although we did not address these operations in this paper, their implementation in our model is quite straightforward.

A notable difference of our implementation is that it uses a single lock to ensure mutual exclusion during the commit, whereas others, e.g. [9], rely on a single CAS operation to perform the commit, because all the values are put in place during the transaction's execution. We may use a similar strategy in the future.

Regarding the reduction of transaction conflicts, Herlihy et al. [9] proposed the use of contention managers, which decide whether conflicting transactions should abort other transactions or wait for them to finish. Since then, several proposals of different contention management policies have been proposed [6,15,16]. By contrast with this work, our approach is to eliminate conflicts by providing abstractions that allow the implementation of concurrency-friendly objects. We believe that this approach is better suited to the development needs of a common programmer, as she can concentrate on specific objects, involved in specific transactions, instead of general contention management policies. Nevertheless, we believe that the two approaches complement each other, but more research is needed to determine how they should be combined.

Moreover, although we presented the discussion on the reduction of conflicts integrated with our model of versioned boxes, most of what we propose applies equally well to other STM approaches. For instance, the per-transaction boxes do not depend on the existence of versions and, therefore, could be used with any other STM, provided that there are some means to execute arbitrary code during the commit of a transaction.

Finally, the idea of re-executing part of a transaction during the commit is similar to the use of the *field calls* technique to reduce the lock duration in traditional database transaction processing [5].

6. Conclusions

In this paper we propose a new approach to implement software transactional memories, based on boxes that hold multiple versions of their contents. This allows the execution of read-only transactions that never conflict with other concurrent transactions. We implemented this approach in Java, as a library, and in the paper we discuss some of the design decisions of this implementation. The source code of our implementation can be found in [10].

The JVSTM is being used in a production environment by the FénixEDU project [3]. It replaced all the concurrency control mechanisms based on locks in this web application, with significant improvements on the perceived application performance. Versioned boxes were used to store each field of each one of the 240 domain classes, including all the collections used to store the 320 bidirectional associations between domain classes. The database has approximately 1 GB of data, and the application is deployed in a cluster with three servers with two processors each.

To improve the concurrency of high-contention transactions, we propose in this paper two new abstractions: the per-transaction box and the restartable transaction. These new abstractions facilitate the development of transactional objects that allow the reduction of conflicts. The per-transaction box mechanism is implemented in the JVSTM and is being heavily used in the FénixEDU project to reduce the conflicts on the collections used to implement the domain relationships.

Finally, we discuss an extension to the STM model that aims at providing better support for implementing restartable transactions.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments that helped us to improve this paper. Also, we would like to thank the Fénix team, for their support in putting our STM model to work in a production environment. Finally, we thank INESC-ID's Software Engineering Group, and especially to António Leitão, for the fruitful discussions during our group meetings.

References

[1] P.A. Bernstein, N. Goodman, Multiversion concurrency control — theory and algorithms, ACM Transactions on Database Systems 8 (4) (1983) 465–483.
[2] J.O. Coplien, Advanced C++ Programming Styles and Idioms, Addison-Wesley, Reading, MA, USA, 1992.
[3] FenixEDU, FénixEDU, 2005, Home page at: http://fenixedu.sourceforge.net.
[4] P. Graham, K. Barker, Effective optimistic concurrency control in multiversion object bases, in: E. Bertino, S. Urban (Eds.), Proceedings of the International Symposium on Object-Oriented Methodologies and Systems, vol. 858, Springer-Verlag, 1994, pp. 313–328.
[5] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, Los Altos, CA, USA, 1992.
[6] R. Guerraoui, M. Herlihy, S. Pochon, Toward a theory of transactional contention management, in: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing, ACM Press, Las Vegas, NV, USA, 2005.
[7] T. Harris, K. Fraser, Language support for lightweight transactions, in: Proceedings of the OOPSLA 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications, in: SIGPLAN Notices, vol. 36, ACM Press, Anaheim, CA, USA, 2003, pp. 388–402.
[8] T. Harris, S. Marlowe, S. Peyton-Jones, M. Herlihy, Composable memory transactions, in: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press, Chicago, IL, USA, 2005.
[9] M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer III, Software transactional memory for dynamic-sized data structures, in: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York, USA, 2003, pp. 92–101.
[10] JVSTM, JVSTM, 2005, Home page at: http://www.esw.inesc-id.pt/~jcachopo/jvstm.
[11] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, 2nd edition, Addison-Wesley, Reading, MA, USA, 1999.
[12] J.E.B. Moss, A.L. Hosking, Nested transactional memory: Model and preliminary architecture sketches, in: Workshop on Synchronization and Concurrency in Object-Oriented Languages, SCOOL05, San Diego, USA. Available at: http://hdl.handle.net/1802/2099, October 2005.
[13] D.P. Reed, Naming and synchronization in a decentralized computer system, Ph.D. Thesis, MIT, Cambridge, MA, USA, 1978.
[14] D.P. Reed, Implementing atomic actions on decentralized data, ACM Transactions on Computer Systems 1 (1) (1983) 3–23.
[15] W.N. Scherer III, M.L. Scott, Contention management in dynamic software transactional memory, in: Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs, St. John's, NL, Canada, July 2004.
[16] W.N. Scherer III, M.L. Scott, Advanced contention management for dynamic software transactional memory, in: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing, ACM Press, Las Vegas, NV, USA, 2005.
[17] N. Shavit, D. Touitou, Software transactional memory, in: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, ACM Press, Ottawa, Canada, 1995, pp. 204–213.