

A Single-Version STM that is Multi-Version Permissive*

Hagit Attiya
Technion

hagit@cs.technion.ac.il

Eshcar Hillel
Technion & Yahoo! Labs

eshcar@yahoo-inc.com

April 21, 2011

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. PermiSTM also avoids unnecessary contention on the memory, being *strictly disjoint-access parallel*.

We first present a variant of PermiSTM that uses *k-compare-single-swap* primitive. Then we show a variant with similar properties using only CAS, and we also show how the livelocks it may incur can be avoided with *best-effort hardware transactions*.

1 Introduction

Transactional memory is a leading paradigm for programming concurrent applications for multicores. It is 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 their union is 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 [12]. 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.

*This research is supported in part by the *Israel Science Foundation* (grants 1344/06 and 1227/10), and by funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 238639, ITN project TRANSFORM.

Avoiding spurious aborts has been an important goal for STM design, and several conditions have been proposed to evaluate how well it is achieved [10, 12, 15, 19, 24]. A *permissive* STM [12] never aborts a transaction unless necessary to ensure consistency. A stronger condition, called *strong progressiveness* [15], 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* [24] 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 be provided by a *multi-version* STM, maintaining multiple versions of each data item. It has been suggested [24] that refraining to abort read-only transactions mandates the overhead associated with maintaining multiple versions: additional storage, a complex data structure to represent the 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 [24, 25] and more theoretical [19, 22], keeping many versions per item. It can be achieved by other multi-version STMs [4, 26], if sufficiently many 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., [7–9, 27]. For each data item, it maintains a single version, a lock, as well as 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 [8, 20], 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* [13], namely, processes executing transactions with disjoint data sets do not access the same base objects.

It has been proved [24, Theorem 2] that a *weakly disjoint-access parallel* STM [3, 17] cannot be MV-permissive. PermiSTM, satisfying the even stronger property of *strict disjoint-access parallelism*, shows that this impossibility result critically depends on the assumption that a transaction *delays only due to a pending operation* (by another transaction). In PermiSTM, a transaction may delay due to

another pending transaction reading from its write set, even if no operation of the reading transaction is pending.

The next section presents the model that is used to describe transactions and the properties of STMs. To simplify the exposition of PermiSTM, Section 3 presents a variant in which transactions that are not read-only use a *k-compare-single-swap* (*k*CSS) primitive at commit-time; the properties of the algorithm are discussed in Section 4. Section 5 outlines the modifications needed to obtain an STM with similar properties using only CAS; this results in more costly read operations. In Section 6, we show how hardware transactional memory can be employed to avoid livelocks that may occur in the latter variant. Finally, we conclude and discuss related work in Section 7.

2 Preliminaries

We briefly describe the transactional memory approach. We concentrate on describing the features that are required to present and prove our algorithm; more details on this model can be found in [18].

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. In more detail, a *read* operation specifies the item to read, and returns the value read by the operation; a *write* operation specifies 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.¹ In this case, we say that the transaction is *forcibly aborted*.²

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*; compare&swap(*o, exp, new*) (CAS) writes the value *new* to base object *o* if its value is equal to *exp*, and returns a success or failure indication (see Figure 1).

A *configuration* is a complete description of the system at some point in time, i.e., the state of each process and the state of each shared base object.

An *execution interval* α is a finite or infinite alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \dots$, where C_k is a configuration, ϕ_k is an event and the application of ϕ_k to C_k results in C_{k+1} , for every $k = 0, 1, \dots$

¹Two 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.

²In the full model, read and write operations may also cause a transaction to abort; however, this does not happen in our algorithms.

```

boolean CAS(obj, exp, new) {
  // Atomically
  if obj = exp then
    obj ← new
    return TRUE
  return FALSE
}

```

Figure 1: The CAS primitive.

The *interval of a transaction* T is the execution interval that starts at the first event of T and ends at the last event of T , if there is one, taken by the process executing the algorithm for T . If T does not have a last event in the execution, then the interval of T is the (possibly infinite) execution interval starting at the first event of T . Two transactions *overlap* if their intervals overlap; they are non-overlapping, otherwise.

2.1 Opacity

An STM is *serializable* [23] if transactions appear to execute sequentially, one after the other, in some *serialization order*. An STM is *strictly serializable* [23] if the serialization order preserves the order of non-overlapping transactions.

Opacity [14] requires all committed transactions to appear to execute sequentially in an order that agrees with the order of non-overlapping transactions, as in strict serializability. It also requires all transactions, including aborted ones, to read values that are consistent with a serialization of the committed transactions.

2.2 Reducing Spurious Aborts

The following two conditions restrict the situations in which a transaction is forcibly aborted.

Definition 1 *An STM is multi-version (MV-)permissive [24] if a transaction is forcibly aborted only if it is an update transaction that has a conflict with another update transaction.*

Definition 2 *An STM is strongly progressive [15] 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.*

These two properties are incomparable: unlike MV-permissiveness, 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, a property ensured by strong progressiveness.

2.3 Disjoint Access Parallelism (DAP)

Disjoint-access parallelism [17] captures the intuition that transactions accessing disjoint parts of the data should not interfere with each other. A *conflict graph* captures the distance between overlapping operations by representing conflicts between transactions that overlap in time. The vertices in the conflict graph represent transactions, and edges connect two transactions that access the same item.

Two transactions T_1 and T_2 are *disjoint access* if there is no path between them in the conflict graph of the minimal execution interval containing the intervals of T_1 and T_2 . Two transactions T_1 and T_2 are *strictly disjoint access* if there is no edge between them in the conflict graph of the minimal execution interval containing the intervals of T_1 and T_2 .

An STM is (*strongly*) *disjoint-access parallel* if two processes p_1 and p_2 , executing transactions T_1 and T_2 , concurrently access the same base object, only if T_1 and T_2 are not disjoint access.

An STM is *strictly disjoint-access parallel* [13] 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 T_1 and T_2 are not strictly disjoint access. That is, the data sets of T_1 and T_2 are disjoint. This notion is much stronger than ordinary disjoint-access parallelism, which allows two transactions with disjoint data sets to access the same base objects, provided they are connected via other transactions.

At the middle-ground, an STM has *2-local contention* [1] if two processes, executing transactions T_1 and T_2 , access the same base object, only if their distance in the conflict graph is 2. That is, either the data sets of T_1 and T_2 intersect, or their data sets intersect the data set of a third transaction. Note that this property implies disjoint-access parallelism [3].

3 The Design of PermiSTM

The design of PermiSTM is 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 read 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 return consistent values without requiring validation and without specifying them as such in advance.

To guarantee consistent updates of data items, we need ordinary locks to ensure that only one transaction is modifying a data item at each point. Thus, before writing its changes, an update transaction acquires locks. To avoid deadlocks, the transaction acquires the locks at commit time, when all the items are known, in a predetermined order.

Having two different mechanisms to ensure consistency—locks and counters—in our design, requires care in putting them together. 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, at 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 .

This necessitates that counters be decremented before acquiring locks, and raises the issue of ensuring that the values read are still consistent while the transaction writes new values. Since an update transaction first decrements read counters, it must ensure consistency by acquiring locks also for items in its read set. Therefore, before 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.

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 does not allow both transactions to be forcibly aborted. Thus, read counters get preference over locks, and they can always be incremented.

Since committing a transaction and writing its updates to the write set 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. For example, consider the transaction depicted next: T_1 writing to items a and b , and T_2 reading these items after T_1 committed. If T_1 writes the new values to the items after T_2 reads the value from item a , but before T_2 reads the value from item b , the view of T_2 has the value of a before T_1 , and the value of b after T_1 completed, which are not consistent.

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

Therefore, a read operation has to read the *current* value of the item, which can be found either in the item, or in the data of the transaction.³ In the example, this means that when T_2 reads an item, and it discovers that T_1 , the owner of the item, has committed, T_2 reads the new value of the item from the write set of T_1 ; this ensures T_2 has a consistent view even if the values are not written yet in the items.

To simplify the exposition of PermiSTM, an update transaction is committed while ensuring that the read counters of the items in its write set are all zero, using a *k-compare-single-swap* primitive (*k*CSS). The *k*CSS primitive [21] is similar to CAS, but compares the values of *k* independent base objects (see Figure 2). Later, we describe how the implementation is modified to use only CAS.

³This is analogous to the notion of current version of a transactional object in DSTM [16].

```

boolean kCSS(o[k], e[k], new) {
  // Atomically
  if o[1] = e[1] and ... o[k] = e[k] then
    o[1] ← new
    return TRUE
  return FALSE
}

```

Figure 2: The *k*-compare-single-swap primitive.

3.1 Data Structures

Figure 3 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 value is committed to the item, and 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.

The descriptor of a transaction consists of the *read set*, the *write set*, 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*, *data* is the value read from the item, and *seq* is the sequence number of this value.
- A data item in the *write set* includes a reference to an *item*, *data* is the value to be written in the item, and *seq* is the sequence number of the new value, i.e., the current sequence number 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.

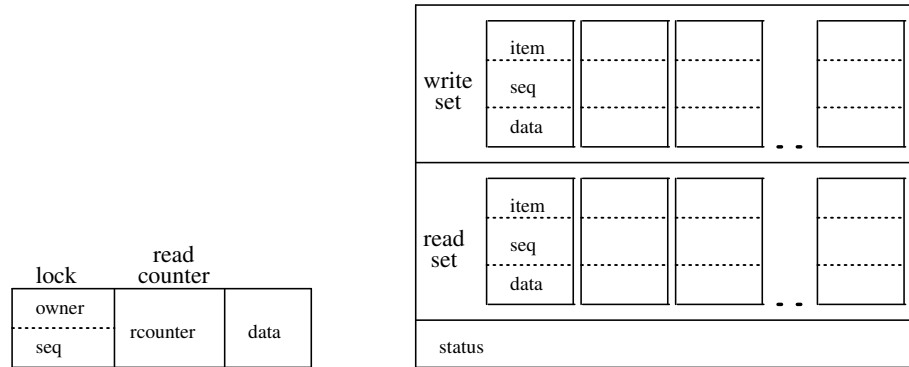


Figure 3: Data structures used in the algorithm: an item (left) and a transaction descriptor (right).

3.2 The Algorithm

Next we give a detailed description of the main methods for handling the operations, appearing in Pseudocode 1; additional methods appear in Pseudocode 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), the transaction returns the value from the read set (line 3). Otherwise, it increments 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), the transaction adds the item to its write set (line 12), and sets *data* of the item in the write set to the new value to be written (line 13). *No lock is acquired at this stage.*

tryCommit method: The transaction decrements all the read counters of the items in its read set (line 16). If the transaction is *read-only*, i.e., the write set of the transaction is empty (line 17), then the transaction commits (line 18) and returns (line 19).

Otherwise, this is an *update* transaction and it continues to 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: The transaction acquires locks on all items in its data set by their order (line 30). If the item is in the read set (line 33), the transaction confirms that the sequence number in the read set, *seq*, (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).

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

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

```
1: Data read(Item item) {
2:   if item in readset then
3:     dataitem ← readset.get(item) // the item is already in the read set
4:   else
5:     incrementReadCounter(item)
6:     dataitem ← getAbsVal(item) // read the data from the item
7:     readset.add(item,dataitem)
8:   return dataitem.data
9: }

10: write(Item item, Data data) {
11:   if item not in writeset then
12:     writeset.add(item,⟨item,0,0⟩)
13:   writeset.set(item,⟨item,0,data⟩) // set the value to be written in the write set
14: }

15: tryCommit() {
16:   decrementReadCounters() // decrement read counters
17:   if writeset 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 writeset do // commit the changes to the items
24:     dataitem ← owner.writeset.get(item)
25:     WRITE(item.data, dataitem.data)
26:   releaseLocks() // release locks on all the data set
27: }

28: acquireLocks() {
29:   dataset ← writeset.add(readset) // items in the data set (read and write sets)
30:   for each item in dataset by their order do
31:     repeat
32:       cur ← getAbsVal(item) // current value
33:       if item in readset then // check validity of read set
34:         dataitem ← readset.get(item) // value read by the transaction
35:         if dataitem.seq ≠ cur.seq then // the data is overwritten
36:           abort()
37:         return
38:       until CAS(item.lock, ⟨0,cur.seq⟩, ⟨self,cur.seq⟩)
39:     if item in writeset then
40:       dataitem ← writeset.get(item)
41:       writeset.set(item,⟨item,cur.seq+1,dataitem.data⟩)
42: }
```

Pseudocode 2 Additional methods for PermiSTM

```
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] ← writeset.get(i).item.rcounter
47:   repeat no-op
48:   until  $kCSS(kCompare, \langle \text{NULL}, 0 \dots 0 \rangle, \text{COMMITTED})$  // until no reading transactions is pending
49: }

50: incrementReadCounter(Item item) {
51:   repeat  $m \leftarrow \text{READ}(item.rcounter)$ 
52:   until  $\text{CAS}(item.rcounter, m, m + 1)$ 
53: }

54: decrementReadCounters() {
55:   for each item in readset do
56:     repeat  $m \leftarrow \text{READ}(item.rcounter)$ 
57:     until  $!CAS(item.rcounter, m, m - 1)$ 
58: }

59: releaseLocks() {
60:   dataset ← writeset.add(readset)
61:   for each item in dataset do
62:     dataitem ← dataset.get(item)
63:     WRITE(item.lock,  $\langle 0, dataitem.seq \rangle$ )
64: }

65: DataItem getAbsVal(Item item) {
66:   lck ← READ(item.lock)
67:   data ← READ(item.data)
68:   dataitem ←  $\langle item, lck.seq, data \rangle$  // values from the item
69:   if  $lck.owner \neq 0$  then
70:     sts ← READ( $lck.owner.status$ )
71:     if sts = COMMITTED then
72:       if item in  $lck.owner.writeset$  then
73:         dataitem ←  $lck.owner.writeset.get(item)$  // values from the write set of the owner
74:   return dataitem
75: }

76: abort() {
77:   dataset ← writeset.add(readset)
78:   for each item in dataset do
79:     lck ← READ(item.lock)
80:     if  $lck.owner = self$  then // the transaction owns the item
81:       WRITE(item.lock,  $\langle 0, lck.seq \rangle$ ) // release lock
82:   WRITE(status, ABORTED)
83: }
```

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

commitTx method: The transaction collects in an array the k locations to be compared, i.e., the status of the transaction (line 44) and the read counters of items in its write set (line 46). Then, it uses k CSS to set *status* to COMMITTED, while ensuring that all read counters of items in the transaction's write set are 0 (line 48). If the read counter of one of these items is not 0, a pending transaction is reading from this item, thus the transaction spins, until all *rcounters* are 0.

4 Properties of PermiSTM

4.1 Opacity

Since PermiSTM is lock-based, it is simple to argue that it preserves opacity. First, we show that all transactions, including aborted ones, maintain a consistent view. Then, we complete by proving strict view serializability of committed transactions.

It is straightforward from the pseudocode that an update transaction commits only if the read counters of all items in its read set are zero.

Lemma 1 *An update transaction does not commit (line 48) while the read counter of an item in its write set is greater than 0.*

Proof: The transaction uses k CSS to set *status* to COMMITTED, while ensuring that all read counters of items in its write set are 0 (line 48). If the read counter of one of these items is not 0, another pending transaction is reading from this item, and the transaction waits until all *rcounters* are 0. ■

At any point during the execution of a transaction, the *view* of the transaction is a list of item-value pairs, which includes all the items read by the transaction until this point and their values.

Lemma 2 *As long as a transaction T does not abort or call `tryCommit` method, the view of T is consistent and the values in the items has not changed since T read them.*

Proof: The proof is by induction on k , the number of items read by the transaction. The base case is when $k = 0$, and the lemma vacuously holds.

For the induction step, assume that the view of the transaction is consistent after reading k items, $k \geq 1$. Consider d the $(k + 1)$ -th item the transaction is reading. First, The transaction increments the read counter of d (line 5). Next, the transaction reads the current value of d (line 6): the transaction reads the value of d (line 68) and then checks the lock on d (line 69).

If another transaction T' is the owner of d and T' is committed (line 71), the transaction reads the value T' is writing to d from the write set of T' (line 73). By Lemma 1, T' commits before T increases the reader counter of d , and no other transaction writes to d and commits since then. Whether or not

another transaction holds the lock on d , T returns a value that is committed to d before T increases the reader counter of d , or the initial value, in case no transaction writes to d before T .

By the inductive assumption, the first k items are consistent and have not changed their values, therefore the value read from d is consistent with the values of the first k items. Furthermore, T decrements d 's read counter only when the transaction tries to commit or when it aborts. Since we consider only the execution before `abort` and `tryCommit`, then by Lemma 1, no transaction writing to d commits after T returns the value at the end of the read operation, and the lemma holds. ■

Lemma 3 *A transaction T forcibly aborts while executing the `tryCommit` method if and only if another committed transaction writes to an item after T read it.*

Proof: Before reading the current value of an item, a transaction reads the current sequence number of the item. If the value is read from the item (line 68), the sequence number is read from the item as well (line 66). Otherwise, the value and sequence number are read atomically from the write set of the committing transaction (line 73).

The sequence number in the write set of an update transaction is set (line 41) while acquiring the locks on the items (line 20) before committing (line 22). By Lemma 2, the values in the items read by T do not change before T calls the `tryCommit` method. When trying to commit, an update transaction decreases the read counters of its read set (line 16) and acquires locks on all items in its data set (line 20). Other transactions might change the values of the items read by T during this interval, but T verifies that none of the sequence numbers of items in its read set have changed (line 35) when acquiring the locks on its data set; T aborts if any of them has changed (line 36). ■

Theorem 4 *PermiSTM is opaque.*

Proof: By Lemma 2, all aborted transactions have consistent view before aborting. Therefore, to prove opacity it is left to show that all committed transactions are strictly serializable.

By Lemma 3, the values of the items read by a transaction T do not change since T read them and before T commits. This ensures that the view of the transaction is still consistent, and allows serializing update transactions at their commit point. A read-only transaction is serialized when it returns the value of the last read operation by the transaction. ■

4.2 Disjoint-Access Parallelism

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 transaction owning the item, which results in contention only when transactions are not disjoint; 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 transactions read from the same base object only if they all intersect with the owning transaction. Thus, PermiSTM has 2-local contention.

4.3 Progress and Permissiveness

By the code, read-only transactions are never forcibly aborted. Additionally, by Lemma 3, an update transaction forcibly aborts only if a committed transaction writes to an item in its read set.

Theorem 5 *PermiSTM is MV-permissive.*

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 a continuous sequence of transactions reading from its write set; thus, PermiSTM is blocking. However, PermiSTM guarantees strong progressiveness, since transactions are forcibly aborted only due to another committed transaction with a read-after-write conflict. Furthermore, read-only transactions are *obstruction-free* [16]. 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.

5 CAS-Based PermiSTM

This section describes the modifications to PermiSTM that allow to use CAS instead of a *k*CSS primitive. 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 (see proof in the next section). To avoid using *k*CSS when update transactions commit, we shift the responsibility of “notifying” the update transaction that it cannot commit to the read operations, charging them with the task 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 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 data structures used by the algorithm appear in Figure 4. 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,

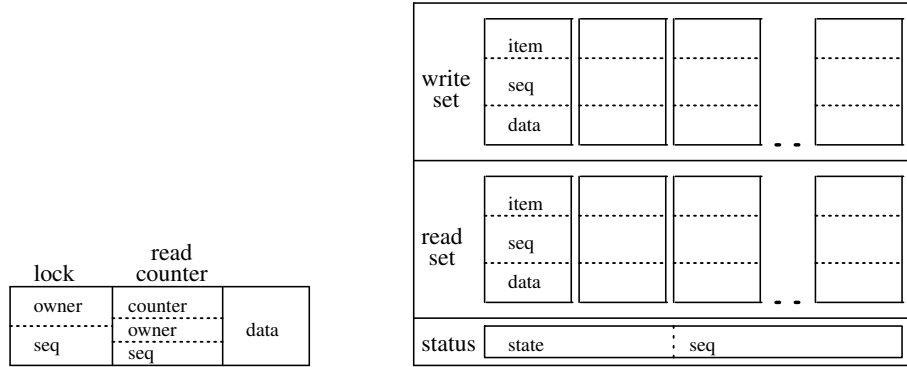


Figure 4: Data structures used in CAS-based PermiSTM.

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 these modifications.

In order to commit, an update transaction waits for the read counter of every item in its write set to become 0 (lines 87-88). When a read counter is 0, the update transactions annotates the 0 with its descriptor and status sequence number (line 89). Finally it sets the status to COMMITTED 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.

A read operation that finds the read counter of the item “marked” (lines 94-95) continues 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, the methods `tryCommit` and `abort` are adjusted to write the state indicator through the new status fields, i.e., by applying `WRITE(status, ⟨val, status.seq+1⟩)` instead of `WRITE(status, val)`, where `val` is COMMITTED or ABORTED.

5.1 Properties of CAS-Based PermiSTM

In order to prove that CAS-based PermiSTM is opaque we revisit Lemma 1; the other proofs hold also for this variant as the relevant code remains the same.

Lemma 1’ *An update transaction does not commit (line 90) while the read counter of an item in its write set is greater than 0.*

Pseudocode 3 methods for CAS-based PermiSTM

```
84: commitTx() {
85:   repeat
86:     sts ← READ(status)
87:     for each item in writeset do
88:       repeat cntr ← READ(item.rcounter) // spin until no readers
89:       until CAS(item.rcounter, ⟨0,cntr.owner,cntr.seq⟩, ⟨0,self,sts.seq⟩)
          // commit in a “quiescent” configuration
90:   until CAS(status, ⟨NULL,sts.seq⟩, ⟨COMMITTED,sts.seq+1⟩)
91: }

92: incrementReadCounter(Item item) {
93:   repeat
94:     cntr ← READ(item.rcounter)
95:     if cntr.owner != 0 then // the read counter is “marked”
96:       CAS(cntr.owner.status, ⟨NULL,cntr.seq⟩, ⟨NULL,cntr.seq+1⟩)
97:   until CAS(item.rcounter, cntr, ⟨cntr.counter+1,cntr.owner,cntr.seq⟩)
98: }

99: decrementReadCounters() {
100:  for each item in readset do
101:    repeat cntr ← READ(item.rcounter)
102:    until CAS(item.rcounter, cntr, ⟨cntr.counter-1,0,0⟩) // clean counter
103: }
```

Proof: Before committing (line 90), an update transaction T reads its status (line 86) and sets its status sequence number in the zero read counter of every item in its write set (lines 87-89). If another transaction T' reads any item in the write set of T after T annotated the read counter (line 89) and before T committed (line 90) then T' uses the sequence number in the read counter to invalidate the status of T (line 96), and therefore T fails to apply the CAS to the status attribute and does not commit. ■

CAS-based PermiSTM is not strictly disjoint-access parallel. Consider three transactions: two transactions, T_1 and T_2 , read items a and b , respectively, while the third transaction, T_3 , updates these items.

T_1 :	read(a)
T_2 :	read(b)
T_3 :	write(a) write(b) try-commit

The data sets of T_1 and T_2 do not intersect, but they may access the same base object, when checking and invalidating the status of T_3 .

CAS-based PermiSTM, however, has 2-local contention and is (strongly) disjoint-access parallel, as this memory contention is always due to T_3 , which intersects both T_1 and T_2 .

CAS-based PermiSTM is MV-permissive, nevertheless, it may have livelocks. In the last example, the read operation of T_1 may invalidate the status of T_3 and then discover that T_3 was fast enough to

complete a full round re-marking the read counter of a with a new sequence number, so the read operation of T_1 fails to increment the read counter of a . However, if T_3 has not committed yet, then the read operation of T_1 may invalidate T_3 once more. This scenario, in which neither the update transaction commits nor any of the read operations increments the read counter, can repeat itself over and over again. The next section shows how to avoid this problem using hardware transactional memory.

6 Using Best-Effort Hardware TM to Avoid Livelocks

Best-effort hardware transactional memory (BEHTM) (cf. [5, 6]) aims at facilitating concurrent programming by extending common architectures with hardware transactions primitives. BEHTM is very effective for short transactions with static data set of minimal size. Current BEHTM must be applied carefully while adhering to several strict limitations, otherwise, a hardware transaction may constantly fail. Most notably, BEHTM limits the number of data items that can be accessed within a transaction; in some cases the maximal number is 4. BEHTM transactions are also limited in terms of the instructions they can execute, since interrupts and exceptions cause hardware transactions to abort.

A hardware transaction that accesses two memory locations atomically provides an escape from the tie-up described at the end of the previous section, for CAS-based PermiSTM; it ensures that at least one read operation avoids the livelock. Instead of applying CAS twice, (lines 96 and 97), the reader (T_1 or T_2 , in our example) uses the hardware transaction HT_1 depicted in Pseudocode 4 to atomically invalidate the status of the transaction owning the item (T_3), while increasing the read counter.⁴ This guarantees that at least the read operation that invalidates the status of the update transaction succeeds to increase the counter of the item it is reading.

To avoid complications due to transactional and non-transactional accesses to the same variable [2, 11], other methods accessing the status of the transaction and read counters of items are replaced with best-effort hardware transaction code; this is also depicted in Pseudocode 4.

Next we show that the conditions that reduce the probability of transactions to spuriously abort hold in our hardware transactions. Transactions HT_3 , HT_4 and HT_5 read and write to a single memory location; transaction HT_2 reads a single memory location; and HT_1 reads and writes to two memory locations. Clearly this does not exceed the limit on the number of memory locations a hardware transaction may access. Additionally, none of the transactions performs prohibited instructions, such as division, remote or system calls. Since the transactions are very short they are less likely to have interrupts or hardware exceptions during their execution.

If, for some reason, one of the hardware transactions spuriously aborts, we cannot guarantee progress for any of the read operations. However, best-effort TM guarantees consistency, so the safety of the algorithm is ensured, that is, BEHTM-based PermiSTM is opaque.

⁴A simpler alternative is to use DCAS, but it is more likely that architectures will support BEHTM than DCAS in the near future.

Pseudocode 4 Best-effort hardware transaction code replacing methods in the CAS-based PermiSTM.

```
incrementReadCounter(Item item) {
  repeat
    BETM_BEGIN  $HT_1$ 
    cntr  $\leftarrow$  BETM_READ(item.rcounter)
    if cntr.owner  $\neq$  0 then
      sts  $\leftarrow$  BETM_READ(cntr.owner.status)
      if sts =  $\langle$ NULL,cntr.seq $\rangle$  then
        // The atomicity of the next two writes avoids the livelock
        BETM_WRITE(cntr.owner.status,  $\langle$ NULL,cntr.seq+1 $\rangle$ )
        BETM_WRITE(item.rcounter,  $\langle$ cntr.counter+1,cntr.owner,cntr.seq $\rangle$ )
      else
        BETM_WRITE(item.rcounter,  $\langle$ cntr.counter+1,cntr.owner,cntr.seq $\rangle$ )
    BETM_COMMIT  $HT_1$ 
  until  $HT_1$  is COMMITTED
}

commitTx() {
  repeat
    BETM_BEGIN  $HT_2$ 
    sts  $\leftarrow$  BETM_READ(status)
    BETM_COMMIT  $HT_2$ 
    for each item in writeset do
      repeat
        BETM_BEGIN  $HT_3$ 
        cntr  $\leftarrow$  BETM_READ(item.rcounter)
        if cntr.rcounter  $\neq$  0 then
          BETM_ABORT  $HT_3$ 
          BETM_WRITE(item.rcounter,  $\langle$ 0,self,sts.seq $\rangle$ )
          BETM_COMMIT  $HT_3$ 
        until  $HT_3$  is COMMITTED
      BETM_BEGIN  $HT_4$ 
      sts4  $\leftarrow$  BETM_READ(status)
      if sts4  $\neq$  sts then
        BETM_ABORT  $HT_4$ 
        BETM_WRITE(status,  $\langle$ COMMITTED,sts.seq+1 $\rangle$ )
        BETM_COMMIT  $HT_4$ 
    until  $HT_4$  is COMMITTED
}

decrementReadCounters() {
  for each item in readset do
    repeat
      BETM_BEGIN  $HT_5$ 
      cntr  $\leftarrow$  BETM_READ(item.rcounter)
      BETM_WRITE(item.rcounter,  $\langle$ cntr.counter-1,0,0 $\rangle$ )
      BETM_COMMIT  $HT_5$ 
    until  $HT_5$  is COMMITTED
}
```

7 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.

In PermiSTM, update transactions are not *obstruction free* [16], since they may block due to other conflicting transactions. Indeed, a single-version, obstruction-free STM cannot be *strictly* disjoint-access parallel [13].

Read-only transactions in PermiSTM modify the read counters of all items in their read set incurring a cost of $O(k)$. This matches the lower bound for read-only transactions that never abort, for (strongly) disjoint-access parallel STMs [3].

Section 5 describes a CAS-based variant of PermiSTM that removes the need of k CSS by explicitly changing the implementation of the `commitTx` method and the methods that access the read counters. An alternative is to use the k CSS implementation from CAS [21]. They implement LL, SC operations from CAS primitives, and use them to change the value of the first location, and to get a SNAPSHOT operation to confirm that the values in the other locations match the expected values. The SNAPSHOT operation repeatedly collects the values from the locations until two successive collects have the same values. Being a general purpose implementation, the k CSS operation first takes a snapshot of all the locations, and only then confirms they match the expected values. This incurs an overhead that can be avoided in PermiSTM, which looks for a very specific snapshot in which all counters are zero.

The design of PermiSTM allows to decompose the algorithm in a fairly straightforward manner, to use short hardware transactions that access at most two memory locations. This guarantees progress even in the face of contention (as shown in Section 6), while the implementation of [21] may suffer from livelocks when k CSS operations obstruct each other.

Our work joins recent efforts to exploit the capabilities of best-effort hardware transactional memory [5, 6], and suggests how it can be utilized to avoid livelocks in a generic k CSS implementation.

Several design principles of PermiSTM are inspired by TLRW [8], which uses *read-write* locks. The lock contains a byte per each *slotted* reader, and a reader-count that is modified by other, *unslotted* readers. 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 [20]) instead of read locks.

UP-MV STM [24] maintains many versions for each data item to be MV-permissive. It keeps the shortest suffix of version that is required to allow all active read-only transaction to commit, and removes old versions that are no longer required. To allow this, the implementation holds a global transaction set, with the descriptors of all completed transactions yet to be collected by the garbage collection mechanism, and maintains the transactions precedence graph by keeping pointers between transactions. Our algorithm improves on the multi-version UP-MV STM [24], which is not weakly disjoint-access parallel (nor strictly disjoint-access parallel), as it uses the global set of completed transactions. 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 [7] could make the algorithm blocking.

References

- [1] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *PODC '97*, pages 111–120.
- [2] H. Attiya and E. Hillel. The cost of privatization. In *DISC '10*, pages 35–49.
- [3] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *SPAA '09*, pages 69–78.
- [4] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT '08*.
- [5] F. Carouge and M. Spear. A scalable lock-free universal construction with best effort transactional hardware. In *DISC '10*, pages 50–63.
- [6] 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*, pages 325–334.
- [7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06*, pages 194–208.
- [8] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *SPAA '10*, pages 284–293.
- [9] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, 2006.
- [10] V. Gramoli, D. Harmanci, and P. Felber. Towards a theory of input acceptance for transactional memories. In *OPODIS '08*, pages 527–533.
- [11] R. Guerraoui, T. A. Henzinger, M. Kapalka, and V. Singh. Transactions in the jungle. In *SPAA '10*, pages 263–272.
- [12] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC '08*, pages 305–319.
- [13] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *SPAA '08*, pages 304–313.
- [14] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08*, pages 175–184.
- [15] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *POPL '09*, pages 404–415.
- [16] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03*, pages 92–101.
- [17] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC '94*, pages 151–160.
- [18] M. Kapalka. *Theory of Transactional Memory*. PhD thesis, EPFL, 2010.
- [19] I. Keidar and D. Perelman. On avoiding spurious aborts in transactional memory. In *SPAA '09*, pages 59–68.
- [20] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09*.
- [21] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *SPAA '03*, pages 314–323.

- [22] J. Napper and L. Alvisi. Lock-free serializable transactions. Technical Report TR-05-04, The University of Texas at Austin, 2005.
- [23] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.
- [24] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC '10*, pages 16–25.
- [25] D. Perelman and I. Keidar. SMV: Selective Multi-Versioning STM. In *TRANSACT '10*.
- [26] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06*, pages 284–298.
- [27] 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*, pages 187–197.