Single-Version STMs can be Multi-Version Permissive[⋆]

(Extended Abstract)

Hagit Attiya[1,2] and Eshcar Hillel[1]

[1] Department of Computer Science, Technion
[2] Ecole Polytechnique Federale de Lausanne (EPFL)

Abstract. We present PermiSTM, a *single-version* STM that satisfies a practical notion of *permissiveness*, usually associated with keeping many versions: it never aborts read-only transactions, and it aborts other transactions only due to a conflicting transaction (which writes to a common item), thereby avoiding spurious aborts. It avoids unnecessary contention on the memory, being *strictly disjoint-access parallel*.

1 Introduction

Transactional memory is a leading paradigm for programming concurrent applications for multicores. It is seriously considered as part of software solutions (abbreviated STMs) and as a basis for novel hardware designs, which exploit the parallelism offered by contemporary multicores and multiprocessors. A *transaction* encapsulates a sequence of operations on a set of *data items*: it is guaranteed that if a transaction commits, then all its operations appear to be executed atomically. A transaction may *abort*, in which case none of its operations are executed. The data items written by the transaction are its *write set*, the data items read by the transaction are its *read set*, and together they are the transaction's *data set*.

When an executing transaction may violate consistency, the STM can *forcibly* abort it. Many existing STMs, however, sometimes *spuriously* abort a transaction, even when in fact, the transaction may commit without compromising data consistency [9]. Frequent spurious aborts can waste system resources and significantly impair performance; in particular, this reduces the chances of long transactions, which often only read the data, to complete.

Avoiding spurious aborts has been an important goal for STM design, and several conditions have been proposed to evaluate how well it is achieved [8, 9, 12, 16, 20]. A *permissive* STM [9] never aborts a transaction unless necessary to ensure consistency. A stronger condition, called *strong progressiveness* [12], further ensures that even when there are conflicts, at least one of the transactions involved in the conflict is not aborted.

Alternatively, *multi-version (MV) permissiveness* [20] focuses on *read-only* transactions (whose write set is empty), and ensures they never abort; *update* transactions, with non-empty write set, may abort when in conflict with other transactions writing to the same items. As its name suggests, multi-version progressiveness was meant to

[⋆] This research is supported in part by the *Israel Science Foundation* (grant number 953/06).

be provided by a *multi-version* STM, maintaining multiple versions of each data item. It has been suggested [20] that refraining to abort read-only transactions mandates the overhead associated with maintaining multiple versions: additional storage, a complex implementation of a precedence graph (to track versions), as well as an intricate garbage collection mechanism, to remove old versions. Indeed, MV-permissiveness is satisfied by current multi-version STMs, both practical [20, 21] and more theoretical [16, 19], keeping many versions per item. It can be achieved by other multi-version STMs [3,22], if enough versions of the items are maintained.

This paper shows it is possible to achieve MV-permissiveness while keeping only a single version of each data item. We present *PermiSTM*, a single-version STM that is both MV-permissive and strongly progressive, indicating that multiple versions are not the only design choice when seeking to reduce spurious aborts. By maintaining a single version, PermiSTM avoids the high space complexity associated with multi-version STMs, which is often unacceptable in practice. This also eliminates the need for intricate mechanisms of maintaining and garbage collecting old versions.

PermiSTM is *lock-based*, like many contemporary STMs, e.g., [5–7, 23]. For each data item, it maintains a single version, as well as a lock, and a *read counter*, counting the number of pending transactions that have read the item. Read-only transactions never abort (without having to declare them as such, in advance); update transactions abort only if some data item in their read set is written by another transaction, i.e., at least one of the conflicting transactions commits. Although it is blocking, PermiSTM is deadlock-free, i.e., always some transaction can make progress.

The design choices of PermiSTM offer several benefits, most notably:

– Simple lock-based design makes it easier to argue about correctness.
– Read counters avoid the overhead of incremental validation, thereby improving performance, as demonstrated in [6, 17], especially in read-dominated workloads. Read-only transactions do not require validation at all, while update transactions validate their read sets only once.
– Read counters circumvent the need for a central mechanism, like a global version clock. Thus, PermiSTM is *strictly disjoint-access parallel* [10], namely, processes executing transactions with disjoint data sets do not access the same base objects.

It has been proved [20, Theorem 2] that a *weakly disjoint-access parallel* STM [2, 14] cannot be MV-permissive. PermiSTM, satisfying the even stronger property of *strict disjoint-access parallelism*, shows that this impossibility result depends on a strong progress condition: a transaction *delays only due to a pending operation* (by another transaction). In PermiSTM, a transaction may delay due to another transaction reading from its write set, even if no operation of the reading transaction is pending.

2 Preliminaries

We briefly describe the transactional memory model [15]. A *transaction* is a sequence of operations executed by a single process. Each operation either accesses a *data item* or tries to commit or abort the transaction. Specifically, a *read* operation specifies the item to read, and returns the value read by the operation; a *write* operation specifies

```
boolean CAS($obj$, $exp$, $new$) {
    // Atomically
    if $obj$ = $exp$ then
        $obj \leftarrow new$
        return TRUE
    return FALSE
}
```

```
boolean $k$CSS($o[k]$, $e[k]$, $new$) {
    // Atomically
    if $o[1]$ = $e[1]$ and ... $o[k]$ = $e[k]$ then
        $o[1] \leftarrow new$
        return TRUE
    return FALSE
}
```

Fig. 1. The CAS and *k-compare-single-swap* primitives.

the item and value to be written; a *try-commit* operation returns an indication whether the transaction committed or aborted; an *abort* operation returns an indication that the transaction is aborted. While trying to commit, a transaction might be aborted, e.g., due to conflict with another transaction.[3] A transaction is *forcibly aborted* if an invocation of a try-commit returns an indication that the transaction is aborted.

Every transaction begins with a sequence of read and write operations. The last operation of a transaction is either an access operation, in which case the transaction is pending, or a try-commit or an abort operation, in which case the transaction is *committed* or *aborted*.

A *software implementation* of *transactional memory* (*STM*) provides data representation for transactions and data items using *base objects*, and algorithms, specified as *primitives* on the base objects, which *asynchronous* processes follow in order to execute the operations of the transactions.

An *event* is a computation *step* by a process consisting of local computation and the application of a primitive to base objects, followed by a change to the process's state, according to the results of the primitive. We employ the following primitives: READ(o) returns the value in base object o; WRITE(o, v) sets the value of base object o to v; CAS(o, exp, new) writes the value *new* to base object o if its value is equal to *exp*, and returns a success or failure indication; kCSS is similar to CAS, but compares the values of k independent base objects (see Figure 1).

2.1 STM Properties

We require the STM to be *opaque* [11]. Very roughly stated, opacity is similar to requiring *strict view serializability* applied to all transactions (included aborted ones).

Restrictions on spurious aborts are stated by the following two conditions.

Definition 1. *A* multi-version (MV-)permissive *STM [20] forcibly aborts a transaction only if it is an update transaction that has a conflict with another update transaction.*

Definition 2. *An STM is* strongly progressive *[12] if a transaction that has no conflicts cannot be forcibly aborted, and if a set of transactions have conflicts on a single item then not all of them are forcibly aborted.*

[3] Two transactions *conflict* if they access the same data item; the conflict is *nontrivial* if at least one of the operations is a write. In the rest of the paper all conflicts are nontrivial conflicts.

These two properties are incomparable: strong progressiveness allows a read-only transaction to abort, if it has a conflict with another update transaction; on the other hand, MV-permissiveness does not guarantee that at least one transaction is not forcibly aborted in case of a conflict.

Finally, an STM is *strictly disjoint-access parallel* [10] if two processes, executing transactions T_1 and T_2, access the same base object, at least one with a non-trivial primitive, only if the data sets of T_1 and T_2 intersect.

3 The Design of PermiSTM

The design of PermiSTM is very simple. The first and foremost goal is to ensure that a read-only transaction never aborts, while maintaining only a single-version. This suggests that the data returned by a read operation issued by a read-only transaction T should not be overwritten until T completes. A natural way to achieve this goal is to associate a read counter with each item, tracking the number of pending transactions reading from the item. Transactions that write to the data items respect the read counters; an update transaction commits and updates the items in its write set only in a "quiescent" configuration, where no (other) pending transaction is reading an item in its write set. This yields read-only transactions that guarantee consistency without requiring validation and without specifying them as such in advance.

The second goal is to guarantee consistent updates of data items, by using ordinary locks to ensure that only one transaction is modifying a data item at each point. Thus, before writing its changes, at commit time, an update transaction acquires locks.

Having two different mechanisms—locks and counters—in our design, requires care in combining them. One question is when during the executing, a transaction decrements the read counters of the items in its read set? The following simple example demonstrates how a deadlock may happen if an update transaction does not decrement its counters, before acquiring locks:

T_1:	read(a)		write(b)		try-commit
T_2:		read(b)		write(a)	try-commit

T_1 and T_2 incremented the read counters of a and b, respectively, and later, in commit time, T_1 acquires a lock on b, while T_2 acquires a lock on a. To commit, T_1 has to wait for T_2 to complete and decrement the read counter of b, while T_2 has to wait for the same to happen with T_1 and item a. Since an update transaction first decrements read counters, it must ensure consistency by acquiring locks also for items in its read set. Therefore, an update transaction acquires locks for *all* items in its data set.

Finally, read counters are incremented as they are encountered during the execution of the transaction. What happens if read-only transactions wait for locks to be released? The next example demonstrates how this can create a deadlock:

T_1:	read(a)		read(b)
T_2:	write(b)	write(a)	try-commit

If T_2 acquires a lock on b, then T_1 cannot read b until T_2 completes; T_2 cannot commit as it has to wait for T_1 to complete and decrease the read counter of a; MV-permissiveness

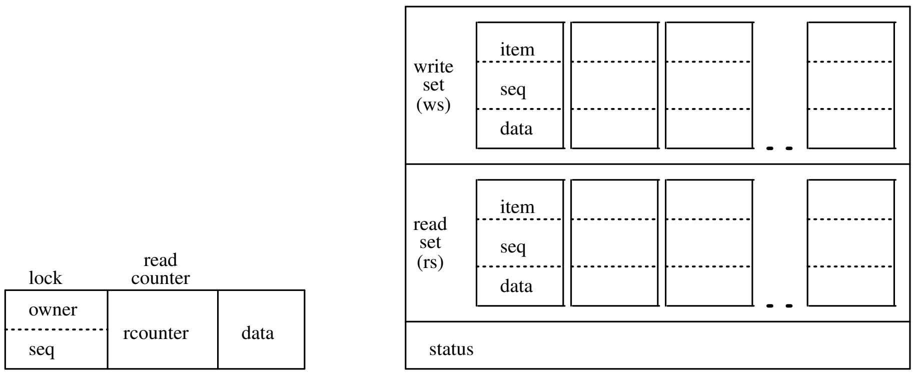

Fig. 2. Data structures used in the algorithm: an item (left) and a transaction descriptor (right).

does not allow both transactions to be forcibly aborted. Thus, read counters get preference over locks, and they can always be incremented. Prior to committing, an update transaction first decrements its read counters, and then acquires locks on all items in its data set, in a fixed order (while validating the consistency of its read set); this avoids deadlocks due to blocking cycles, and livelocks due to repeated aborts.

Since committing a transaction and committing its updates are not done atomically, a committed transaction that has not yet completed updating all the items in its write set, can yield an inconsistent view for a transaction reading one of these items. If a read operation simply reads the value in the item, it might miss the up-to-date value of the item. Therefore, a read operation is required to read the *current* value of the item, which can be found either in the item, or in the data of the transaction.[4]

To simplify the exposition of PermiSTM, k-compare-single-swap (k`CSS`) [18] is applied to commit an update transaction while ensuring that the read counters of the items in its write set are all zero. Section 4 describes how the implementation can be modified to use only `CAS`; the resulting implementation is (strongly) disjoint-access parallel but is not strictly disjoint-access parallel.

Data Structures. Figure 2 presents the data structures of items and transactions' descriptors used in our algorithm. We associate a lock and a read counter with each item, as follows:

- A *lock* includes an *owner* field, and an unbounded sequence number, *seq*, that are accessed atomically. The *owner* field is set to the id of the update transaction owning the lock and is 0 if no transaction holds the lock. The *seq* field holds the sequence number of the *data*, it is incremented whenever a new data is committed to the item, and it is used to assert the consistency of reads.
- A simple *read counter*, *rcounter*, tracks how many transactions are reading the item.
- The *data* field holds the value that was last written to the item, or its initial value if no transaction yet written to the item.

[4] This is analogous to the notion of current version of a transactional object in DSTM [13].

The descriptor of a transaction consists of the read set, *rs*, the write set *ws*, and the *status* of the transaction. The read and write sets are collections of *data items*.

- A data item in the *read set* includes a reference to an *item*, the *data* read from the item, and the sequence number of this data, *seq*.
- A data item in the *write set* includes a reference to an *item*, the *data* to be written in the item, and the sequence number of the new data, *seq*, i.e., the sequence number of the current data plus 1.
- A *status* indicates if the transaction is COMMITTED or ABORTED, initially NULL.

The *current data* and *sequence number* of an item are defined as follows: If the lock of the item is owned by a committed transaction that writes to this item, then the current data and sequence number of the item appear in the write set of the owner transaction. Otherwise (*owner* is 0, or the owner is not committed, or the item is not in the owner's write set), the current data and current sequence number appear in the item.

The Algorithm. Next we give a detailed description of the main methods, for handling the operations; the code appears in Pseudocodes 1 and 2. The reserved word *self* in the pseudocode is a self-reference to the descriptor of the transaction whose code is being executed.

read method: If the item is already in the transaction's read set (line 2), return the value from the read set (line 3). Otherwise, increment the read counter of the item (line 5). Then, the reading transaction adds the item to its read set (line 7) with the current data and sequence number of the item (line 6).

write method: If the item is not already in the transaction's write set (line 11), then add the item to the write set (line 12). Set *data* of the item in the transaction's write set to the new data to be written (line 13). *No lock is acquired at this stage.*

tryCommit method: Decrement all the read counters of the items in the transaction's read set (line 16). If the transaction is *read-only*, i.e., the write set of the transaction is empty (line 17), then commit (line 18); the transaction completes and returns (line 19).

Otherwise, this is an *update* transaction and it continues: acquire locks on all items in the data set (line 20); commit the transaction (line 22) and the changes to the items (lines 23-25); release locks on all items in the data set (line 26). The transaction may abort while acquiring locks due to a conflict with another update transaction (line 21).

acquireLocks method: Acquire locks on all items in the data set of the transaction by their order (line 30). If the item is in the read set (line 33), check that the sequence number in the read set (line 34) is the same as the current sequence number of the item (line 32). If the sequence number has changed (line 35) then the data read is overwritten by another committed transaction and the transaction aborts (line 36).

Use CAS to acquire the lock: set *owner* from 0 to the descriptor of the transaction; if the item is in the read set this is done while asserting that *seq* is unchanged (line 38). If the CAS failed then *owner* is non-zero since there is another owner (or *seq* has changed), so spin, re-reading the lock (line 38) until *owner* is 0.

If the item is in the write set (line 39), set the sequence number of the item in the transaction's write set, *seq*, to the sequence number of the current data plus 1 (line 41).

commitTx method: Use kCSS to set *status* to COMMITTED, while ensuring that all read counters of items in the transaction's write set are 0 (line 47). If the read counter

Pseudocode 1 Methods for read, write and try-commit operations

```
 1: Data read(Item item) {
 2:     if item in rs then
 3:         di ← rs.get(item)
 4:     else
 5:         incrementReadCounter(item)
 6:         di ← getAbsVal(item)
 7:         rs.add(item,di)
 8:     return di.data
 9: }
10: write(Item item, Data data) {
11:     if item not in ws then
12:         ws.add(item,⟨item,0,0⟩)
13:     ws.set(item,⟨item,0,data⟩)
14: }
15: tryCommit() {
16:     decrementReadCounters() // decrement read counters
17:     if ws is empty then // read-only transaction
18:         WRITE(status, COMMITTED)
19:         return
          // update transaction
20:     acquireLocks() // lock all the data set
21:     if ABORTED = READ(status) then return
22:     commitTx() // commit update transaction
23:     for each item in ws do // commit the changes to the items
24:         di ← owner.ws.get(item)
25:         WRITE(item.data, di.data)
26:     releaseLocks() // release locks on all the data set
27: }
28: acquireLocks() {
29:     ds ← ws.add(rs) // items in the data set (read and write sets)
30:     for each item in ds by their order do
31:         do
32:             cur ← getAbsVal(item) // current value
33:             if item in rs then // check validity of read set
34:                 di ← rs.get(item) // value read by the transaction
35:                 if di.seq != cur.seq then // the data is overwritten
36:                     abort()
37:                     return
38:         while !CAS(item.lock, ⟨0,cur.seq⟩, ⟨self,cur.seq⟩)
39:         if item in ws then
40:             di ← ws.get(item)
41:             ws.set(item,⟨item,cur.seq+1,di.data⟩)
42: }
43: commitTx() {
44:     kCompare[0] ← status // the location to be compared and swaped
45:     for $i = 1$ to $k - 1$ do // $k - 1$ locations to be compared
46:         kCompare[i] ← ws.get($i$).item.rcounter
47:     while !$k$CSS(kCompare, ⟨NULL,0 . . . 0⟩, COMMITTED) do
48:         no-op // until no reading transactions is pending
49: }
```

Pseudocode 2 Additional methods for PermiSTM

```
incrementReadCounter(Item item) {
    do $m \leftarrow$ READ(item.*rcounter*)
    while !CAS(item.*rcounter*, $m$, $m+1$)
}
decrementReadCounters() {
    for each item in *rs* do
        do $m \leftarrow$ READ(item.*rcounter*)
        while !CAS(item.*rcounter*, $m$, $m-1$)
}
releaseLocks() {
    ds $\leftarrow$ *ws*.add(*rs*)
    for each item in ds do
        di $\leftarrow$ ds.get(item)
        WRITE(item.*lock*, $\langle$0,di.*seq*$\rangle$)
}
DataItem getAbsVal(Item item) {
    lck $\leftarrow$ READ(item.*lock*)
    dt $\leftarrow$ READ(item.*data*)
    di $\leftarrow$ $\langle$item, lck.*seq*, dt$\rangle$ // values from the item
    if lck.*owner* != 0 then
        sts $\leftarrow$ READ(lck.*owner.status*)
        if sts = COMMITTED then
            if item in lck.*owner.ws* then
                di $\leftarrow$ lck.*owner.ws*.get(item) // values from the write set of the owner
    return di
}
abort() {
    ds $\leftarrow$ *ws*.add(*rs*)
    for each item in *ds* do
        lck $\leftarrow$ READ(item.*lock*)
        if lck.*owner* = *self* then // the transaction owns the item
            WRITE(item.*lock*, $\langle$0,lck.*seq*$\rangle$) // release lock
    WRITE(*status*, ABORTED)
}
```

of one of these items is not 0, a pending transaction is reading from this item, then spin, until all *rcounter*s are 0.

Properties of PermiSTM. Since PermiSTM is lock-based, it is easier to argue that it preserves opacity than in implementations that do not use locks. Specifically, an update transaction holds locks on all items in its data set before committing, which allows updating transactions to be serialized at their commit point. The algorithm ensures that an update transaction does not commit, leaving the value of the items in its write set unchanged, as long as there is a pending transaction reading one of the items to be written. A read operation reads the current value of the item, after incrementing its read counter. So, if an update transaction commits before the read counter is incremented, but the changes are not yet committed in the items, the reading transaction still maintains a consistent state as it reads the value from the write set of the committed transaction, which is the up-to-date value of the item. Hence, a read-only transaction is serialized after the update transaction that writes to one of the read items and is last to be committed. Since all transactions do not decrement read counters until commit time, and since all read operations return the up-to-date value of the item, all transactions maintain a consistent view. As this holds for committed as well as aborted transactions, PermiSTM is opaque.

Next we discuss the progress properties of the algorithm. After an update transaction acquires locks on all items in its data set it may wait for other transactions reading items in its write set to complete, it may even starve due to continual stream of readers; thus, our STM is blocking. However, the STM guarantees strong progressiveness, as transactions are forcibly aborted only due to another committed transaction with a read-after-write conflict; since read-only transactions are never forcibly aborted, PermiSTM is MV-permissive. Furthermore, read-only transactions are *obstruction-free* [13]. A read-only transaction may delay due to contention with concurrent transactions, updating the same read counters, but once it is running solo it is guaranteed to commit.

Write, try-commit and abort operations only access the descriptor of the transaction and the items in the data set of the transaction; this may result in contention only with non-disjoint transactions. A read operation, in addition to accessing the read counter of the item, also reads the descriptor of the owning transaction, which may result in contention only with non-disjoint transactions; thus, PermiSTM is strictly disjoint-access parallel. Note that disjoint transactions may concurrently read the descriptor of a transaction owning items the transactions read, however, this does not violate strict disjoint-access parallelism. Furthermore these disjoint transaction read from the same base object only if they all intersect with the owning transaction; This property is called *2-local contention* [1] and it implies (strong) disjoint-access parallelism [2].

4 CAS-Based PermiSTM

The kCSS operation can be implemented in software from CAS [18], without sacrificing the properties of PermiSTM. However, this implementation is intricate and incurs a step complexity that can be avoided in our case. This section outlines the modifications of PermiSTM needed to obtain an STM with similar properties using CAS instead of a kCSS primitive; this results in more costly read operations.

We still wish to guarantee that an update transaction commits only in a "quiescent" configuration, in which no other transaction is reading an item in its write set. If the committing update transaction does not use kCSS, then the responsibility of "notifying" the update transaction that it cannot commit is shifted to the read operations, and they pay the extra cost of preventing the update transactions from committing in a non-quiescent configuration.

A transaction commits by changing its status from NULL to COMMITTED; a way to prevent an update transaction from committing is by invalidating its status. For this purpose, we attach a sequence number to the transaction status. Prior to committing, an update transaction reads its status, which now includes the sequence number, and repeats the following for each item in its write set: spin on the item's read counter until the read counter becomes zero, then annotate the zero with a reference to its descriptor, and the status sequence number. The transaction changes its status to COMMITTED only if the sequence number of its status has not changed since it has read it. Once it completes annotating all zero counters, and unless it is notified by some read operation that one of the counters changed and it is no longer "quiescent", the update transaction can commit—using only a CAS.

A read operation basically increases the read counter, and then reads the current value of the item. The only change is when it encounters a "marked" counter. If the update transaction annotating the item already committed, the read operation simply increases the counter. Otherwise, the read operation invalidates the status of the update transaction, by increasing its status sequence number. If more than one transaction is reading an item from the write set of the update transaction, at least one of them prevents the update transaction from committing, by changing its status sequence number.

The changes in the data structures used by the algorithm are as follows: The status of a transaction descriptor now includes the *state* of the transaction (NULL, COMMITTED, or ABORTED), as well as a sequence number, *seq*, that is used to invalidate the status; these fields are accessed atomically. The read counter, *rcount*, of an item is a tuple including a *counter* of the number of readers, the *owner* transaction of the item (holding its lock), and *seq* matching the status sequence number of the owner.

We reuse the core implementation of operations from Pseudocodes 1 and 2. The most crucial modification is in the protocol for incrementing the read counter, which invalidates the status of the owner transaction when increasing the item's read counter.

Pseudocode 3 presents the main modifications. In order to commit, an update transaction reads the read counter of every item in its write set (lines 87-88), and when the read counter is 0, the update transactions annotates the 0 with its descriptor and status sequence number, using CAS (line 89). Finally it sets the status to COMMITTED while increasing the status sequence number, using CAS. If the status was invalidated and the last CAS fails, the transaction re-reads the status (line 86) and goes over the procedure again. A successful CAS implies that the transaction committed while no other transaction is reading any item in its write set.

To ensure that an update transaction only commits in a "quiescent" configuration, a read operation that finds the read counter of the item "marked" (lines 94-95) continuous as follows: use CAS to invalidate the status of the owner transaction—by increasing its sequence number (line 96), if the status sequence number has changed, either the owner is committed or its status was already invalidated; finally, the reader transaction simply increases the reader counter using CAS (line 97). If increasing the reader counter fails, the reader repeats the procedure.

While decreasing the read counters the reader transaction cleans each read counter by setting its *owner* and *seq* fields to 0 (line 102).

In addition, methods such as tryCommit and abort are adjusted to handle the new structure, for example, accessing the read counter and state indicator through the new read counters and status fields.

The resulting algorithm is not strictly disjoint-access parallel. Two transactions, T_1 and T_2, reading items a and b, respectively, may access the same base object, when checking and invalidating the status of a third transaction, T_3, updating these items. The algorithm, however, has 2-local contention [1] and is (strongly) disjoint-access parallel, as this memory contention is always due to T_3, which intersects both T_1 and T_2.

Pseudocode 3 methods for avoiding kCSS

```
commitTx() {
    do
        sts ← READ(*status*)
        **for each** item in *ws* **do**
            **do** rc ← READ(item.*rlock*) // spin until no readers
            **while** !CAS(item.*rcounter*, ⟨0,rc.*owner*,rc.*seq*⟩, ⟨0,*self*,sts.*seq*⟩) // annotated 0
        // commit in a “quiescent” configuration
    **while** !CAS(*status*, ⟨NULL,sts.*seq*⟩, ⟨COMMITTED,sts.$seq+1$⟩)
}
incrementReadCounter(Item item) {
    **do**
        rc ← READ(item.*rcounter*)
        **if** rc.*owner* != 0 **then** // the read counter is “marked”
            CAS(rc.*owner.status*, ⟨NULL,rc.*seq*⟩, ⟨NULL,rc.$seq+1$⟩) // invalidate status
    **while** !CAS(item.*rcounter*, rc, ⟨rc.$counter+1$,rc.*owner*,rc.*seq*⟩) // increase counter
}
decrementReadCounters() {
    **for each** item in *rs* **do**
        **do** *rc* ← READ(item.*rcounter*)
        **while** !CAS(item.*rcounter*, rc, ⟨rc.$counter-1$,0,0⟩) // clean and decrease counter
}
```

5 Discussion

This paper presents PermiSTM, a single-version STM that is both MV-permissive and strongly progressive; it is also disjoint-access parallel. PermiSTM has simple design, based on read counters and locks, to provide consistency without incremental validation. This also simplifies the correctness argument.

The first variant of PermiSTM uses a *k-compare-single-swap* to commit update transactions. No architecture currently provides kCSS in hardware, but it can be supported by *best-effort hardware transactional memory* (cf. [4]).

In PermiSTM, update transactions are not *obstruction free* [13], since they may block due to other conflicting transactions. Indeed, a single-version, obstruction-free STM cannot be *strictly* disjoint-access parallel [10]. Read-only transactions modify the read counters of all items in their read set. This matches the lower bound for read-only transactions that never abort, for (strongly) disjoint-access parallel STMs [2].

Several design principles of PermiSTM are inspired by TLRW [6], which uses *read-write* locks. TLRW, however, is not permissive as read-only transactions may abort due to a timeout while attempting to acquire a lock. We avoid this problem by tracking readers through read counters (somewhat similar to SkySTM [17]) instead of read locks.

Our algorithm improves on the multi-versioned UP-MV STM [20], which is not weakly disjoint-access parallel (nor strictly disjoint-access parallel), as it uses a global transaction set, holding the descriptors of all completed transactions yet to be collected by the garbage collection mechanism. UP-MV STM requires that operations execute atomically; its progress properties depend on the precise manner this atomicity is guaranteed, which is not detailed. We remark that simply enforcing atomicity with a global lock or a mechanism similar to TL2 locking [5] could make the algorithm blocking.

Acknowledgements. We thank the anonymous refereees for helpful comments.

References

1. Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *PODC 97*, 111–120.
2. H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *SPAA '09*, 69–78.
3. U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT '08*.
4. D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *SPAA '10*, 325–334.
5. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06*, 194–208.
6. D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *SPAA '10*, 284–293.
7. R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, 2006.
8. V. Gramoli, D. Harmanci, and P. Felber. Towards a theory of input acceptance for transactional memories. In *OPODIS '08*, 527–533.
9. R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC '08*, 305–319.
10. R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *SPAA '08*, 304–313.
11. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08*, 175–184.
12. R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *POPL '09*, 404–415.
13. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03*, 92–101.
14. A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC '94*, 151–160.
15. M. Kapalka. *Theory of Transactional Memory*. PhD thesis, EPFL, 2010.
16. I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. In *SPAA '09*, 59–68.
17. Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09*.
18. V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *SPAA '03*, 314–323.
19. J. Napper and L. Alvisi. Lock-free serializable transactions. Technical Report TR-05-04, The University of Texas at Austin, 2005.
20. D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC '10*, 16–25.
21. D. Perelman and I. Keidar. SMV: Selective Multi-Versioning STM. In *TRANSACT '10*.
22. T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06*, 284–298.
23. B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06*, 187–197.