IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

On Maintaining Multiple Versions in STM

Dmitri Perelman, Rui Fan and Idit Keidar

CCIT Report #765
May 2010

Electronics
Computers
Communications

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT REPORT #765 May 2010

On Maintaining Multiple Versions in STM

Dmitri Perelman[*] Rui Fan[*] Idit Keidar[*]

May 16, 2010

Abstract

An effective way to reduce the number of aborts in software transactional memory (STM) is to keep multiple versions of transactional objects. In this paper, we study inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions.

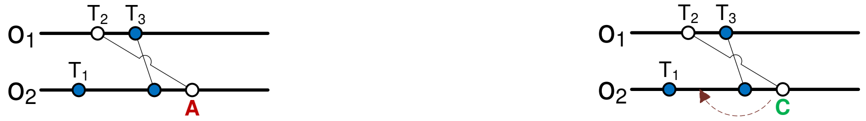
We first show that these STMs cannot be disjoint-access parallel. We then consider the problem of garbage collecting old object versions, and show that no STM can be optimal in the number of previous versions kept. Moreover, we show that garbage collecting useless versions is impossible in STMs that implement invisible reads. Finally, we present an STM algorithm using visible reads that efficiently garbage collects useless object versions.

1 Introduction

Transactional memory [12, 18] is a popular paradigm for concurrent computing in modern multi-core architectures. Most current transactional memory implementations are software toolkits, or *STM*s for short. STMs speculatively allow multiple transactions to proceed concurrently, before knowing all possible data dependencies between them. This optimistic approach inevitably leads to aborting transactions in some cases, such as when data dependencies introduce inconsistencies. When many transactions contend on the same data objects, aborts may become frequent, causing a devastating effect on performance [2, 15]. Therefore, reducing the number of aborts is an important challenge for STMs.

While some aborts are unavoidable, existing STMs tend to be over-conservative, and also abort transactions that could have been committed without violating consistency. Such unnecessary aborts often stem from coarse-grained inconsistency detection. Consider the scenario depicted in Figure 1. We depict transactional histories in the style of [17]. 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. In the scenario depicted in Figure 1 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.

[*] Department of Electrical Engineering, Technion, Haifa, Israel {dima39@tx,rfan@ee,idish@ee}.technion.ac.il



(a) Single-versioned TM, T_2 aborts.

(b) Multi-versioned TM, T_2 commits.

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

We call aborts that can be avoided, such as T_2's abort in Figure 1(a), *spurious*. We can capture the amount of spurious aborts that we allow using the notion of *permissiveness*. Some previously defined permissiveness conditions, such as *single-version* permissiveness [8], are too weak, and still allow many spurious aborts. Other permissiveness conditions, such as *online* π-permissiveness [13], prevent all spurious aborts, but require complex algorithms to implement (see Section 2 for details). In Section 4, we define the new notion of *multi-versioned (MV)* permissiveness. It ensures that read-only transactions never abort, and permits update transactions to abort only when they conflict with other update transactions. This property can be achieved by practical algorithms. In fact, the algorithms in [16, 3, 2] would all satisfy MV-permissiveness if they kept enough object versions.

A key challenge when maintaining multiple versions is knowing when to *garbage collect* (GC) old object versions. On the one hand, an STM needs to keep versions that might be needed in the future. On the other hand, keeping unneeded versions wastes memory. In Section 5, we show that this problem is inherent. We prove that no STM algorithm can be *space optimal*, i.e., ensure that it always maintains the minimum number of object versions possible. We then define an achievable GC property called *useless prefix (UP)* GC, based on maintaining object versions only when they may be needed by some existing read-only transactions.

Satisfying MV-permissiveness (and UP GC) imposes costs on an STM. A key contribution of our paper is a systematic study of such necessary and sufficient costs. In Section 6, we show that an MV-permissive STM cannot be *weakly disjoint-access parallel (DAP)*. Roughly speaking, this means that in order to ensure that read-only transactions never abort, it is necessary for transactions to communicate with each other, even when they do not access the same transactional objects. We also show that if an STM is MV-permissive and satisfies UP GC, then read-only transactions must leave some trace of themselves in shared memory, even after they have committed. Note that this implies the STM cannot use *invisible reads* [6], an important technique for optimizing read-only transactions. We also note that if the UP GC requirement is omitted, then it is possible to implement an STM using invisible reads, as done in our companion paper [15], assuming there exists a garbage collection thread that sees the private ("invisible") memory of all transactions, such as the Java GC.

Finally, to complete our exploration of the design space of MV-permissiveness and garbage collection, we present in Section 7 a non-DAP algorithm using visible reads, satisfying MV-permissiveness and UP GC. Our results are summarized in Table 1.

2 Related Work

Permissiveness. The notion of permissiveness was first introduced by Guerraoui *et al.* [8]. Informally, an STM satisfies π-permissiveness for a correctness criterion π, if every history that does not violate π is accepted by the STM. However, Guerraoui *et al.* focused on a model with single-versioned objects, which

	MV-Permissiveness (Sec. 4)
Space Optimality	Impossible (Sec. 5.1)
DAP	Impossible (Sec. 6.1)
UP GC (Sec. 5.2)	Impossible when read-only transactions leave no trace after commit. (Sec. 6.2) Possible with non-DAP algorithm using visible reads. (Sec. 7)

Table 1: Summary of our results.

is insufficient for avoiding many spurious aborts.

Another permissiveness condition, online π-permissiveness, was presented in our earlier paper [13]. Online permissiveness does not allow aborting transactions if there is a way to continue the run without violating π [10]. This condition is strong enough to avoid all spurious aborts, but is too complex to achieve with practical algorithms, and also requires keeping a large number of object versions. In fact, object versions overwritten by a write-only transaction T cannot be garbage collected until all transactions that started before T's commit terminate.

Garbage collection. Any practical multi-versioned STM has to address the problem of removing old object versions. Some earlier STMs, such as LSA [16] and Versioned Boxes [3], keep a fixed number of old object versions. This approach is neither necessary nor sufficient: certain object versions kept by these algorithms may be GCed without causing additional aborts, while the algorithms sometimes do not keep enough object versions to ensure all read-only transactions commit.

Another approach for garbage collection was presented in our *selective multi-versioning (SMV)* STM [15]. SMV keeps a variable number of old versions, which reduces memory usage while ensuring read-only transactions can always commit. Nevertheless, SMV does not satisfy UP GC, and hence keeps more object versions than the algorithm we present in Section 7. In addition, our new algorithm is more efficient for read-only transactions. The tradeoff is that update transactions are more costly.

Impossibility of DAP. An important technique for optimizing STM performance is disjoint-access parallelism. As described earlier, this means that transactions that do not access the same objects should also not access the same memory locations, thereby avoiding memory contention. Kapalka and Guerraoui [9] show that a single-versioned, obstruction free [11] STM cannot be strictly DAP. However, their proof does not apply in the multi-versioned setting we consider.

Attiya *et al.* [1] show that there is no STM implementing DAP that uses invisible reads, in which read-only transactions always terminate. In Section 6.1, we show that no MV-permissive STM can be DAP. As stated earlier, MV-permissiveness ensures all read-only transactions commit, and update transactions abort only when they conflict with other update transactions. Thus, our results show that the requirement of invisible reads in [1] can be replaced by precluding update transactions from aborting when they conflict with read-only transactions.

3 System Model

Transactions. A *transaction* consists of a sequence of *transactional operations*, where each operation is comprised of an invocation step and a subsequent matching response step, collectively called *transactional steps*. The system contains a set of *transactional objects*. Each transactional operations either accesses a transactional object, or tries to commit or abort the transaction. More precisely, let T be a transaction, o be a transactional object, and v be a value. Then a transactional operation is one of the following. (1) An invocation step $read(T, o)$, followed by a response step that either gives the current value of o, or responds $A(T)$, meaning that the transaction is *aborted*. (2) An invocation $write(T, o, v)$, followed by a response either acknowledging the write, or responding $A(T)$. (3) An invocation *Abort*(T), followed by response $A(T)$. (4) An invocation $Commit(T)$, followed either by response $C(T)$, meaning T committed, or $A(T)$.

We say the *read set*, resp. *write set* of a transaction is the set of transactional objects read, resp. written to by T. We say T is *read-only* if its write set is empty. An update transaction is any transaction that is not read-only. We say two transactions *conflict* if they both access a common transactional object, and at least one of the accesses is a write. We assume that the steps in a transaction are not known ahead of time, but it is known a priori whether a transaction is a read-only or update transaction. Detection of read-only behavior can be done at compile time or using programmer annotations.

A *transactional history* H is a sequence of transactional steps, interleaved in an arbitrary order. A transaction is *live* in H if it is neither committed nor aborted, it is *complete* otherwise. We let $complete(H)$ denote the set of completed transactions in H.

Serializations. Two transactional histories H and H' are *equivalent* if they contain the same transactions, and every transaction performs the same sequence of invocations and receives the same responses in both histories. The *real-time order* of a transactional history H, written $\preceq_H$, is a partial order on the transactions in H. Given transactions T, T' in H, we define $T \preceq_H T'$ when the response step for $Commit(T)$ occurs before the first step of T'. T, T' are *concurrent* if neither $T \preceq_H T'$, nor $T' \preceq_H T$.

A transactional history S is *sequential* if it has no concurrent transactions. S is *legal* if it respects the sequential specification of each transactional object accessed in S. H is *strictly serializable* [14] if $complete(H)$ is equivalent to some legal sequential history S, and $\preceq_S$ is a refinement of $\preceq_H$. Note that strict serializability is strictly weaker than the commonly used correctness condition of *opacity* [10]. Our lower bounds hold for algorithms satisfying strict serializability. It can be shown that our algorithm satisfies opacity, though for simplicity, we only consider strict serializability herein.

STM. A *software transactional memory (STM)* is an algorithm for running transactions. In this paper, we assume the algorithm consists of a set of *threads*. The threads communicate with each other using *shared memory*, and each thread also has *private memory* which it alone can access. Each transaction is *run* by a thread, and each thread runs at most one transaction at a time. To run a transaction T, a thread runs each of T's transactional operations, as follows. (1) Take as input an invocation step of T. (2) Perform a sequence of private and shared memory steps, which are determined by the input and the memory. (3) Return as output a response step to T. We write $thr(T)$ for the thread running T.

We call the memory objects accessed by the threads *base objects*. Note that these are conceptually distinct from the transactional objects accessed by the transactions. We also call the steps performed by the threads *base steps*. We assume that all the base steps for running a transactional step appear to execute atomically. In practice, this atomicity can be achieved using locks, or by lock-free algorithms [7].

The STM guarantees that each operation invocation eventually gets a response, even if all other threads do not invoke new transactional operations. This limits the STM'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 our model does allow waiting for concurrent transactional operations to complete, such as the use of locks in TL2 [4]. In other words, the STM provides lock-freedom at the level of transactional operations.

A *configuration* of an STM consists of the states of the shared memory, private memory, and threads. An *execution* of an STM is an alternating sequence of configurations and base steps, starting with a configuration in which the memory and threads are all in their initial states. Two executions are *indistinguishable* to a thread if it performs the same sequence of state changes in both executions. Given a configuration C and a transaction T, we let the *configuration external to* T in C consist of the state of the shared memory and the states and private memories of all threads other than $thr(T)$ in C.

Given a set of transactions $\mathcal{T}$ and an execution α, the *execution interval* of $\mathcal{T}$ in α, written $interval(\alpha, \mathcal{T})$, is the smallest subsequence of α containing all the base steps for the transactions in $\mathcal{T}$.

DAP. We define the notion of *weak disjoint-access parallelism*, following [1]. Let T_1, T_2 be transactions, and let α be an execution. Let $\mathcal{T}$ 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 $\mathcal{T}$. 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 $T \in \mathcal{T}$ 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 1. *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.*

4 Multi-Versioned Permissiveness

One of the main benefits of multi-versioning is reducing the aborts rate. In order to evaluate the effectiveness of multi-versioned STMs, we need to formally define the set of aborts that are avoided. Such restrictions on aborts are captured by *permissiveness* conditions. As noted in Section 2, many existing permissiveness notions are either too weak or too strong. In this section, we define a practically achievable permissiveness property that is suited for multi-versioned STMs.

Multi-versioning is particularly useful for avoiding aborts of read-only transactions. In fact, by keeping enough versions, read-only transaction can always find appropriate object versions to read, and commit successfully. Our permissiveness condition captures this property. In addition, it captures the property that read-only transactions do not cause update transactions to abort.

Definition 2. *An STM satisfies multi-versioned (MV)-permissiveness if a transaction aborts only when it is an update transaction that conflicts with another update transaction.*

We say that an STM satisfying MV-permissiveness is *MV-permissive*.

Most multi-versioned algorithms [16, 3, 2] are not MV-permissive, because they do not always keep all the object versions needed to commit all read-only transactions. However, the algorithm we present in Section 7, as well as the algorithm in [15], are MV-permissive.

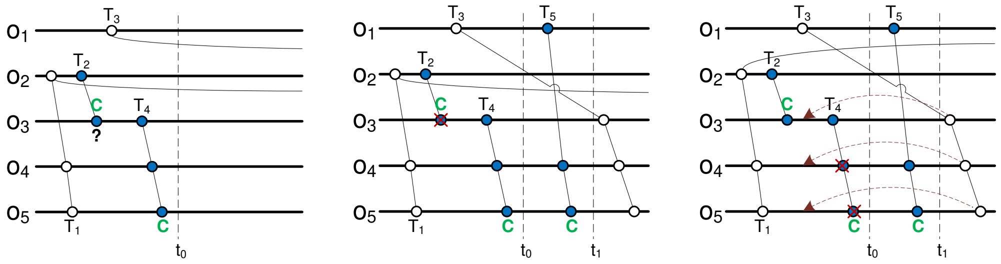
5 Garbage Collection Properties

A key aspect to maintaining multiple versions is a mechanism for garbage collecting (GC) old object versions. This section considers two sides to this problem. In Section 5.1 we show that no STM can always keep the minimum number of old object versions. Then in Section 5.2, we define an achievable GC property that removes many old versions.

5.1 Impossibility of Space Optimal STM

Definition 3. *An MV-permissive STM $\mathcal{X}$ is* online space optimal*, if for any other MV-permissive STM $\mathcal{X}'$ and any transactional history H, the number of versions kept by $\mathcal{X}$ at any point of time during H is less than or equal to the number of versions kept by $\mathcal{X}'$.*

Theorem 1. *No 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 obliges an STM to keep the versions written by T_4 to o_4 and o_5 even after they are overwritten.

(c) Keeping o_3^1 allows removing the versions written by T_4 to o_4 and o_5 after they are overwritten.

Figure 2: 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. 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 $t_1 > t_0$. Thus, no STM can keep the minimum number of versions at all times, and so is not online space optimal.

Formally, assume for contradiction that there exists an online space optimal STM $\mathcal{X}$ satisfying MV-permissiveness. Consider the transactional history H depicted in Figure 2(a). At time t_0, $\mathcal{X}$ should either remove object version o_3^1 or keep it. We show that for either one of these decisions, there exists an MV-permissive STM that keeps fewer versions than $\mathcal{X}$ during H or an extension of H.

Assume first that $\mathcal{X}$ keeps o_3^1 at time t_0. Consider another STM $\mathcal{X}'$ which behaves the same as $\mathcal{X}$ until time t_0, but GCs o_3^1 as soon as T_4 performs its write to o_3. Then $\mathcal{X}'$ keeps fewer object versions than $\mathcal{X}$. It remains to show that $\mathcal{X}'$ does not violate MV-permissiveness by GCing o_3^1. Notice that it suffices to show that at time t_0, all live read-only transactions, namely T_1 and T_3, can commit. Now, T_1's first read step precedes T_2's first write step. Thus, T_1 cannot read o_3^1 when invoking a read operation of o_3. $\mathcal{X}$ is MV-permissive, hence there exists a version $o_3^x \neq o_3^1$, which is kept by $\mathcal{X}$ at time t_0 and which can be read by T_1. Other than removing version o_3^1, $\mathcal{X}$ and $\mathcal{X}'$ are the same — T_1 can read o_1^x when invoking a read operation of o_3. Also, T_3 can return o_2^3, by serializing T_3 after T_4. So both T_1 and T_3 can commit after $\mathcal{X}'$ removes o_1^3, and so $\mathcal{X}'$ satisfies MV-permissiveness. Thus, $\mathcal{X}$ is not online space optimal.

Next, suppose that o_3^1 is GCed at time t_0. Consider the transactional history H_1 depicted in Figure 2(b), which extends H. We claim that the second step of T_3 cannot read o_3^0. Indeed, T_3 starts after T_2 finished, and T_2's second step overwrote o_3^0. So, T_3's second step must read o_3^2, and so T_4 precedes T_3 in any strict serialization. Also, T_3 precedes T_5 in any strict serialization, because the first step of T_3 does not read o_1^1. From this, we get that the third and fourth steps of T_3 must read o_4^1 and o_5^1, resp. So, these object versions cannot be GCed at time t_1. Now, to show that $\mathcal{X}$ is not online space optimal, consider another STM $\mathcal{X}'$ that keeps o_3^1 at time t_0, but GCs o_4^1 and o_5^1 at time t_0. We claim that $\mathcal{X}'$ satisfies MV-permissiveness. Again, it suffices to show the live read-transactions T_1 and T_3 at time t_0 can commmit. Indeed, T_1's second and third steps read o_4^0 and o_5^0, resp., so T_1 can commit. Also, T_3's second, third and fourth steps can read o_3^1, o_4^0 and o_5^0, resp., by serializing T_3 after T_2, and so T_3 can also commit. This is illustrated in Figure 2(c). Thus,

$\mathcal{X}'$ satisfies MV-permissiveness. So, since $\mathcal{X}'$ keeps 6 object versions at t_1 and $\mathcal{X}$ keeps 7, $\mathcal{X}$ is not online space optimal. $\square$

5.2 Useless-Prefix GC

Though we have just seen that no MV-permissive STM is online space optimal, we would still like an STM to garbage collect as many old versions as it can. To this end, we define the following.

Definition 4. *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 a live 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.

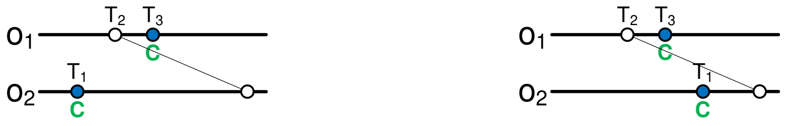
6 Inherent Limitations

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. However, we show in this section that MV-permissive STMs cannot satisfy this property.

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. In our second result in this section, we prove a stronger statement. We show that it is not possible for an MV-permissive STM to perform UP GC, even when we allow read-only transactions to write, and only require that when such a transaction runs alone, the external configurations before and after the transaction are the same. This means that read-only transactions must leave some trace of their existence, even after they have committed. In particular, even keeping current readers lists for the objects [7], or using non-zero indicators for conflict detection [5] does not suffice.

6.1 Disjoint-Access Parallelism

Theorem 2. *An 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.

Figure 3: In a weakly DAP STM T_1 does not distinguish between H_1 and H_2 and cannot be MV-permissive.

Proof. Suppose for contradiction that there exists an STM satisfying MV-permissiveness that is weakly DAP. Consider the transactional histories in Figure 3. 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 a live transaction T_2. Note that since an STM satisfies MV-permissiveness, T_3 neither aborts nor waits for T_2's termination upon a write to o_1. We claim the following. (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. So there is no STM that is both MV-permissive and weakly DAP. In the following, let s_1, s_2, s_3 denote the first steps of T_1, T_2, T_3, resp., and let s_2' denote the second step of T_2.

To show (1), note that T_1 performs the last write on o_2 before the start of T_2 in H_1. So by strict serializability, s_2' returns o_2^1.

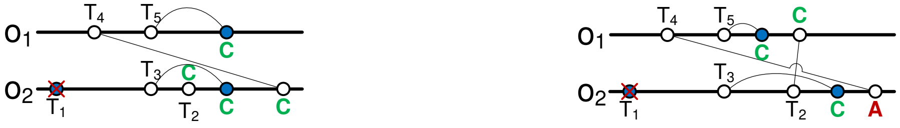
To show (2), we show that H_1 and H_2 are indistinguishable to $thr(T_2)$. We first claim that the base steps of s_1 and s_2 in H_1 do not contend. Indeed, consider another transactional history H_3 in which T_2 commits after its first step s_2. T_1 and T_2 are disjoint-access in H_3, so the base steps of s_1 and s_2 in H_3 do not contend. After s_2, $thr(T_1)$ and $thr(T_2)$ do not distinguish H_1 from H_3, because the steps of T_2 are not known ahead of time. Thus, the base steps of s_1 and s_2 in H_1 also do not contend. Next, we claim that the base steps of s_1 and s_3 in H_1 do not contend. This is because T_1 and T_3 are disjoint-access in H_3, so the base steps of s_1 and s_3 in H_3 do not contend. Since $thr(T_1)$ and $thr(T_2)$ do not distinguish H_1 from H_3 after s_3, then $thr(T_3)$ does not distinguish them after s_3. So, the base steps of s_1 and s_3 do not contend in H_1. Now, since the base steps for s_1, s_2 and s_1, s_3 in H_1 do not contend, then the configuration after the base steps of s_3 in H_1, and after the base steps of s_1 in H_2, are the same. Thus, $thr(T_2)$ does not distinguish between H_1 and H_2. So since s_2' returns o_2^1 in H_1, it also returns o_2^1 in H_2.

(3) is true because s_2 occurs before s_3 in H_2, and so s_2 returns o_1^0.

To show (4), let S be any legal sequential history that is equivalent to H_2. Since s_2 returns o_1^0 and s_2' returns o_2^1, then $T_2 \preceq_S T_3$ and $T_1 \preceq_S T_2$. Also, since T_1 starts after T_3 commits, then $T_3 \preceq_S T_1$. But then $T_1 \preceq_S T_2 \preceq_S T_3 \preceq_S T_1$, which is a contradiction. Thus, H_2 is not strictly serializable. □

6.2 Read Visibility

Theorem 3. *Suppose an 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.

Figure 4: H_1 and H_2 are indistinguishable if a read-only transaction T_2 does not leave any trace after its execution.

Proof. Suppose for contradiction that there exists an 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 4. We claim the following. (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.

To show (1), first note that the second step of T_4 can read o_2^2, since this is equivalent to the legal sequential history $T_1T_2T_3T_4T_5$. Also, any read transaction that starts after H_1 follows T_3 in real-time, and so it cannot return o_2^1. Thus, in every extension of H, a live transaction can read o_2^2 or a later version. So by the definition of UP GC, o_2^1 is GCed.

We now show (2). In H_1 and H_2, T_2 is a read-only transaction, and its execution interval does not contain steps of any other transactions. So by assumption, the external configuration before and after T_2 are the same. Thus, after T_2's second step in H_2, the only thread that distinguishes between H_1 and H_2 is $thr(T_2)$. Note that $thr(T_2)$ does not GC o_2^1, since o_2^1 is the latest version of o_2 during T_2's execution interval. Then, since o_2^1 is GCed in H_1, it is also GCed in H_2.

To show (3), assume for contradiction that T_4 commits in H_2. Let S be a legal sequential history equivalent to H_2. Since o_1^2 is GCed in H_2, then T_4 must return o_2^2 in its second read step. Thus, we have $T_3 \preceq_S T_4$. Next, we have $T_4 \preceq_S T_5$, because T_4 does not read o_1^2 in its first read step. We have $T_5 \preceq_S T_2$, because T_2 starts after T_5 commits. Finally, we have $T_2 \preceq_S T_3$, because the first step of T_2 does not return o_2^2. Combining the above, we have $T_2 \preceq_S T_3 \preceq_S T_4 \preceq_S T_5 \preceq_S T_2$, which is a contradiction. Thus, T_4 does not commit in H_2, and so the lemma is proved. □

7 UP Multi-Versioning Algorithm

We present *UP Multi-Versioning (UP-MV)*, an STM algorithm satisfying MV-permissiveness and UP GC. Section 7.1 overviews the principles underlying UP-MV's design. The data structures used by UP-MV and its algorithm are described in Section 7.2. UP-MV's properties are analyzed in Section 7.3.

7.1 Algorithm Overview and Design Principles

First we explain how the algorithm finds the versions to read and write, and then explain the garbage collection mechanism.

Versions written and read. As UP-MV satisfies MV-permissiveness, each read-only transaction commits. Almost all STMs abort an update transaction whenever its read-set is overwritten [11, 4, 16, 7]. Our first design principle mandates that we abort *only* in such situations:

Design Principle 1. *Update transaction T aborts if and only if one of the objects in its read-set has been overwritten after being read by T and before T commits.*

This rule is trivially checked at commit time by validating that each version in the read-set is still the latest one. To expedite these checks, we use a global version clock, as in TL2 [4] and LSA [16]. The clock is incremented by each committed transaction, and object versions are tagged with its values.

The writes to a transactional object o create a sequence of versions $o^0, o^1, \ldots$. Like [4, 7, 6], UP-MV defers the writes to commit time, and does not allow for "write reordering":

Design Principle 2. *When an update transaction commits, it adds a new object version as the latest one.*

Since update transactions abort whenever their read-set is overwritten, they read only the last object versions. A read-only transaction reads the latest version that it can read without violating correctness. To specify this, we define the transaction precedence relation recursively as follows: T_j *precedes* T_i if:

- T_j terminates before the start of T_i (real-time order);
- T_i reads the value written by T_j (read-after-write);
- T_i writes to object o_k, which was previously written to by T_j (write-after-write);
- T_i writes to object o_k and T_j reads the version overwritten by T_i (write-after-read); or
- $\exists T_k$ s.t. T_i precedes T_k and T_k precedes T_j.

If T_j precedes T_i, we say that T_i *follows* T_j. Note that any serialization order must respect the precedence order. We can now specify which versions are read:

Design Principle 3. *Consider a transaction T_i reading object o_j. If T_i is an update transaction, it reads the latest version. Otherwise, let T_k be the earliest update transaction that follows T_i and writes to o_j. Then T_i reads the version of o_j overwritten by T_k. If no such T_k exists, T_i reads the last version of o_j.*

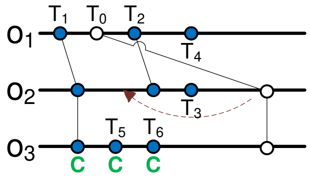


Figure 5: Transaction T_0 reads the latest object versions it can correctly read: when reading o_2 it accesses o_2^1, which was overwritten by T_2; when reading o_3, it accesses the last version.

For example, in Figure 5, when transaction T_0 reads o_2 it should read o_2^1, because this version is overwritten by T_2, which follows T_0 and writes to o_2. We say that a live transaction T_i is a *potential reader* of version o_i^j if T_i precedes o_i^{j+1}*.writer* and does not precede o_i^j*.writer*. In order to maintain the precedence information, UP-MV keeps a graph whose vertices are transactional descriptors for each transaction, and whose edges correspond to the precedence relations created by transactional steps during the run.

Note that if a read-only transaction does not conflict with any update transaction, then it has no following transactions, and therefore reads the last version of every object. Thus, by default, read-only transactions access the last object versions, which are referenced directly by *object handles*. In addition, each read-only transaction should be able to find references to relevant old object versions. But since, by UP GC, such versions may exist only as long as there are live transactions that can read them, these versions have to somehow be linked to their potential readers. This leads to the following design principle:

Design Principle 4. *Every read-only transaction T has a* map *of references from objects to old versions of which T is a potential reader.*

The responsibility for maintaining such maps lies on update transactions: before a committing update transaction writes to an object, it copies the reference to the overwritten version to all the maps of its live preceding transactions, (which are the potential readers of that version). The potential readers are found by traversing the precedence graph. In case the map already includes a version for this object, the version numbers are compared, and the earlier one is kept.

In Appendix A, we prove that our algorithm satisfies the following invariant:

Invariant 1. *Transaction T_i has o_i^j in its map if and only if o_i^j is not o_i's last version and o_i^j is the latest version that T_i can read without violating correctness.*

Garbage Collection. To satisfy the UP GC, an old object version is deleted at time t_0 if it cannot be read by any transaction after t_0. By Design Principle 3, version o_i^j may be read if and only if it has a potential reader. Version o_i^j is deleted at time t_0 if it may have no potential readers from t_0 onward. Our algorithm ensures that if there are no potential readers at time t_0, then no such readers may appear after t_0.

We deduce the following design rule for garbage collecting old object versions:

Design Principle 5. *Every old object version is deleted when its last potential reader terminates.*

In addition to removing old object versions, UP-MV's garbage collection should clean up transactional descriptors of terminated transactions from the precedence graph. As noted above, this graph is needed to allow committing transactions to copy overwritten versions to their live preceding transactions. Once a terminated transaction T has no live preceding transactions, its descriptor become useless. Hence:

Design Principle 6. *The descriptor of terminated transaction T is deleted when the last live preceding transaction of T terminates.*

7.2 UP-MV's Data Structures and Algorithm

Algorithm 1 UP-MV algorithm data structures.

```
Object Handle $o_j$:
    Version: latest ▷ latest version of the object

Version $o_j^k$:
    Data: data ▷ actual data
    Tid: writerId ▷ Id of the version's writer
    int: versionNum ▷ ordered version number of $o_j^k$
    TxnDsc[]: readers ▷ current live readers
    int: potentialCount ▷ the number of live read-only transactions that might need the version in future

TxnDsc $T_i$:
    {Live, Terminated}: status
    int: clockVal ▷ global clock at the beginning of transaction
    ⟨Object, Version⟩[]: readSet
    ⟨Object, Version⟩[]: writeSet
    TxnDsc[]: prev ▷ immediate predecessors of $T_i$
    TxnDsc[]: next ▷ immediate successors of $T_i$
    ⟨Object, Version⟩[]: toRead ▷ if $T_i$ cannot read the latest version of $o_j$, then the legal version is kept in $T_i$.toRead[$o_j$]

Global Variables:
    int: globalClock ▷ incremented by committing update txn
    TxnDsc[]: finished ▷ finished txns that have not been GCed
    ⟨Tid, TxnDsc⟩[]: txnMap
```

Memory layout. The data structures used in the algorithm are depicted in Algorithm 1. Transactional objects are accessed via object handles, which point to the last object versions. In order to facilitate garbage collection, old versions are referenced directly by their potential readers.

Each version keeps a counter of potential readers, *potentialCount*; when this counter becomes zero the version is deleted. Additionally, each version keeps the version number, *versionNum*, as read from the global clock when the version is written. Each object version also keeps the list of its current live reading transactions, *readers*, which is used by update transactions to maintain precedence information. This is where the algorithm violates read invisibility, as required for UP GC (see Section 6.2).

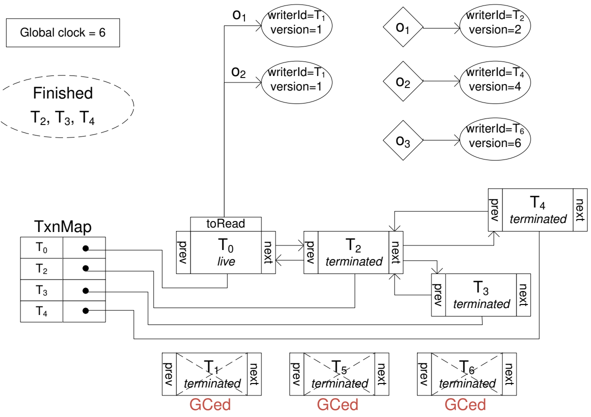


Figure 6: An example of memory layout: object handles keep last versions only, old versions are kept as long as they have potential readers, terminated transactions are GCed once they have no live preceding transactions.

Each transaction is represented by its transactional descriptor keeping the read-set and the write-set of the accessed objects. A data structure TxnMap keeps pointers to all the non-GCed transactions' descriptors. Some of the transactional descriptors point to each other, forming a subgraph of the precedence graph. Transactional steps add edges according to read-after-write, write-after-write, and write-after-read relations. Edges reflecting real-time precedence are added at startup, as we explain below. The transactional descriptor of a terminated transaction is GCed once it has no incoming edges. If transaction T_i has no live preceding transactions at the end of its run, T_i's descriptor is deleted by T_i itself. Otherwise, T_i's descriptor is deleted by the last live transaction preceding T_i when it terminates.

In order to track real-time order, the algorithm maintains a global transaction set *finished*, which holds the descriptors of all the terminated transactions that have not been GCed. A transaction T that cannot GC its descriptor inserts it to this set upon termination, and the descriptor is removed from *finished* when it is GCed. Note that *finished* is always empty in runs without conflicts. When a new transaction starts, it adds edges from every transaction in *finished* to itself. The use of this set is where the algorithm violates the DAP property, as necessary for MV-permissiveness (see Section 6.1). Although the use of a global clock, which is incremented by each committing transaction, and copied to every written version, also violates DAP, we use it only to optimize consistency checks, and it is not needed for correctness.

In Figure 6, we see the memory layout for the scenario depicted in Figure 5: a live read-only transaction T_0 precedes committed transactions $T_2 \ldots T_4$, so these transactions are not GCed, whereas committed transactions T_1, T_5, T_6, which have no live preceding transactions, are deleted.

The map of old object versions T_i may read is stored in T_i.*toRead*. Invariant 1 guarantees that if a read-only transaction T_i cannot read the last version of object o_j, then T_i.toRead contains a mapping from o_j to the old version that should be read by T_i. In Figure 6, the object versions overwritten by T_1 are referenced by its live preceding transaction T_0. All other old object versions are GCed because they have no potential readers.

Handling update transactions. The pseudo-code for update transaction T_i is depicted in Algorithm 2. At startup, transaction T_i saves the value of the global clock in its local variable *clockVal* and adds edges from all the descriptors in *finished* to itself (line 34).

Write operations postpone most of the work till the commit phase; a write operation merely updates the local copy of the object and puts it in its write-set. A read operation may only return the last version of the object. To that end, the last version's number is validated. If a read operation succeeds, T_i updates the

Algorithm 2 UP-MV algorithm for update transaction T_i.

```
 1: Write to $o_j$:
 2:     if ($o_j \in T_i$.writeSet) then update $T_i$.writeSet[$o_j$]; return
 3:     localCopy ← $o_j$.latest.clone()
 4:     writeSet[$o_j$] ← localCopy
 5:     update localCopy

 6: Read $o_j$:
 7:     if ($o_j \in T_i$.writeSet) then return $T_i$.writeSet[$o_j$]
 8:     version ← $o_j$.latest
 9:     if (version.versionNum $>$ $T_i$.clockVal) then
10:         if ¬validateReadSet() then abort
11:         clockVal ← version.versionNum
        ▷ update precedence information
12:     lastWriter ← txnRepository.get(version.writerId)
13:     if (lastWriter $\neq \perp$) then addEdge(lastWriter, $T_i$)
14:     readSet[$o_j$] ← version
15:     return version.data

16: Commit:
17:     if ¬validateReadSet() then abort
18:     overwritten ← $\emptyset$             ▷ keep the versions overwritten by $T_i$
19:     globalClock ← globalClock + 1
20:     foreach $o_j \in T_i$.writeSet do:
            ▷ update precedence info
21:         prevWriter ← txnRepository.get($o_j$.latest.writerId)
22:         if (prevWriter $\neq \perp$) then addEdge(prevWriter, $T_i$)
23:         foreach $T_j \in o_j$.latest.readers do: addEdge($T_j$, $T_i$)
            ▷ install the new version
24:         $o_j$.latest.potentialReadersCount ← 0
25:         overwritten[$o_j$] ← $o_j$.latest
26:         localCopy.versionNum ← globalClock
27:         $o_j$.latest ← localCopy
        ▷ pass the overwritten versions to the txns preceding $T_i$
28:     foreach $T_j \in T_i$.prev do:
29:         overwrittenVersions($T_j$, overwritten)

30: Startup:
31:     $T_i$.status ← Live
32:     $T_i$.clockVal ← globalClock
33:     foreach $T_j \in$ finished do:
34:         addEdge($T_j$, $T_i$)                ▷ RTO dependence

35: Termination:
36:     $T_i$.status ← Terminated
37:     finished ← finished ∪ $T_i$
38:     GC($T_i$)

39: Function GC($T_i$)
        ▷ remove the transactions with no live preceding transactions
40:     if (prev = $\emptyset$) then
41:         txnRepository ← txnRepository \ $T_i$
42:         finished ← finished \ $T_i$
43:         foreach $T_j \in T_i$.next do:
44:             $T_j$.prev ← $T_j$.prev \ $T_i$
45:             GC($T_j$)
46:         delete $T_i$'s descriptor

47: Function validateReadSet()
48:     foreach ⟨$o_j$, version⟩ ∈ $T_i$.readSet do:
49:         if $o_j$.latest $\neq$ version then return false
50:     return true

51: Function overwrittenVersions($T_j$, overwritten)
52:     if ($T_j$.status = Live) then
53:         foreach ⟨$o_i$, ver$_i$⟩ ∈ overwritten do:
54:             curVer ← $T_j$.toRead[$o_i$]
55:             if (curVer = $\perp$ ∨ curVer.versionNum > ver$_i$.versionNum)
56:                 ver$_i$.potentialCount++
57:                 $T_j$.toRead[$o_i$] ← ver$_i$
58:     foreach $T_k \in T_j$.prev do:
59:         overwrittenVersions($T_k$, overwritten)
```

precedence information: if the last version's writer T_j was not GCed, then T_i adds an edge from T_j to itself.

Transaction T_i commits successfully if and only if no object in its read-set is overwritten after being read by T_i and before T_i commits. This is checked similarly to TL2 [4], using the global clock, and without using precedence information. A commit operation starts by revalidating T_i's read-set (line 17). If the validation fails, T_i aborts. Otherwise, T_i executes the following: 1) increments the global clock; 2) for each $o_j \in T_i$.writeSet, T_i adds edges from o_j's writer and from o_j's readers to itself, and then installs the new version (lines 21–27); and 3) calls the function overwrittenVersions to update potential readers' maps with the versions overwritten by T_i (line 29).

The process of updating potential readers with overwritten versions (lines 51–59) is executed recursively for every preceding transaction. For a live transaction T_j, the overwritten versions are inserted to its *toRead* map. If for some object o_i, *toRead* already contains a version of o_i, the version with the smaller versionNum is chosen (lines 54–57). This way, the algorithm guarantees that a read-only transaction that reads o_i accesses the version overwritten by the *earliest* following transaction.

When T_i terminates, it adds its descriptor to *finished* and starts the GC procedure (lines 39–46). The transactional descriptor may be deleted if it has no incoming edges. Since deleting one transactional descriptor decreases the number of incoming edges in its successors, the GC continues recursively with them.

Handling read-only transactions. The pseudo-code for read-only transactions appears in Algorithm 3.

Algorithm 3 UP-MV algorithm for read-only transaction T_i.

```
60: Read $o_j$:
61:     if ($o_j \in T_i$.readSet) then return readSet[$o_j$].data
        ▷ find the version to read
62:     if ($o_j \in T_i$.toRead) then
63:         verToRead $\leftarrow T_i$.toRead[$o_j$]
64:     else
65:         verToRead $\leftarrow o_j$.latest
        ▷ update precedence information
66:     writer $\leftarrow$ txnRepository.get(verToRead.writerId)
67:     if (writer $\neq \perp$) then
68:         addEdge(writer, $T_i$)
        ▷ pass the overwritten versions to the preceding transactions
69:     foreach $T_j \in T_i$.prev do:
70:         overwrittenVersions($T_j$, $T_i$.toRead)
71:     verToRead.readers $\leftarrow$ verToRead.readers $\cup T_i$
72:     readSet[$o_j$] $\leftarrow$ verToRead
73:     return verToRead.data

74: Termination:
75:     $T_i$.status $\leftarrow$ Terminated
76:     finished $\leftarrow$ finished $\cup\ T_i$
77:     if ($T_i$.prev $= \emptyset$) then
78:         foreach $\langle o_j$, version$\rangle \in T_i$.readSet do:
79:             version.readers $\leftarrow$ version.readers $\setminus T_i$
80:     foreach $\langle o_j$, oldVersion$\rangle \in T_i$.toRead do:
81:         oldVer.potentialCount $\leftarrow$ oldVer.potentialCount $- 1$
82:         if (oldVer.potentialCount $= 0$) then delete oldVersion
83:     GC($T_i$)
```

To read object o_j (lines 3–6), T_i checks whether the object is in *toRead*. If not, then T_i reads the last version of o_j. Otherwise, T_i reads the version from its *toRead* list.

When a read-only transaction T_i terminates, it decrements the counter of potential readers for all the versions in its *toRead* list. If a version's number of potential readers becomes zero, the old object version is deleted (lines 80–82).

7.3 Properties

UP-MV's MV-permissiveness immediately follows from the code, since read-only transactions never abort and update transactions abort only if some object in their read-set is modified during their lifecycle. UP GC is also easy to see, since each version has a counter, which is non-zero only if the version is in the map of a read-only transaction. But by Invariant 1, T_i has o_i^j in its *toRead* map only if o_j^j is the last version T_i can read.

UP-MV's correctness follows from the following arguments:

1. The algorithm maintains precedence order correctly. That is, if T_i is a live preceding transaction of T_j, then there is a path from T_i's descriptor to T_j's.
2. The precedence graph remains acyclic, since (a) by Invariant 1, a read-only transaction can always find a version to read without creating a cycle; and (b) update transactions have no followers as long as they are alive because they abort on every conflict, and so their steps also do not create cycles.
3. Any total order that preserves the precedence order of an H is a legal serialization of H [13].

8 Conclusions

This paper studied the use of multi-versioning to reduce the number of aborts in STMs, as well as techniques for garbage collection to reduce the memory consumption of multi-versioned STMs. We first defined the property of multi-version permissiveness. Then we showed that no MV-permissive STM can guarantee to always garbage collect the maximum number of unneeded object versions. We also showed that an MV-permissive STM cannot be weakly disjoint-access parallel. We defined an achievable garbage collection property, useless-prefix GC, and showed that in an MV-permissive STM satisfying UP GC, even read-only transactions must make lasting changes to the system state. Finally, we presented an MV-permissive STM satisfying UP GC that uses visible reads and is non-DAP, showing that these conditions are not only necessary but also sufficient.

Our paper suggests a number of areas for future research. For example, while we showed that no MV-permissive STM can be online space optimal, it is interesting to consider whether there exist approximately optimal STMs. 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.

Acknowledgments

We thank Eshcar Hillel and Hagit Attiya for helpful comments. This work was partially supported by the Israeli Ministry of Science Knowledge Center on Chip Multiprocessors, Intel Corporation, Semiconductors Research Corporation (SRC), and Hasso Plattner Institute.

References

[1] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 69–78.

[2] U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[3] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.

[5] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: scalable NonZero indicators. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 13–22, New York, NY, USA, 2007. ACM.

[6] R. Ennals. Cache sensitive software transactional memory. Technical report.

[7] K. Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003.

[8] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in Transactional Memories. In *Proceedings of the 22th International Symposium on Distributed Computing*, 2008.

[9] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313, 2008.

[10] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, 2008.

[11] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[13] I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 59–68, 2009.

[14] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 1979.

[15] D. Perelman and I. Keidar. SMV: Selective Multi-Versioning STM. In *Fifth ACM SIGPLAN workshop on Transactional Computing*, 2010.

[16] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 284–298, 2006.

[17] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber. From causal to z-linearizable transactional memory. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, pages 340–341, 2007.

[18] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

A Invariant Proof

We now prove Invariant 1 presented in Section 7.1.

It was shown earlier, that history H has a legal serialization if and only if its precedence graph is acyclic [13]. We use this property to prove the following lemma:

Lemma 1. *Transaction T_i can read object version o_i^j without violating correctness if and only if T_i does not precede o_i^j.writer.*

Proof. T_i can correctly read o_i^j if and only if the read operation does not create a cycle in the precedence graph. When T_i reads o_i^j, two new precedence relations are added: $(o_i^j.\mathit{writer}, T_i)$ and $(T_i, o_i^{j+1}\mathit{writer})$.

($\Rightarrow$:) If T_i already precedes $o_i^j.\mathit{writer}$, then adding relation $(o_i^j.\mathit{writer}, T_i)$ creates a cycle in the precedence graph, so T_i cannot read o_i^j.

($\Leftarrow$:) The algorithm always installs new versions at the end, so $o_i^j.\mathit{writer}$ precedes $o_i^{j+1}.\mathit{writer}$. If T_i does not precede $o_i^j.\mathit{writer}$, then adding relation $(o_i^j.\mathit{writer}, T_i)$ cannot create a cycle in the precedence graph. Therefore, T_i can read o_i^j if it does not precede $o_i^j.\mathit{writer}$. □

The following lemma can be proven by easy induction on the steps of the algorithm:

Lemma 2. *The transactional descriptor graph of UP-MV is at any given time a subgraph of the precedence graph, which includes a path from every live transaction T_i to each of its followers.*

Invariant 2. *Let S_i^j be the set of committed update transactions following T_i that write to o_j. If S_i^j is empty then T_i's map contains no mapping for o_j. Otherwise, T_i's map contains the first version of o_j that is overwritten by a transaction in S_i^j.*

Proof. We prove the invariant by showing that it is correct at the beginning of each transaction and is preserved after each algorithm step. Upon startup, T_i's map is empty and T_i does not precede any other transaction, so $S_i^j = \emptyset$. Hence, the invariant holds.

In order to show that the invariant is preserved after each algorithm's operation, we show the following: (1) each change in T_i's mapping for o_j corresponds to a change in S_i^j, (2) S_i^j may only grow during the lifetime of T_i, (3) the invariant is preserved when a new transaction joins S_i^j. These three steps together complete the proof.

To show (1), observe that T_i's mapping for o_j changes only in the function *overwrittenVersions* (line 57), which operates on T_i's descriptor as a result of one of two events. First, a transaction T_k that follows T_i writes to o_j and commits (line 29, and recursively, line 57). In this case, $T_k \in S_i^j$. Second, a live read-only transaction T_k, which precedes some T_l that writes o_j, reads the value written by one of T_i's followers (line 70). In this case, T_i starts preceding T_l via T_k, and therefore $T_j \in S_i^j$. Each member of S_i^j writes to o_j once, Therefore, every change in T_i's mapping for o_j corresponds to a change of S_i^j.

Claim (2) follows directly from the observation that if T_i precedes T_j at time t_0, this relation persists in every extension of the history.

To show (3), we examine all the possible ways for a new transaction to join S_i^j:

- A transaction T_l that follows T_i writes to o_j and commits. In this case, T_l calls *overwrittenVersions* (line 29), which traverses recursively all the predecessors of T_l's descriptor. By Lemma 2, T_i's descriptor is a predecessor of T_l's descriptor, hence *overwrittenVersions* is executed with T_i, and in line 57, it compares T_i's mapping for o_j with the version overwritten by T_j, and chooses the version with the earlier version number. The invariant is preserved.

- A (committed) writer of o_j has a new preceding transaction T_l. This happens only when a live read-only transaction T_k, which precedes T_l, reads a value written by one of T_i's followers. In this case, T_k calls *overwrittenVersions* (line 70), which traverses recursively all the predecessors of T_k's descriptor, including T_i's descriptor (by Lemma 2). According to the invariant assumption, T_k's mapping for o_j contains the version of o_j that is overwritten by the earliest transaction in S_k^j. This version is compared with T_i's current mapping for o_j in line 57, and the version with the earlier version number is chosen. Note that if an update transaction $T_m \in S_k^j$ is not the earliest one in S_k^j, then it cannot be the earliest one in S_i^j, because $S_k^j \subseteq S_i^j$. Therefore, the invariant is preserved.

We have shown that the invariant is correct at the beginning of each transaction and is preserved after each algorithm step. □

Invariant 1 follows directly from Lemma 1 and Invariant 2:

Invariant 1. (restated) *Transaction T_i has o_i^j in its map if and only if o_i^j is not o_i's last version and o_i^j is the latest version that T_i can read without violating correctness.*