Multi-versioning in Transactional Memory

Idit Keidar[1] and Dmitri Perelman[2]

[1] Technion, Israel Institute of Technology
idish@ee.technion.ac.il
[2] Facebook Inc.
dmitrip@fb.com

Abstract. Reducing the number of aborts is one of the biggest challenges of most transactional systems: existing TMs may abort many transactions that could, in fact, commit without violating correctness. Historically, the commonly used method for reducing the abort rate was maintaining multiple object versions. Multiversion concurrency control is a classical approach for providing concurrent access to the database in database management systems. Its idea is to let a reading transaction obtain a consistent snapshot corresponding to an arbitrary point in time (e.g., defined at the beginning of a transaction) – concurrent updates are isolated through maintaining old versions rather than via scheduling decisions.

Multi-versioning was adopted by transactional memory algorithms as well. In this chapter we overview the multi-versioning approach by studying the inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions. We first consider the challenges of garbage collecting of old object versions, and show that no STM can be optimal in the number of previous versions kept, while following the naïve approach of keeping a constant number of last versions per object might lead to an exponential memory growth. We then show the potential performance challenges of multi-versioned STMs, including disjoint-access parallelism and visibility of read-only transactions.

We demonstrate the advantages of implementing multi-versioned STMs in managed memory environments by presenting Selective Multi-Versioning (SMV) algorithm. SMV relies on automatic garbage collection, and thus efficiently deals with old versions while still allowing invisible read-only transactions.

1 Why Multiple Versions

1.1 Because Read-Only Transactions Matter

Frequent aborts, especially in the presence of long-running transactions, may have a devastating effect on performance and predictability of the execution [3,11,18].

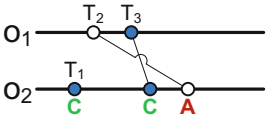
Of particular interest in this context is reducing the abort rate of read-only transactions (transactions with empty write-sets). Read-only transactions play a significant role in various types of applications, including linearizable data structures with a strong prevalence of read-only operations [19], or client-server applications where an STM infrastructure replaces a traditional DBMS approach (e.g., FenixEDU web application [8]). Particularly long read-only transactions are employed for taking consistent snapshots of dynamically updated systems, which are then used for checkpointing, process replication, monitoring program execution, gathering system statistics, etc.

R. Guerraoui and P. Romano (Eds.): Transactional Memory, LNCS 8913, pp. 150–165, 2015.

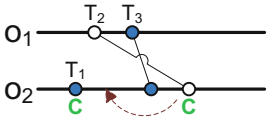
Unfortunately, long read-only transactions might be repeatedly aborted for arbitrarily long periods of time. As we show in [26], the time for completing such a transaction varies significantly under contention, to the point that some read-only transactions simply cannot be executed without "stopping the world". This kind of instability becomes a practical disadvantage for STM adoption in the real-world systems.

Historically, one of the commonly used methods for reducing the number of aborts was maintaining multiple object versions. Multiversion concurrency control is a classical approach for providing concurrent access to the database in database management systems [6,25]. Its idea is to let a reading transaction obtain a *consistent snapshot* [5] corresponding to an arbitrary point in time (typically defined at the beginning of a transaction) – concurrent updates are isolated through maintaining old versions rather than through a process of locks or mutexes.

Multi-versioning technique was adopted by transactional memory algorithms as well [3,24,14,7,26]. By keeping multiple versions it is possible to ensure that every read-only transaction successfully commits. Consider, for example, the scenario depicted in Figure 1. [1] In this run transaction T_2 reads an object o_1, then another transaction T_3 updates objects o_1 and o_2, and commits. Assume that T_2 now tries to read o_2. Reading the value o_2^2 written by T_3 would violate correctness, since T_2 does not read the value o_2^1 written by T_3. In a single-versioned STM, illustrated in Figure 1(a), T_2 must abort. However, a multi-versioned STM may keep both versions o_2^1 and o_2^2 of o_2, and may return o_2^1 to T_2, as illustrated in Figure 1(b). This allows T_2 to successfully commit, in spite of its conflict with T_3.



(a) Single-versioned TM, T_2 aborts.



(b) Multi-versioned TM, T_2 commits.

Fig. 1. Keeping multiple versions avoids aborts, which are inevitable in STMs with only one object version

1.2 Formalizing the Advantages of Multi-versioned Solutions

As mentioned earlier, keeping multiple versions has a potential to significantly improve STM's performance and predictability: we now need rigorous metrics to grasp

[1] We depict transactional histories in the style of [29]. An object o_i's state in time is represented as a horizontal line, with time proceeding left to right. Transactions are drawn as polylines, with circles representing accesses to objects. Filled circles indicate writes, and empty circles indicate reads. A commit is indicated by the letter **C**, and an abort by the letter **A**. A read operation returning an old value of an object is indicated by a dotted arc line. The initial value of object o_i is denoted by o_i^0, and the value written to o_i by the j'th write is denoted by o_i^j.

this intuition. At a high level, we can talk about two aspects of transactional performance: 1) *responsiveness*, for measuring the progress of individual transactional operations, and 2) *permissiveness*, for measuring the wasted operations belonging to aborted transactions.

Responsiveness

We say that a TM is *responsive* if it guarantees that each operation invocation eventually gets a response, even if all other threads do not invoke new transactional operations. This limits the responsive TM's behavior upon operation invocation, so that it may either return an operation response, or abort a transaction, but cannot wait for other transactions to invoke new transactional operations. Note that we do allow for a responsive TM to wait for concurrent transactional operations to complete, for example TL2 [9] is responsive in spite of the use of locks. One may say that a responsive TM provides lock-freedom at the level of transactional operations.

Multi-versioned Permissiveness

We can capture the amount of spuriously aborted transactions using the notion of *permissiveness*, first introduced by Guerraoui et al. [17]. Intuitively, permissiveness defines properties of transactional histories for which no aborts are allowed. Various levels of permissiveness have been defined. *Single-version* π-permissiveness [17] focuses on a model with single-version objects and thus allows many spurious aborts. Another permissiveness condition, *online* π-permissiveness [20], prevents all spurious aborts, which comes with an inherent cost of extremely complex algorithms to implement.

In order to grasp the unique advantages coming with the use of multiple versions in an STM implementation, we use *multi-versioned (MV)* permissiveness: an STM satisfies MV-permissiveness if a transaction aborts only if it is an update transaction that conflicts with another update transaction. In other words, with MV-permissiveness read-only transactions never abort and do not cause aborts of update transactions. We say that an STM satisfying MV-permissiveness is MV-permissive.

Multi-versioning Alternatives: Losing Responsiveness

Besides multi-versioning, there exist multiple approaches for avoiding aborts of read-only transactions, demonstrating the richness of the solution space defined by responsiveness and permissiveness. As a trivial example, we can think of an STM implemented with a single global lock acquired in the beginning of each transaction and released upon commit: while being highly permissive (zero aborted transactions), the global-lock STM is non-responsive (all transactions are mutually exclusive).

There exist various real STMs that avoid aborts of read-only transactions without being multi-versioned:

- Dependence-aware transactional memory [28] reduces the number of aborts by allowing transactions to read uncommitted values and then waiting for the successful commit of the writer.

- TLRW [10] reduces the aborts of read-only transactions by using read-write locks to block in case of concurrency.
- PermiSTM [1] provides MV-permissiveness by having every update transaction being blocked until the termination of all the conflicting readers.

Note that in all the cases mentioned above we lose different degrees of responsiveness (transactions cannot always progress independently) for the sake of reduced overhead and abort rate.

2 Memory Management Challenges of Multi-versioned STMs

One of the key aspects to maintaining multiple versions is a mechanism for garbage collecting (GC) old object versions. In this section we show that while keeping a constant number of versions per object might be suboptimal, a space optimal solution is impossible as well.

2.1 STMs with a Constant Number of Versions for Every Object

The simplest multi-versioning STM approach is to keep a constant preconfigured number of old versions for every object. However, this technique has two main issues.

First, we lose a premise that every read-only transaction successfully commits in a non-blocking manner (responsive MV-permissiveness). Indeed, for every constant number k of object versions, there exists a scenario in which some hot object is updated $k+1$ times after a read of a read-only transaction T_r, such that the old version corresponding to the consistent snapshot of T_r is deleted and the reader has to abort.

Secondly, keeping a constant number of object versions causes an inherent memory consumption problem. A naïve assessment of the memory consumption of a k-versioned STM would probably estimate that it takes up to k times as much more memory as a single-versioned STM.

However, in [26] we demonstrate that, in fact, the memory consumption of a k-versioned STM in runs with n transactional objects might grow like k^n. Intuitively, this happens because previous object versions continue to keep references to already deleted objects, which causes deleted objects to be pinned in memory.

Consider, for example, a 2-versioned STM in the scenario depicted in Figure 2. The STM keeps a linked list of three nodes. When removing node 30 and inserting a new node 40 instead, node 30 is still kept as the previous version of 20.*next*. Next, when node 20 is replaced with node 25, node 30 is still pinned in memory, as it is referenced by node 20. After several additional node replacements, we see that there is a complete binary tree in memory, although only a linked list is used in the application.

More generally, with a k-versioned STM, a linked list of length n could lead to $\Omega(k^n)$ node versions being pinned in memory (though being still linear to the number of write operations). This demonstrates an inherent limitation of keeping a constant number of versions per object. Our observation is confirmed by the empirical results shown in [26], where the algorithms keeping k versions cannot terminate in the runs with a limited heap size.

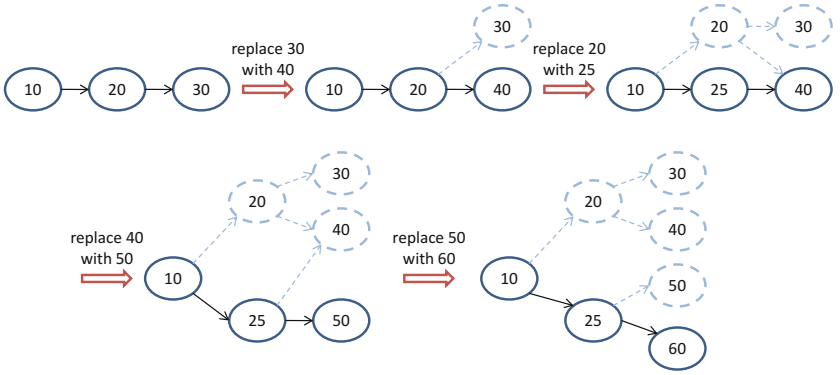


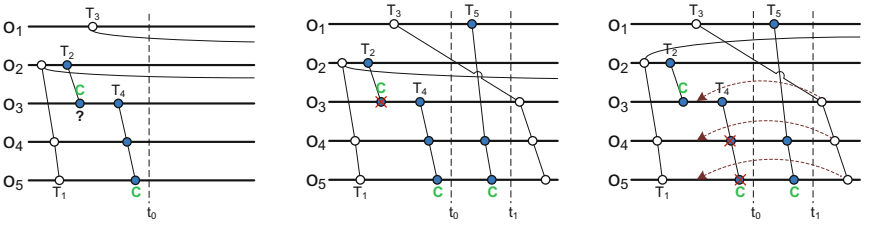
Fig. 2. Example demonstrating exponential memory growth for an STM keeping 2 versions of each object. A linked list implementation creates a whole binary tree to be pinned in memory because previous node versions continue keeping references to already deleted nodes.

2.2 Impossibility of Space Optimal STM

While keeping a constant number of versions does not work, we need a smarter way to manage old object versions. Unfortunately, responsive MV-permissive cannot be space optimal as we show below.

Definition 1. *A responsive MV-permissive STM $\mathscr{X}$ is* online space optimal*, if for any other responsive MV-permissive STM $\mathscr{X}'$ and any transactional history H, the number of versions kept by $\mathscr{X}$ at any point of time during H is less than or equal to the number of versions kept by $\mathscr{X}'$.*

Theorem 1. *No responsive MV-permissive STM can be online space optimal.*



(a) An STM does not know whether to remove o_3^1.

(b) Removing o_3^1 leads to keeping the versions of o_4 and o_5 after they are overwritten.

(c) Keeping o_3^1 allows removing the versions of o_4 and o_5 after they are overwritten.

Fig. 3. No STM can be online space optimal — it is not known at time t_0 whether to remove the version of o_3 written by T_2

Proof (full proof in [27]). The main idea is to construct a transactional history in which any STM that keeps the minimum number of object versions at a time t_0 will keep more than the minimum number of object versions at time $t_1 > t_0$. Consider the transactional history H depicted in Figure 3(a). At time t_0, $\mathscr{X}$ should either remove object version o_3^1 or keep it. In [27] we show that for either one of these decisions, there exists a responsive MV-permissive STM that keeps fewer versions than $\mathscr{X}$ during H or an extension of H. Thus, no STM can keep the minimum number of versions at all times, and so is not online space optimal.

2.3 Garbage Collecting Useless Prefixes

Though we have just seen that no responsive MV-permissive STM is online space optimal, we would still like an STM to manage old versions better than a constant number of object versions approach. Intuitively, we want to garbage collect as many old versions as we can by truncating the whole prefix of a versions list. To this end, we define the following.

Definition 2. *An MV-permissive STM satisfies useless-prefix (UP) GC if at any point in a transactional history H, an object version o_i^j is kept only if there exists an extension of H with an active transaction T_i, such that (1) T_i can read o_i^j, and (2) T_i cannot read any version written after o_i^j.*

In other words an STM satisfying UP GC, removes the longest possible prefix of versions for each object at any point in time and keeps the shortest suffix of versions that might be needed by read-only transactions.

Note that STMs satisfying UP GC are going to keep all the versions of an object that have been added since the snapshot time of the oldest read-only transaction. Therefore, the number of old versions of an object is defined by the ratio of its update rate to the duration time of read-only transactions in the system: rarely updated objects will usually keep the last version only, while hot objects might still keep a lot of previous versions if a long read-only transaction is stuck.

3 Performance Challenges of Multi-versioned STMs

3.1 Disjoint-Access Parallelism

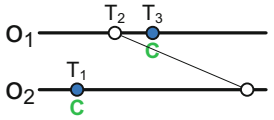
In shared memory systems, cache contention due to concurrent memory accesses, and especially concurrent writes, is a significant performance bottleneck. Thus, it is desirable to try to separate the memory locations accessed by different transactions as much as possible. One natural requirement seems to be that transactions that access different transactional objects access only different base objects. This property is formally captured by the notion of *weak disjoint-access parallelism* [2], which is defined below.

Let T_1, T_2 be transactions, and let α be an execution. Let $\mathscr{T}$ be the set of all transactions whose execution interval overlaps with the execution interval of $\{T_1, T_2\}$ in α. Let X be the set of transactional objects accessed by $\mathscr{T}$. Let $G(T_1, T_2, \alpha)$ be an undirected graph with vertex set X, and an edge between vertices $x_1, x_2 \in X$ whenever there is a

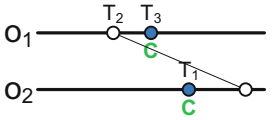
transaction $T \in \mathscr{T}$ accessing both x_1 and x_2. We say T_1, T_2 are *disjoint-access* in α if there is no path between T_1 and T_2 in $G(T_1, T_2, \alpha)$. Given two sets of base steps, we say they *contend* if there is a base object that is accessed by both sets of steps, and at least one of the accesses changes the state of the object.

Definition 3. *An STM is* weakly disjoint-access parallel (weakly DAP) *if, given any execution α, and transactions T_1, T_2 that are disjoint-access in α, the base steps for T_1 and T_2 in α do not contend.*

Theorem 2. *A responsive STM satisfying MV-permissiveness cannot be weakly disjoint-access parallel.*



(a) H_1: $T_1 \preceq T_3$, T_2 must read the value written by T_1.



(b) H_2: $T_3 \preceq T_1$, T_2 cannot read the value written by T_1.

Fig. 4. In a weakly DAP STM T_1 does not distinguish between H_1 and H_2 and cannot be MV-permissive

Proof (full proof in [27]). Suppose for contradiction that there exists a responsive STM satisfying MV-permissiveness that is weakly DAP. Consider the transactional histories in Figure 4. In both H_1 and H_2, transactions T_2 and T_3 conflict on object o_1: T_3 writes to o_1 and commits, overriding the value read by an active transaction T_2. Note that since an STM is responsive and satisfies MV-permissiveness, T_3 neither aborts nor waits for T_2's termination upon a write to o_1. In [27] we prove the following claims: (1) The second step of T_2 returns o_2^1 in H_1. (2) The second step of T_2 returns o_2^1 in H_2. (3) The first step of T_2 returns o_1^0 in H_2. (4) H_2 is not strictly serializable if the first step of T_2 returns o_1^0, and the second step returns o_2^1. Conclusion (4) contradicts the strict serializability of the STM, which proves that there is no responsive STM that is both MV-permissive and weakly DAP.

It is interesting to note that the previous result stems from the real-time order requirement of opacity used as our correctness criterion: independent transactions still need a common base object to designate their real-time order. If we are ready to tolerate real-time order violation of disjoint transactions, we can imagine an implementation of DAP multi-versioned STM.

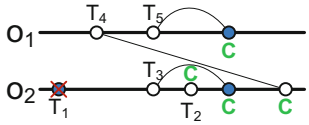
3.2 Read Visibility

Another desirable property for an STM is not to update shared memory during read-only transactions. Such STMs are said to use *invisible reads*. It is easy to show that an

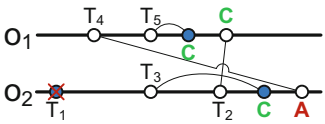
STM satisfying MV-permissiveness and UP GC cannot use invisible reads. Indeed, UP GC requires knowing about existing read-only transactions, in order to determine which object versions to GC; such knowledge cannot be obtained unless read-only transactions write.

However, it is possible to show a much stronger statement: UP GC is impossible even if we allow read-only transactions to write, and only require that the external configurations before and after the transaction are the same. In other words, UP GC requires read-only transactions to leave some trace of their existence, even *after* they have committed. In particular, even keeping active readers lists for the objects [15], or using non-zero indicators for conflict detection [12] does not suffice.

Theorem 3. *Suppose a responsive STM satisfies MV permissiveness and UP GC. Consider a read-only transaction whose execution interval does not contain base steps of any other transaction. Then the configuration external to the transaction, immediately before and after the transaction, cannot be the same.*



(a) H_1: o_2^1 is GCed, T_4 can read o_2^2 and commits.



(b) H_2: o_2^1 is GCed, T_4 cannot read o_2^2 and aborts.

Fig. 5. H_1 and H_2 are indistinguishable if a read-only transaction T_2 does not leave any trace after its execution

Proof (full proof in [27]). Suppose for contradiction that there exists a responsive STM satisfying MV-permissiveness and UP GC, in which the external configurations before and after a read-only transaction are the same, when the transaction's interval does not overlap the steps of any other transaction. Consider the transactional histories in Figure 5. In [27] we prove the the following claims: (1) o_2^1 is GCed in H_1. (2) o_2^1 is GCed in H_2. (3) T_4 aborts in H_2. Conclusion (3) is a contradiction, because T_4 is a read-only transaction, and cannot abort because of MV-permissiveness.

4 Multi-versioned STM in Managed Memory Environments

4.1 Concurrent Algorithms Are Simpler with Garbage Collection

As demonstrated in Sections 2 and 3, maintaining multiple versions in an STM is a challenging task. Space optimality is impossible and even with a non-optimal useless-prefix garbage collection, read-only transactions must leave a trace of their existence, which might devastate STM performance.

Combining invisible readers with effective garbage collection is problematic — if read-only transactions are invisible, then other transactions have no way of telling

whether potential readers of an old version still exist! Some STM implementations take the approach of special cleanup threads, like JVSTM [7]: in this case the visibility of the readers' operations can be limited to cleanup threads only. However, in garbage collected environments it is possible to exploit the designated GC threads, which are running in the system anyway. GC threads have access to all the threads' private memories, so that even operations that are invisible to other transactions are visible to the garbage collector.

We now give a brief reminder of the garbage collection mechanism. An object can be reclaimed by the garbage collector once it becomes unreachable from the call stack or global variables. Reachability is a transitive closure over *strong* memory references: if a reachable object o_1 has a strong reference to o_2, then o_2 is reachable as well (strong references are the default ones). In contrast, *weak references* [16] do not protect the referenced object from being GCed; an object referenced by weak references only is considered unreachable and may be removed.

Generally speaking, an automatic deletion of unreachable objects in garbage collected environments plays a significant role in various concurrent systems way beyond the STM world, dramatically simplifying the algorithmic part in comparison with native environments. One nice side effect of an automated GC is the elimination of the ABA problem that might occur in dynamic data structures [22]: object memory cannot be reallocated to another object as long as this memory is reachable by a live thread. This property was used in the adaptation of Michael-Scott non-blocking concurrent queue [23] to Java concurrency library, as well as in CAFÉ, scalable producer consumer Java library [4].

4.2 Selective Multi-Versioning (SMV) STM

We now want to exemplify the principles discussed earlier in this section, in which garbage collection of old versions is delegated to the already existing GC mechanisms of the managed environment. For that purpose we present *Selective Multi-Versioning (SMV)* [26], an STM which keeps old object versions that are still useful to potential readers, while allowing read-only transactions to remain invisible by ensuring that old object versions become *garbage collectible* once there are no transactions that can safely read them.

SMV is especially efficient for read-dominated workloads with long read-only transactions, in situations where other transactions would either repeatedly abort readers or block update transactions for extended periods of time.

4.2.1 Overview of Data Structures

SMV's main goal is to reduce aborts in workloads with read-only transactions, without introducing high space or computational overheads. SMV is based on the following design choices: 1) Read-only transactions do not affect the memory that can be accessed by other transactions. This property is important for performance in multi-core systems, as it avoids cache thrashing issues [13,30]. 2) Read-only transactions always commit. A read-only transaction T_i observes a consistent snapshot corresponding to T_i's start time — when T_i reads object o_j, it finds the latest version of o_j that has been written before T_i's start. 3) Old object versions are removed once there are no live read-only

transactions that can consistently read them. To achieve this with invisible reads, SMV relies on the omniscient GC mechanism available in managed memory systems.

As in other object-based STMs, transactional objects in SMV are accessed via *object handles*. An object handle includes a history of object values, where each value keeps a *versioned lock* [9] – a data structure with a version number and a lock bit. In order to facilitate automatic garbage collection, object handles in SMV keep strong references only to the latest (current) versions of each object, and use weak references to point to other versions.

Each transaction is associated with a *transactional descriptor*, which holds the relevant transactional data, including a read-set, a write-set, status, etc. In addition, transactional descriptors play an important role in keeping strong references to old object versions, as we explain below.

Version numbers are generated using a global version clock, where transactional descriptors act as "time points" organized in a one-directional linked list. Upon commit, an update transaction appends its transactional descriptor to the end of the list (a special global variable *curPoint* points to the latest descriptor in this list). For example, if the current global version is 100, a committing update transaction sets the time point value in its transactional descriptor to 101 and adds a pointer to this descriptor from the descriptor holding 100.

Version management is based on the idea that old object versions are pointed to by the descriptors of transactions that over-wrote these versions (see Figure 6). A committing transaction T_w includes in its transactional descriptor a strong reference to the previous version of every object in its write set before diverting the respective object handle to the new version.

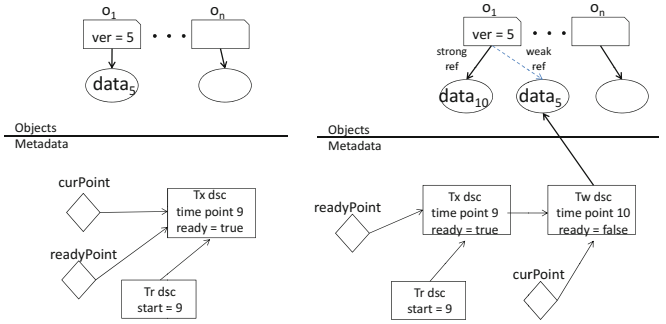
When a read-only transaction T_i begins, it keeps (in its local variable *startTP*) a pointer to the latest transactional descriptor in the list of committed transactions. This pointer is cleared upon commit, making old transactional descriptors at the head of the list GCable.

This way, active read-only transaction T_r keeps a reference chain to version o_i^j if this version was over-written after T_r's start, thus preventing o_i^j's garbage collection. Once there are no active read-only transactions that started before o_i^j was over-written, this version stops being referenced and thus becomes GCable .

Figure 6 illustrates the commit of an update transaction T_w that writes to object o_1 (the use of *readyPoint* variable will be explained in Section 4.2.3). In this example, T_w and a read-only transaction T_r both start at time 9, and hence T_r references the transactional descriptor of time point 9. The previous update of o_1 was associated with version 5. When T_w commits, it inserts its transactional descriptor at the end of the time points list with value 10. T_w's descriptor references the previous value of o_1. This way, the algorithm creates a reference chain from T_r to the previous version of o_1 via T_w's descriptor, which ensures that the needed version will not be GCed as long as T_r is active.

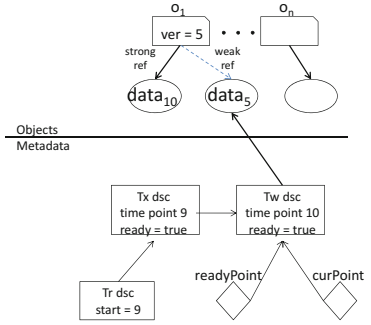
4.2.2 Basic Algorithm

We now describe the SMV algorithm. For the sake of simplicity, we present the algorithm in this section using a global lock for treating concurrency on commit — in Section 4.2.3 we show how to remove this lock.



(a) T_r's descriptor points to the latest committed transaction.

(b) T_w commits and begins write-back.



(c) T_w's write-back is finished.

Fig. 6. Transactional descriptor of T_W references the over-written version of o_1 (data$_5$). This way, read-only transaction T_r keeps a reference chain to the versions that have been overwritten after T_r's start.

SMV handles read-only and update transactions differently. We assume that transaction's type can be provided to the algorithm beforehand by a compiler or via special program annotations. If not, each transaction can be started as read-only and then restarted as update upon the first occurrence of a write operation.

Handling Update Transactions

The protocol for update transaction T_i is depicted in Algorithm 1. The general idea is similar to the one used in TL2 [9]. An update transaction T_i aborts if some object o_j read by T_i is over-written after T_i begins and before T_i commits. Upon starting, T_i saves the value of the latest time point in a local variable *startTime*, which holds the latest time at which an object in T_i's read-set is allowed to be over-written.

A read operation of object o_j reads the latest value of o_j, and then post-validates its version (function *validateRead*. The validation procedure checks that the version is not locked and it is not greater than T_i.startTime, otherwise the transaction is aborted.

A write operation (lines 12–14) creates a copy of the object's latest version and adds it to T_i's local write set.

Algorithm 1. SMV algorithm for update transaction T_i.

```
1:  Upon Startup:
2:      T_i.startTime ← curPoint.commitTime
3:  Read o_j:
4:      if (o_j ∈ T_i.writeSet)
5:          then return T_i.writeSet[o_j]
6:      data ← o_j.latest
7:      if ¬validateRead(o_j) then abort
8:      readSet.put(o_j)
9:      return data
10: Write to o_j:
11:     if (o_j ∈ T_i.writeSet)
12:         then update T_i.writeSet.get(o_j); return
13:     localCopy ← o_j.latest.clone()
14:     update localCopy; writeSet[o_j] ← localCopy
15: Function validateReadSet
16:     foreach o_j ∈ T_i.readSet do:
17:         if ¬validateRead(o_j) then return false
18:     return true
19: Commit:
20:     foreach o_j ∈ T_i.writeSet do: o_j.lock()
21:     if ¬validateReadSet() then abort
        ▷ txn dsc should reference the over-written data
22:     foreach o_j ∈ T_i.writeSet do:
23:         T_i.prevVersions.put(⟨o_j, o_j.latest⟩)
24:     timeLock.lock()
25:     T_i.commitTime ← curPoint.commitTime + 1
        ▷ update and unlock the objects
26:     foreach ⟨o_j, data⟩ ∈ T_i.writeSet do:
27:         o_j.version ← T_i.commitTime
28:         o_j.weak_references.append(o_j.latest)
29:         o_j.latest ← data; o_j.unlock()
30:     curPoint.next ← T_i; curPoint ← T_i
31:     timeLock.unlock()
32: Function validateRead(Object o_j)
33:     return (¬o_j.isLocked ∧ o_j.version ≤ T_i.startTime)
```

Commit (lines 20–31) consists of the following steps:

1. Lock the objects in the write set (line 20). Deadlocks can be detected using standard mechanisms (e.g., timeouts or Dreadlocks [21]), or may be avoided if acquired in the same order by every transaction.
2. Validate the read set (function *validateReadSet*).

3. Insert strong references to the over-written versions to T_i's descriptor (line 23). This way the algorithm guarantees that the over-written versions stay in the memory as long as T_i's descriptor is referenced by some read-only transaction.
4. Lock the time points list (line 24). Recall that this is a simplification; in Section 4.2.3 we show how to avoid such locking.
5. Set the commit time of T_i to one plus the value of the commit time of the descriptor referenced by *curPoint*.
6. Update and unlock the objects in the write set (lines 26–29). Set their new version numbers to the value of T_i.commitTime. Keep weak references to old versions.
7. Insert T_i's descriptor to the end of the time points list and unlock the list (line 30).

Handling Read-Only Transactions

Algorithm 2. SMV algorithm for read-only transaction T_i.

```
1: Upon Startup:
2:     T_i.startTP ← curPoint

3: Read o_j:
4:     latestData ← o_j.latest
5:     if (o_j.version ≤ T_i.startTP.commitTime) then return latestData
6:     return the latest version ver in o_j.weak_references, s.t.
7:         ver.version ≤ T_i.startTP.commitTime

8: Commit:
9:     T_i.startTP ← ⊥
```

The pseudo-code for read-only transactions appears in Algorithm 2. Such transactions always commit without waiting for other transactions to invoke any operations. The general idea is to construct a consistent snapshot based on the start time of T_i. At startup, T_i.*startTP* points to the latest installed transactional descriptor (line 2); we refer to the time value of startTP as *T_i's start time*.

For each object o_j, T_i reads the latest version of o_j written before T_i's start time. When T_i reads an object o_j whose latest version is greater than its start time, it continues to read older versions until it finds one with a version number older than its start time. Some old enough version is guaranteed to be found, because the updating transaction T_w that over-wrote o_j has added T_w's descriptor referencing the over-written version somewhere after T_i's starting point, preventing GC.

The commit procedure for read-only transactions merely removes the pointer to the starting time point, in order to make it GCable, and always commits.

4.2.3 Allowing Concurrent Access to the Time Points List

We show now how to avoid locking the time points list (lines 24, 31 in Algorithm 1), so that update transactions with disjoint write-sets may commit concurrently.

We first explain the reason for using the lock. In order to update the objects in the write-set, the updating transaction has to know the new version number to use. However, if a transaction exposes its descriptor before it finishes updating the write-set, then

some read-only transaction might observe an inconsistent state. Consider, for example, transaction T_w that updates objects o_1 and o_2. The value of *curPoint* at the beginning of T_w's commit is 9. Assume T_w first inserts its descriptor with value 10 to the list, then updates object o_1 and pauses. At this point, $o_1.version = 10$, $o_2.version < 10$ and $curPoint \rightarrow commitTime = 10$. If a new read-only transaction starts with time 10, it can successfully read the new value of o_1 and the old value of o_2, because they are both less than or equal to 10. Intuitively, the problem is that the new time point becomes available to the readers as a potential starting time before all the objects of the committing transaction are updated.

To preserve consistency without locking the time points list, we add an additional boolean field *ready* to the descriptor's structure, which becomes *true* only after the committing transaction finishes updating all objects in its write-set. In addition to the global *curPoint* variable referencing the latest time point, we keep a global *readyPoint* variable, which references the latest time point in the *ready prefix* of the list (see Figure 6).

When a new read-only transaction starts, its *startTP* variable references *readyPoint*. In the example above, a new transaction T_r begins with a start time equal to 9, because the new time point with value 10 is still not ready. Generally, the use of *readyPoint* guarantees that if a transaction reads an object version written by T_w, then T_w and all its preceding transactions had finished writing their write-sets.

Note, however, that when using ready points we should not violate the real time order — if a read-only transaction T_r starts after T_w terminates, then T_r must have a start time value not less than T_w's commit time. This property might be violated if update transactions become ready in an order that differs from their time points order, thus leaving an unready transaction between ready ones in the list.

In [26] we have implemented two approaches to enforce real-time order: 1) An update transaction does not terminate until the ready point reaches its descriptor. A similar approach was previously used by RingSTM [31] and JVSTM [14]. 2) A new read-only transaction notes the time point of the latest terminated transaction and then waits until the *readyPoint* reaches this point before starting. Note that unlike the first alternative, read-only transactions in the second approach are not wait-free.

According to [26], both techniques demonstrate similar results. The waiting period remains negligible as long as the number of transactional threads does not exceed the number of available cores; when the number of threads is two times the number of cores, waiting causes a $10-15\%$ throughput degradation (depending on the workload) — this is the cost we pay for maintaining real-time order.

5 Conclusions

An effective way to reduce the number of aborts in transactional memory is keeping multiple versions of transactional objects. We studied the inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions (we call such STMs MV-permissive). We presented the challenge of efficient garbage collection of old object versions by demonstrating that the memory consumption of algorithms keeping a constant number of versions for each object can grow exponentially. We then showed that no responsive MV-permissive STM can be optimal in the

number of previous versions kept and that no responsive MV-permissive STM can be disjoint-access parallel. We defined an achievable garbage collection property, useless-prefix GC, and showed that in a responsive MV-permissive STM satisfying UP GC, even read-only transactions must make lasting changes to the system state.

Theoretical study of multi-versioning in STM is far from being complete. There are clear tradeoffs between the quality of garbage collection, permissiveness and the computational complexity of transactional operations: we believe that understanding these tradeoffs may be valuable to improving the performance and utility of transactional memory.

We referred to practical implications of multi-versioning by discussing SMV, a multi-versioned STM that achieves high performance in the presence of read-only transactions. Despite keeping multiple versions, SMV can work well in memory constrained environments. It keeps old object versions as long as they might be useful while still allowing read-only transactions to remain invisible by relying on automatic garbage collection to dispose of obsolete versions.

SMV exemplifies the idea of delegating disposal responsibilities to the independent GC module that is being developed and upgraded by a very large community. We think that this approach can be the key to achieving good performance not only in STMs, but also in a range of concurrent data structures.

References

1. Attiya, H., Hillel, E.: Single-version STMs can be multi-version permissive (Extended abstract). In: Aguilera, M.K., Yu, H., Vaidya, N.H., Srinivasan, V., Choudhury, R.R. (eds.) ICDCN 2011. LNCS, vol. 6522, pp. 83–94. Springer, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=1946143.1946151
2. Attiya, H., Hillel, E., Milani, A.: Inherent limitations on disjoint-access parallel implementations of transactional memory. In: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2009, pp. 69–78. ACM, New York (2009), http://doi.acm.org/10.1145/1583991.1584015
3. Aydonat, U., Abdelrahman, T.: Serializability of transactions in software transactional memory. In: Second ACM SIGPLAN Workshop on Transactional Computing (2008)
4. Basin, D., Fan, R., Keidar, I., Kiselov, O., Perelman, D.: CAFÉ: Scalable task pools with adjustable fairness and contention. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 475–488. Springer, Heidelberg (2011)
5. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 1–10 (1995)
6. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
7. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming 63(2), 172–185 (2006)
8. Carvalho, N., Cachopo, J., Rodrigues, L., Rito-Silva, A.: Versioned transactional shared memory for the FenixEDU web application. In: Proceedings of the 2nd Workshop on Dependable Distributed Data Management, pp. 15–18 (2008)
9. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)

10. Dice, D., Shavit, N.: TLRW: Return of the read-write lock. In: TRANSACT 2009: 4th Workshop on Transactional Computing (February 2009)
11. Dragojević, A., Harris, T.: Stm in the small: Trading generality for performance in software transactional memory. In: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys 2012, pp. 1–14. ACM, New York (2012), http://doi.acm.org/10.1145/2168836.2168838
12. Ellen, F., Lev, Y., Luchangco, V., Moir, M.: Snzi: Scalable nonzero indicators. In: PODC 2007: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 13–22. ACM, New York (2007)
13. Ennals, R.: Cache sensitive software transactional memory. Tech. rep.
14. Fernandes, S.M., Cachopo, J.A.: Lock-free and Scalable Multi-Version Software Transactional Memory. In: PPoPP 2011, pp. 179–188 (2011)
15. Fraser, K.: Practical lock freedom. Ph.D. thesis, Cambridge University Computer Laboratory (2003)
16. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley Longman (2005)
17. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in Transactional Memories. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 305–319. Springer, Heidelberg (2008)
18. Heber, T., Hendler, D., Suissa, A.: On the impact of serializing contention management on stm performance. J. Parallel Distrib. Comput. 72(6), 739–750 (2012), http://dx.doi.org/10.1016/j.jpdc.2012.02.009
19. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
20. Keidar, I., Perelman, D.: On avoiding spare aborts in transactional memory. In: SPAA 2009, pp. 59–68 (2009)
21. Koskinen, E., Herlihy, M.: Dreadlocks: Efficient deadlock detection. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 297–303 (2008)
22. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. 15, 491–504 (2004)
23. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1996, pp. 267–275 (1996)
24. Napper, J., Alvisi, L.: Lock-free serializable transactions. Tech. rep., The University of Texas at Austin (2005)
25. Papadimitriou, C.H., Kanellakis, P.C.: On concurrency control by multiple versions. ACM Trans. Database Syst., 89–99 (1984)
26. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: Selective multi-versioning STM. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 125–140. Springer, Heidelberg (2011)
27. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in STM. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, PODC 2001, pp. 16–25 (2010)
28. Ramadan, H.E., Roy, I., Herlihy, M., Witchel, E.: Committing conflicting transactions in an STM. SIGPLAN Not 44(4), 163–172 (2009)
29. Riegel, T., Fetzer, C., Sturzrehm, H., Felber, P.: From causal to z-linearizable transactional memory. In: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing, pp. 340–341 (2007)
30. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Proceedings of the 20th International Symposium on Distributed Computing, pp. 284–298 (2006)
31. Spear, M.F., Michael, M.M., von Praun, C.: RingSTM: Scalable transactions with a single atomic instruction. In: SPAA 2008, pp. 275–284 (2008)