Theory Comput Syst (2012) 51:425–446
DOI 10.1007/s00224-012-9406-3

# A Single-Version STM that Is Multi-Versioned Permissive

**Hagit Attiya · Eshcar Hillel**

Published online: 5 May 2012


**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 (writing to a common data item), thereby avoiding spurious aborts. PermiSTM also avoids unnecessary contention on the memory, being *disjoint-access parallel*.

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

**Keywords** Transactional memory · Versions · Permissiveness · Disjoint access parallelism · Compare-and-swap · *k*-compare-single-swap · Best-effort hardware transactional memory

## 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

A preliminary version of this paper appeared in ICDCN 2011.
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.

H. Attiya (✉) · E. Hillel
Department of Computer Science, Technion, Haifa, Israel
e-mail: hagit@cs.technion.ac.il

E. Hillel
Yahoo! Labs, Haifa, Israel
e-mail: eshcar@yahoo-inc.com

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 [15]. 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 [13, 15, 17, 21, 27]. One of these conditions, called *multi-versioned* (*MV*) *permissiveness* [27], focuses on *read-only* transactions (whose write set is empty), and ensures they never abort; *update* transactions, with nonempty write set, may abort when in conflict with other transactions writing to the same data items. As its name suggests, multi-version permissiveness was meant to be provided by a *multi-version* STM, maintaining multiple versions of each data item. It has been suggested [27] 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 [5, 12, 27, 28] and more theoretical [21, 25], keeping many versions per data item. It can be achieved by other multi-version STMs [4, 30], if sufficiently many versions of the data items are maintained.

This paper shows that 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 MV-permissive, indicating that multiple versions are not the only design choice when seeking to reduce spurious aborts. By maintaining a single version, PermiSTM avoids the space complexity associated with keeping many versions. This also eliminates the need for intricate mechanisms of keeping and garbage collecting old versions.

PermiSTM is *lock-based*, like many contemporary STMs, e.g., [5, 8, 9, 11, 31]. 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 data 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 committed transaction, i.e., at least one of the conflicting transactions commits. Although it is blocking, PermiSTM is deadlock-free, i.e., some transaction can always 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 [9, 22], 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 strongly *disjoint-access parallel* [3, 19], namely, processes executing transactions with disconnected data sets do not contend on the same base objects (see formal definition in Sect. 2.2).

It has been proved [27, Theorem 2] that a *weakly disjoint-access parallel* STM [3] cannot be MV-permissive. PermiSTM, satisfying the even stronger property of strong 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 or even block 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, Sect. 3 presents a variant in which transactions that are not read-only use a *k-compare-single-swap* (*k*CSS) primitive [23] at commit-time; the properties of the algorithm are discussed in Sect. 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 Sect. 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 Sect. 7.

## 2 Preliminaries

We briefly describe the transactional memory approach. We concentrate on describing the features that are required to present and prove our algorithms; more details on this model can be found in [20].

A *transaction* is a sequence of operations executed by a single process. We have read and write operations on *data items* and operations to commit or abort the transaction: A *read* operation specifies the data item to read, and returns the value read by the operation; a *write* operation specifies the data item and a 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.

A transaction begins with a sequence of read and write operations, possibly followed by a try-commit or abort; a transaction has at most one try-commit or abort operation as its last operation (but not both). If the last operation of a transaction is try-commit or abort, then the transaction is *complete*, and is said to be *committed* or *aborted*, respectively; otherwise, the transaction is *pending*.

The collection of data items accessed by a transaction is its *data set*; the data items it writes are its *write set*, and the data items its reads are its *read set*. A *read-only* transaction accesses the memory only through read operations (its write set is empty); otherwise, it is an *update* transaction.

Two operations *conflict* if they are on the same data item; the conflict is *trivial* if both are read operations, and *nontrivial* otherwise. Two overlapping transactions

**Fig. 1** The CAS primitive

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

*conflict* if they include operations that conflict; the conflict is trivial or nontrivial, depending on the type of conflict between the operations.

While trying to commit, a transaction might be aborted, in which case, we say it is *forcibly aborted*.[1]

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.

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

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 say that the process *accesses* the base objects.

A *configuration* is a complete description of the system at some point, i.e., the state of each process and the state of each shared base object. In the unique *initial* configuration, every process is in its initial state and every base object contains its initial value.

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

The *interval of a transaction* $T$ in an execution $\alpha$ is the execution interval that starts at the first event of $T$, and ends with the last event of $T$, if $T$ completes in $\alpha$, or with the end of $\alpha$, if $T$ does not complete in $\alpha$. Note that if $\alpha$ is infinite and $T$ does not complete in $\alpha$, then the interval of $T$ is infinite.

Two transactions *overlap* if their intervals overlap; otherwise, they are non-overlapping.

## 2.1 Multi-Version Permissiveness

The following condition restricts the situations in which a transaction is forcibly aborted.

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

**Definition 1** An STM is *multi-versioned (MV-)permissive* [27] if a transaction is forcibly aborted only if it is an update transaction that has a nontrivial conflict with another update transaction.

Specifically, read-only transactions are never forcibly aborted in an MV-permissive STM.

### 2.2 Disjoint-Access Parallelism (DAP)

*Disjoint-access parallelism* [19] captures the intuition that transactions on disjoint parts of the data should not interfere with each other. The *conflict graph* of an execution interval captures the distance between overlapping transactions by representing (trivial and nontrivial) conflicts between transactions that overlap in time. The vertices in the conflict graph represent transactions, and edges connect two overlapping transactions with intersecting data sets.

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

Two processes *concurrently access* a base object $o$ if both have pending accesses to $o$ at some configuration; they *concurrently contend* on $o$ if one of these accesses is not a read.

An STM is *weakly disjoint-access parallel* if two processes $p_1$ and $p_2$, executing transactions $T_1$ and $T_2$, respectively, concurrently contend on the same base object, only if $T_1$ and $T_2$ are not disjoint-access. Note that this allows $p_1$ and $p_2$ to concurrently read the same object even if they are disjoint-access. In contrast, 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. Clearly, a strongly disjoint-access parallel STM is also weakly disjoint-access parallel (but not necessarily vice versa).

## 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 data item, tracking the number of pending transactions reading from the data item. Transactions that write to the data items respect the read counters; an update transaction commits and updates the data items in its write set only in a "quiescent" configuration, where no (other) pending transaction is reading a data 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 use 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 data 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 execution, a transaction decrements the read counters of the data 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 $T_2$ increment 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 data 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 ensures consistency by acquiring locks also for data items in its read set. Therefore, before committing, an update transaction first decrements its read counters, and then acquires locks on *all* data 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 decrement the read counter of $a$; MV-permissiveness does not allow either transaction to be forcibly aborted. In order to avoid the deadlock, read counters get preference over locks, and they can always be incremented.

Since committing a transaction and writing its updates to items in its write set are not done atomically, a committed transaction that has not yet completed updating all the data items in its write set, can yield an inconsistent view for a transaction reading one of these data items. If a read operation simply reads the value in the data item, it might miss the up-to-date value of the data item. Consider the following example: $T_1$ writes to data items $a$ and $b$, and $T_2$ reads these data items after $T_1$ committed. If $T_1$ writes the new values to the data items after $T_2$ reads the value from data item $a$, but before $T_2$ reads the value from data 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($b$) | | |

**Fig. 2** The *k-compare-single-swap* primitive

```
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
}
```

Therefore, a read operation has to read the *current* value of the data item, which can be found either in the data item, or in the data set of the transaction.[2] In the example, this means that when $T_2$ reads a data item, and it discovers that $T_1$, the owner of the data item, has committed, $T_2$ reads the new value of the data 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 data items.

To simplify the exposition of PermiSTM, an update transaction is committed while ensuring that the read counters of the data items in its write set are all zero, using a *k-compare-single-swap* primitive (*k*CSS). The *k*CSS primitive [23] is similar to CAS, but compares the values of $k$ independent base objects (see Fig. 2). Later, we describe how the implementation is modified to use only CAS.

### 3.1 Data Structures

Figure 3 presents the data structures of data items and transactions' descriptors used in our algorithm. We associate a lock and a read counter with each data 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 data item, and is used to assert the consistency of reads.
- A simple *read counter*, *rcounter*, tracks how many pending transactions are reading the data item.
- The *data* field holds the value that was last written to the data item, or its initial value if no transaction has written to the data item yet.

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 data 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 data 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.

[2] This is analogous to the notion of current version of a transactional object in DSTM [18].

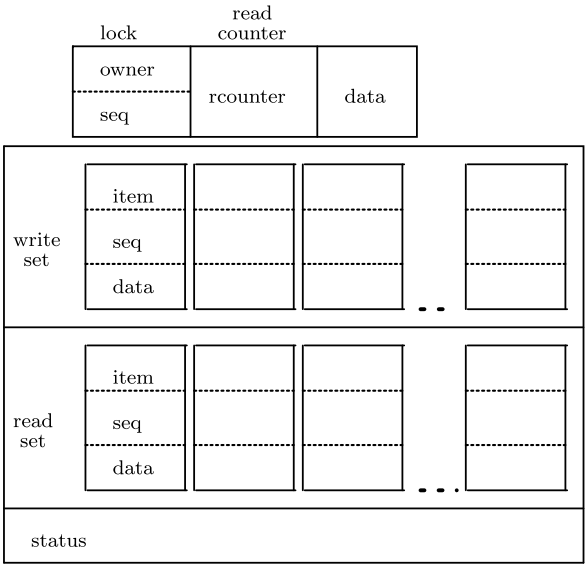


**Fig. 3** Data structures used in the algorithm: an item (*top*) and a transaction descriptor (*bottom*)

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

### 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 data item is already in the transaction's write set (line 2), the transaction returns the value from the write set (line 3). If the data item is already in the transaction's read set (line 4), the transaction returns the value from the read set (line 5). Otherwise, it increments the read counter of the data item (line 7). Then, the reading transaction adds the data item to its read set (line 9) with the *current* value and sequence number of the data item (line 8).

write *method*: If the data item is not already in the transaction's write set (line 13), the transaction adds the data item to its write set (line 14). The transaction sets *data* of the data item in the write set to the new value to be written (line 15). *No lock is acquired at this stage*.

tryCommit *method*: The transaction decrements all the read counters of the data items in its read set (line 18). If the transaction is *read-only*, i.e., the write set of the transaction is empty (line 19), then the transaction commits (line 20) and returns (line 21).

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

```
 1: Data read(Item item) {
 2:     if item in writeset then
 3:         dataitem ← writeset.get(item) // the data item is already in the write set
 4:     else if item in readset then
 5:         dataitem ← readset.get(item) // the data item is already in the read set
 6:     else
 7:         incrementReadCounter(item)
 8:         dataitem ← getCurVal(item) // read the data from the data item
 9:         readset.add(item,dataitem)
10:     return dataitem.data
11: }

12: write(Item item, Data data) {
13:     if item not in writeset then
14:         writeset.add(item,⟨item,0,0⟩)
15:     writeset.set(item,⟨item,0,data⟩) // set the value to be written in the write set
16: }

17: tryCommit() {
18:     decrementReadCounters() // decrement read counters
19:     if writeset is empty then // read-only transaction
20:         WRITE(status, COMMITTED)
21:         return
        // update transaction
22:     acquireLocks() // lock all the data set
23:     if ABORTED = READ(status) then return
24:     commitTx() // commit update transaction
25:     for each item in writeset do // commit the changes to the items
26:         dataitem ← owner.writeset.get(item)
27:         WRITE(item.data, dataitem.data)
28:     releaseLocks() // release locks on all the data set
29: }

30: acquireLocks() {
31:     dataset ← writeset.union(readset) // items in the data set (read and write sets)
32:     for each item in dataset by their order do
33:         repeat
34:             cur ← getCurVal(item) // current value
35:             if item in readset then // check validity of read set
36:                 dataitem ← readset.get(item) // value read by the transaction
37:                 if dataitem.seq != cur.seq then // the data is overwritten
38:                     abort()
39:                     return
40:         until CAS(item.lock, ⟨0,cur.seq⟩, ⟨self,cur.seq⟩)
41:         if item in writeset then
42:             dataitem ← writeset.get(item)
43:             writeset.set(item,⟨item,cur.seq+1,dataitem.data⟩)
44: }
```

**Pseudocode 2** Additional methods for PermiSTM

```
45: commitTx() {
46:     k ← writeset.size()+1
47:     kCompare[0] ← status // the location to be compared and swapped
48:     for item in writeset do // k − 1 locations to be compared
49:         kCompare[i] ← writeset.get(item).rcounter
50:     repeat no-op
51:     until kCSS(kCompare, ⟨NULL,0...0⟩, COMMITTED) // until no reading transactions
            is pending
52: }

53: incrementReadCounter(Item item) {
54:     repeat m ← READ(item.rcounter)
55:     until CAS(item.rcounter, m, m + 1)
56: }

57: decrementReadCounters() {
58:     for each item in readset do
59:         repeat m ← READ(item.rcounter)
60:         until CAS(item.rcounter, m, m − 1)
61: }

62: releaseLocks() {
63:     dataset ← writeset.union(readset)
64:     for each item in dataset do
65:         dataitem ← dataset.get(item)
66:         WRITE(item.lock, ⟨0,dataitem.seq⟩)
67: }

68: DataItem getCurVal(Item item) {
69:     lck ← READ(item.lock)
70:     data ← READ(item.data)
71:     dataitem ← ⟨item, lck.seq, data⟩ // values from the data item
72:     if lck.owner != 0 then
73:         sts ← READ(lck.owner.status)
74:         if sts = COMMITTED then
75:             if item in lck.owner.writeset then
76:                 dataitem ← lck.owner.writeset.get(item) // values from the write set of
                        the owner
77:     return dataitem
78: }

79: abort() {
80:     dataset ← writeset.union(readset)
81:     for each item in dataset do
82:         lck ← READ(item.lock)
83:         if lck.owner = self then // the transaction owns the data item
84:             WRITE(item.lock, ⟨0,lck.seq⟩) // release lock
85:     WRITE(status, ABORTED)
86: }
```

Otherwise, this is an *update* transaction and it acquires locks on all data items in the data set (line 22); commit the transaction (line 24) and the changes to the data items (lines 25–27); and release locks on all data items in the data set (line 28). The transaction may abort while acquiring locks due to a nontrivial conflict with another update transaction (line 23).

acquireLocks *method*: The transaction acquires locks on all data items in its data set by their order (line 32). If the data item is in the read set (line 35), the transaction confirms that the sequence number in the read set, *seq*, (line 36) is the same as the current sequence number of the data item (line 34). If the sequence number has changed (line 37) then the data read is overwritten by another committed transaction and the transaction aborts (line 38). *This is the only case in which a transaction aborts.*

The transaction uses CAS to acquire the lock: it swaps *owner* from 0 to the descriptor of the transaction; if the data item is in its read set this is done while asserting that *seq* is unchanged (line 40). 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 40) until *owner* is 0.

If the data item is in the write set of the transaction (line 41), then it sets *seq* to the sequence number of the current value plus 1 (line 43).

commitTx *method*: The transaction computes $k$, which is the size of the write set plus one (line 46), collects in an array the $k$ locations to be compared, i.e., the status of the transaction (line 47) and the read counters of data items in its write set (line 49). Then, it uses $k$CSS to set *status* to COMMITTED, while ensuring that all read counters of data items in the transaction's write set are 0 (line 51). If the read counter of one of these data items is not 0, a pending transaction is reading from this data item, thus the transaction spins, until all *rcounter*s are 0.

## 4 Properties of $k$CSS-Based PermiSTM

### 4.1 Safety

Proving the safety of PermiSTM is simplified since it is lock-based. For each committed transaction, we identify a *serialization point* inside the interval of the tryCommit method, and show that committed transactions appear to execute alone at their serialization point. Moreover, we prove that the intermediate views of all transactions (including aborted ones) are consistent with these serialization points (in a sense that is made precise below).

For the rest of the proof, fix an execution $\alpha$ of the algorithm.

The serialization point of a read-only transaction is when it calls the tryCommit method; in particular, the serialization point is after the return of the last read operation in the transaction. The serialization point of an update transaction is when its status is set to COMMITTED (line 51).

The serialization points induce a *serialization* $s$ of the execution [26], namely, a sequential execution of committed transactions, one at a time, in which each transaction executes alone at its serialization point. Note that the serialization $s$ preserves

the order of non-overlapping transactions (since the serialization point is inside the transaction's interval).

A *prefix* $T'$ of a transaction $T$ is an initial sequence of read and write operations of the transaction. The *intermediate view* of the transaction $T$ after the prefix $T'$ is a sequence of item-value pairs, containing data items and values read or written in $T'$, in the order of the operations; below, this is abbreviated *the view of* $T'$. The prefix $T'$ is *consistent* with the serialization $s$ if the same view is obtained when $T'$ is executed alone after $s'$, the prefix of $s$ including all the transactions whose serialization point is in the shortest prefix of $\alpha$ that includes $T'$. Intuitively, this means that there is a *virtual* serialization point, in which the transaction (locally) considers itself to execute the prefix $T'$ alone; however, none of the changes in $T'$ are observed by other transactions, until $T$ commits.

We show that any prefix of a transaction $T$ is consistent with $s$. $T$ might abort later, but as long as it is executing, its prefixes are consistent with the serialization $s$. We further show that the result of a committed transaction is as if the transaction executed alone at its serialization point.

The first, simple lemma shows that an update transaction commits only if the read counters of all data items in its read set are zero. This follows from the code, since the transaction uses $k$CSS to set *status* to COMMITTED, while ensuring that all read counters of data items in its write set are 0 (line 51).

**Lemma 1** *An update transaction does not commit* (*by a successful execution of* $k$CSS *in line* 51) *as long as the read counter of a data item in its write set is greater than* 0.

**Lemma 2** *Every prefix* $T'$ *of a transaction* $T$ *is consistent with the serialization* $s$.

*Proof* The proof is by induction on $k$, the number of operations in $T'$. The base case is when $k = 0$, and the claim vacuously holds.

For the induction step, assume that the prefix $T''$ with the first $k \geq 0$ operations of $T$, is consistent with $s$, namely, the same view is obtained when $T''$ is executed alone after $s''$, the prefix of $s$ that includes all transactions whose serialization point is in the shortest prefix of $\alpha$ that includes $T''$.

Let $T'$ be the prefix with the first $k + 1$ operations of $T$. Let $s'$ be the prefix of $s$, extending $s''$, which includes all transactions whose serialization point is in the shortest prefix of $\alpha$ that includes $T'$.

The read counter of any item that $T$ stores in its read set during the prefix $T''$ is greater than 0, until $T$ decrements its read counter in tryCommit. Therefore, no transaction writing to this item commits (by Lemma 1), and its current value does not change. Therefore, the view obtained when $T''$ is executed alone after $s'$ is the same as when $T''$ is executed alone after $s''$.

Consider the $(k + 1)$-st operation of $T$. The view of the prefix $T''$ determines the type of operation (read or write), the data item $d$ to which it is applied, and the value written (if this is a write operation).

If the $(k + 1)$-st operation is a write operation, it writes the same value to the same data item as in the sequential execution corresponding to $s'$. Thus, $T'$ is consistent with the prefix $s'$ of $s$, and the lemma holds.

For the rest of the proof, assume that the $(k+1)$-st operation reads from $d$. If there is a write or a read operation to $d$ in $T''$, then the value of $d$ is returned from the write set (line 3) or the read set (line 5), respectively, and the lemma holds by the induction hypothesis.

Otherwise, $T$ increments the read counter of $d$ (line 7) and calls getCurVal (line 8). We show that getCurVal returns the current value of $d$ (as defined at the end of Sect. 3.1). $T$ reads the lock and sequence number (line 69), and the value of $d$ (line 70); if no transaction owns $d$, $T$ returns the value read from the data item (line 71). If a transaction $T_1$ owns $d$ (line 72) and $T_1$ is committed (line 74), $T$ returns the value $T_1$ writes to $d$ from the write set of $T_1$ (line 76).

$T$ increments the read counter of $d$ (line 7) before calling getCurVal; after that, the read counter of $d$ is greater than 0 until $T$ calls the tryCommit method. By Lemma 1, $T_1$ commits before $T$ increases the read counter of $d$ and no other transaction writes to $d$ and commits since then. Thus, $T$ returns the value that is written to $d$ by the committed transaction with the latest serialization point preceding $T'$, i.e., the latest write to $d$ in $s'$, or the initial value, if no transaction writes to $d$ in $s'$. Together with the fact that the view of $T''$ is consistent with the prefix $s'$, we get that the view of $T'$ is consistent with $s'$. □

The next lemma shows that a write-after-read conflict is a necessary and sufficient condition for a transaction to be forcibly aborted. The sufficiency is used to complete the safety proof, while the necessity is used later, to show that PermiSTM is MV-progressiveness.

**Lemma 3** *A transaction $T$ is forcibly aborted in its* tryCommit *method if and only if another committed transaction writes to a data item after $T$ read it.*

*Proof* If the value of a data item is read from the data item (line 71), then the current sequence number is read from the data item as well (line 69). Otherwise, the value and sequence number are read atomically from the write set of the committing transaction (line 76).

In the tryCommit method, $T$ verifies that none of the sequence numbers of data items in its read set have changed (line 37) when acquiring the locks on its data set. This includes sequence numbers $T$ reads from the write set of a committed transaction, as the sequence number in the write set of an update transaction is set (line 43) while acquiring the locks on the data items (line 22) before committing (line 24). $T$ aborts if and only if one of them changes (line 38), indicating another committed transaction writes to the data item after $T$ read it. □

Lemma 2 ensures that every prefix of a transaction $T$ is consistent with the serialization, even if the transaction is aborted. The lemma holds until $T$ calls its tryCommit method. If $T$ commits successfully, the next lemma carries the consistency into the tryCommit method.

**Lemma 4** *A committed transaction appears to execute alone at its serialization point.*

*Proof* By Lemma 2, the view of $T$ after its last read operation is as if $T$ is executed alone after a prefix of the serialization $s$ that includes the transactions whose serialization point are before $T$ calls the tryCommit method. If $T$ is read-only, its serialization point is when it calls tryCommit and hence, $T$ appears to execute alone at its serialization point.

If $T$ is an update transaction, $T$ decrements the read counters of its read set (line 18) and then acquires locks on all data items in its data set (line 22).

By Lemma 3, if another transaction changes the values of a data item read by $T$ during this interval, then $T$ aborts. Therefore, if $T$ commits then the values were not changed and the view of $T$ is as if $T$ is executed alone after the prefix of $s$ that includes the transactions whose serialization point are before $T$ locks all items in its data set. Since the items in the read set are locked, their values do not change until $T$ sets its status to COMMITTED (which is the serialization point of $T$). Since the items in the write set are locked, their value is updated without interference from other transactions. Therefore, $T$ appears to execute alone at its serialization point. □

### 4.2 Disjoint-Access Parallelism

All operations may access data items in the data set of the transaction. Write and abort operations additionally access only the descriptor of the transaction; this may result in contention only between transactions with non-disjoint data sets. Read and try-commit operations, in addition read the descriptor of the transaction owning the data item. The code implies that a transaction owns an item only if it is in its data set; thus, PermiSTM is strongly disjoint-access parallel (and hence also weakly disjoint-access parallel).

In fact, $k$CSS-based PermiSTM is even *strictly disjoint-access parallel* [16], namely, processes executing transactions with disjoint data sets do not contend on the same memory location. Note that transactions with disjoint data sets may concurrently read the descriptor of a third transaction owning items the transactions read; this, however, is allowed by strict disjoint-access parallelism.

### 4.3 MV-Permissiveness

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

**Theorem 1** *PermiSTM is MV-permissive.*

A read-only transaction is also *obstruction-free* [18]: it may delay due to contention with concurrent transactions, updating the same read counters, but once it is running solo it is guaranteed to commit.

After an update transaction acquires locks on all data items in its data set it may wait for other transactions reading data 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, $k$CSS-based PermiSTM guarantees *strong progressiveness* [17], namely, even when there are nontrivial conflicts, at least one of the

transactions involved is not aborted. This holds because transactions are forcibly aborted only due to another committed transaction with a nontrivial read-after-write conflict.

## 5 CAS-Based PermiSTM

This section describes the modifications to PermiSTM that allow the use of 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 a data 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 data item in its write set: spin on the data 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 data item. The only change is when it encounters a "marked" counter. If the update transaction annotating the data 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 a data 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 Fig. 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, *rcount*, of a data item is a tuple including a *counter* of the number of readers, the *owner* transaction of the data 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 data item's read counter. Pseudocode 3 presents these modifications.

In order to commit, an update transaction waits for the read counter of every data item in its write set to become 0 (lines 90–91). When a read counter is 0, the update

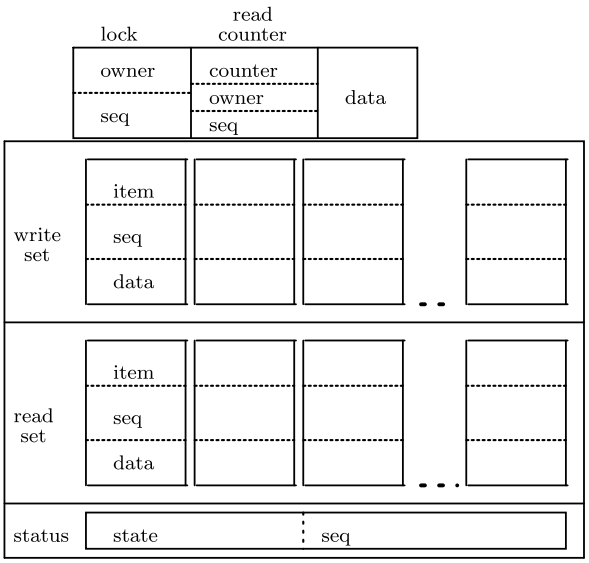


**Fig. 4** Data structures used in CAS-based PermiSTM

transactions annotates the 0 with its descriptor and status sequence number (line 92). 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 89) and goes over the procedure again. A successful CAS implies that the transaction committed while no other transaction is reading any data item in its write set.

A read operation that finds the read counter of the data item "marked" (lines 97–98) continues as follows: use CAS to invalidate the status of the owner transaction—by increasing its sequence number (line 99), 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 read counter using CAS (line 100). If increasing the read 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 105).

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 the safety property of CAS-based PermiSTM, 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* 93) *while the read counter of a data item in its write set is greater than* 0.

*Proof* Before committing (line 93), an update transaction $T$ reads its status (line 89) and sets its status sequence number in the zero read counter of every data item in its

**Pseudocode 3** Methods for CAS-based PermiSTM

```
87:  commitTx() {
88:      repeat
89:          sts ← READ(status)
90:          for each item in writeset do
91:              repeat cntr ← READ(item.rcounter) // spin until no readers
92:              until CAS(item.rcounter, ⟨0,cntr.owner,cntr.seq⟩, ⟨0,self,sts.seq⟩)
             // commit in a "quiescent" configuration
93:      until CAS(status, ⟨NULL,sts.seq⟩, ⟨COMMITTED,sts.seq+1⟩)
94:  }

95:  incrementReadCounter(Item item) {
96:      repeat
97:          cntr ← READ(item.rcounter)
98:          if cntr.owner != 0 then // the read counter is "marked"
99:              CAS(cntr.owner.status, ⟨NULL,cntr.seq⟩, ⟨NULL,cntr.seq+1⟩)
100:     until CAS(item.rcounter, cntr, ⟨cntr.counter+1,cntr.owner,cntr.seq⟩)
101: }

102: decrementReadCounters() {
103:     for each item in readset do
104:         repeat cntr ← READ(item.rcounter)
105:         until CAS(item.rcounter, cntr, ⟨cntr.counter−1,0,0⟩) // clean counter
106: }
```

write set (lines 90–92). If another transaction $T'$ reads any data item in the write set of $T$ after $T$ annotated the read counter (line 92) and before $T$ committed (line 93) then $T'$ uses the sequence number in the read counter to invalidate the status of $T$ (line 99), and therefore $T$ fails to apply the CAS to the status attribute and does not commit. □

Consider three transactions: two transactions, $T_1$ and $T_2$, read data items $a$ and $b$, respectively, while the third transaction, $T_3$, updates these data 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 contend on the same base object, when checking and possibly invalidating the status of $T_3$.

CAS-based PermiSTM is strongly disjoint-access parallel, as this memory contention is always due to $T_3$, which intersects both $T_1$ and $T_2$. (In the terminology of [1] this means that CAS-based PermiSTM has 2-local contention.)

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. [6, 7]) aims to facilitate concurrent programming by extending common architectures with hardware support for transactions. BEHTM is particularly 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 [6]. 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 99 and 100), 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 data item ($T_3$), while increasing the read counter.[3] This guarantees that at least the read operation that invalidates the status of the update transaction succeeds in increasing the counter of the data item it is reading.

To avoid complications due to transactional and non-transactional accesses to the same variable [2, 14], other methods accessing the status of the transaction and read counters of data items are replaced with best-effort hardware transaction code; this is also depicted in Pseudocode 4.

Note that our hardware transactions adhere to the limitations, mentioned above, which reduce the chances of transactions to spuriously abort. 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 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 preserved.

## 7 Discussion

This paper presents PermiSTM, a single-version STM that is MV-permissive and disjoint-access parallel. PermiSTM has simple design, based on read counters and

[3] An alternative is to use DCAS, but we wanted to explore how to use the recently proposed architectures supporting BEHTM [10, 24, 29].

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

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

commitTx() {
    repeat
        repeat
            BETM_BEGIN HT_2
            sts ← BETM_ READ(status)
            BETM_COMMIT HT_2
        until HT_2 is COMMITTED
        for each item in writeset do
            repeat
                BETM_BEGIN HT_3
                cntr ← BETM_ READ(item.rcounter)
                if cntr.rcounter != 0 then
                    BETM_ABORT HT_3
                BETM_WRITE(item.rcounter, ⟨0,self,sts.seq⟩)
                BETM_COMMIT HT_3
            until HT_3 is COMMITTED
        BETM_BEGIN HT_4
        sts_4 ← BETM_ READ(status)
        if sts_4 != sts then
            BETM_ABORT HT_4
        BETM_WRITE(status, ⟨COMMITTED,sts.seq+1⟩)
        BETM_COMMIT HT_4
    until HT_4 is COMMITTED
}

decrementReadCounters() {
    for each item in readset do
        repeat
            BETM_BEGIN HT_5
            cntr ← BETM_ READ(item.rcounter)
            BETM_WRITE(item.rcounter, ⟨cntr.counter−1,0,0⟩)
            BETM_COMMIT HT_5
        until HT_5 is COMMITTED
}
```

locks, to provide consistency without incremental validation. This also simplifies the correctness argument.

In PermiSTM, update transactions are not *obstruction free* [18], since they may block due to other transactions with a nontrivial conflict.

Read-only transactions in PermiSTM modify the read counters of all data 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 avoids $k$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 [23]. 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 Sect. 6), while the implementation of [23] 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 [6, 7], 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 [9], 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 [22]) instead of read locks.

UP-MV STM [27] maintains many versions for each data item to be MV-permissive. It keeps the shortest suffix of versions needed to allow all active read-only transactions to commit, and removes old versions that are no longer required. To support 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 [27], which is not disjoint-access parallel[4] (nor strictly disjoint-access parallel [16]), since UP-MV STM uses a 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

[4]In [3, 27] disjoint-access parallel STMs are referred to as *weakly disjoint-access parallel* as opposed to *strongly disjoint-access parallel* implementations.

atomicity with a global lock or a mechanism similar to TL2 locking [8] could make the algorithm blocking.

**Acknowledgements** We thank the anonymous referees for their comments, which helped to improve the paper.

## References

1. Afek, Y., Merritt, M., Taubenfeld, G., Touitou, D.: Disentangling multi-object operations. In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 111–120 (1997)
2. Attiya, H., Hillel, E.: The cost of privatization. In: Proceedings of the 24th International Conference on Distributed Computing (DISC), pp. 35–49 (2010)
3. Attiya, H., Hillel, E., Milani, A.: Inherent limitations on disjoint-access parallel implementations of transactional memory. Theory Comput. Syst. **49**(4), 698–719 (2011)
4. Aydonat, U., Abdelrahman, T.: Serializability of transactions in software transactional memory. In: 3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT) (2008)
5. Cachopo, J.P., Silva, A.R.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. **63**(2), 172–185 (2006)
6. Carouge, F., Spear, M.: A scalable lock-free universal construction with best effort transactional hardware. In: Proceedings of the 24th International Conference on Distributed Computing (DISC), pp. 50–63 (2010)
7. Dice, D., Lev, Y., Marathe, V., Moir, M., Nussbaum, D., Olszewski, M.: Simplifying concurrent algorithms by exploiting hardware TM. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 325–334 (2010)
8. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proceedings of the 20th International Symposium on Distributed Computing (DISC), pp. 194–208 (2006)
9. Dice, D., Shavit, N.: TLRW: return of the read-write lock. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 284–293 (2010)
10. Diestelhorst, S., Hohmuth, M., Pohlack, M.: Sane semantics of best effort hardware transactional memory. In: 2nd Workshop on the Theory of Transactional Memory (WTTM), September (2010)
11. Ennals, R.: Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report (2006)
12. Fernandes, S.M., Cachopo, J.P.: Lock-free and scalable multi-version software transactional memory. In: Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 179–188 (2011)
13. Gramoli, V., Harmanci, D., Felber, P.: Towards a theory of input acceptance for transactional memories. In: Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS), pp. 527–533 (2008)
14. Guerraoui, R., Henzinger, T.A., Kapalka, M., Singh, V.: Transactions in the jungle. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 263–272 (2010)
15. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: Proceedings of the 22nd International Symposium on Distributed Computing (DISC), pp. 305–319 (2008)
16. Guerraoui, R., Kapałka, M.: On obstruction-free transactions. In: Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 304–313 (2008)
17. Guerraoui, R., Kapałka, M.: The semantics of progress in lock-based transactional memory. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 404–415 (2009)
18. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., III: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 92–101 (2003)
19. Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 151–160 (1994)
20. Kapalka, M.: Theory of transactional memory. Ph.D. thesis, EPFL (2010)

21. Keidar, I., Perelman, D.: On avoiding spare aborts in transactional memory. In: Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 59–68 (2009)
22. Lev, Y., Luchangco, V., Marathe, V.J., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a scalable software transactional memory. In: 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT) (2009)
23. Luchangco, V., Moir, M., Shavit, N.: Nonblocking $k$-compare-single-swap. Theory Comput. Syst. **44**(1), 39–66 (2009)
24. Merritt, R.: IBM plants transactional memory in cpu. EETimes (August 2011). http://www.eetimes.com/electronics-news/4218914/IBM-plants-transactional-memory-in-CPU
25. Napper, J., Alvisi, L.: Lock-free serializable transactions. Technical Report TR-05-04, The University of Texas at Austin (2005)
26. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
27. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in STM. In: Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pp. 16–25 (2010)
28. Perelman, D., Keidar, I.: SMV: selective multi-versioning STM. In: 5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT) (2010)
29. Reinders, J.: Transactional synchronization in haswell (February 2012). http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/
30. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Proceedings of the 20th International Symposium on Distributed Computing (DISC), pp. 284–298 (2006)
31. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 187–197 (2006)