SMV: Selective Multi-Versioning STM

Dmitri Perelman   Idit Keidar

Technion Dept. of Electrical Engineering

{dima39@tx,idish@ee}.technion.ac.il

Abstract

We present *Selective Multi-Versioning (SMV)*, a new STM that reduces the number of aborts, especially those of long read-only transactions. SMV keeps old object versions as long as they might be useful for some transaction to read. SMV is able to do this while still allowing reading transactions to be invisible by relying on automatic garbage collection to dispose of obsolete versions. SMV is most suitable for read-dominated workloads, for which it achieves much better performance (up to 100% more throughput) than a single-version STM; its throughput is also up to 50% higher than that of an STM that keeps a constant number of versions per object, while also requiring 30% less memory. SMV is scalable, and its advantage increases with the number of concurrent threads. However, SMV is less ideal for read-write workloads. We discuss some possible extensions and future research directions for investigating STMs with the goal of achieving "the best of both worlds".

1. Introduction

Transactional memory [9, 20] is an increasingly popular paradigm for concurrent computing in today's and tomorrow's multi-core architectures. Most transactional memory implementations today are software toolkits, or *STMs* for short. A fundamental tenant of virtually all STMs is speculatively allowing multiple transactions to proceed concurrently, before it is known whether there are any data dependencies among them. This optimistic approach inevitably leads to aborting transactions in some cases; in particular, when such dependencies render their speculative execution inconsistent, (e.g., not linearizable). When many transactions contend on the same data objects, aborts may become frequent, which has a devastating effect on performance [2]. Reducing the number of aborts is thus an important challenge for STMs. In this paper, we focus precisely on this challenge.

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 based on conflicts [6, 7, 10], where two transactions are said to *conflict* if both access the same data object, and at least one of these accesses is a write. Note, however, that a conflict does not always entail an inconsistency. Consider, for example, the following scenario: transaction T_1 reads an object o_1, then it pauses and another transaction T_2 updates objects o_1, o_2 and commits. Assume that now T_1 attempts to read o_2. Reading the value written by T_2 would violate correctness, since T_2 cannot be serialized before T_1. But despite the conflict, T_1 may still be able to commit consistently by reading the previous version of o_2, pre-dating the one written by T_2.

More generally, a read-only transaction can *always* commit by reading a *consistent snapshot* [3] of the objects it accesses, e.g., object values that reflect updates by the transactions that committed before it began and no partial updates by concurrent transactions. Then why do such transactions abort? Simply because they cannot find the old object versions needed for such a consistent snapshot. In fact, most leading STMs [5–7, 10] keep only one version of each object. As we show below, this causes many transactions to abort, in particular in read-dominated workloads, where single-version STMs can abort up to 70% of the transactions.

Keeping multiple versions per object, a practice called *multi-versioning* [4], can mitigate this effect; (in the example above, keeping two versions of o_2 avoids the abort of T_1). Nevertheless, efficient use of multiple versions in STMs is far from trivial. Most notably, it raises the challenge of *garbage collection (GC)*: since an STM cannot keep the previous object versions forever, the question is when and how old object versions can be removed.

Ideally, one would like to keep versions that are still useful to some potential readers, and remove ones that are obsolete. Yet previous works on multi-versioned STMs did not succeed in doing so (see Section 2). Interestingly, this shortcoming stems, in part, from a seemingly unrelated consideration: previous work has noted the practical importance of making read operations *invisible* [6, 17], i.e., having no effect on shared memory. But we observe that if read-only transactions are invisible, then other transactions have know way of telling whether there still exist read-only transactions that might read the old version! So it appears that combining invisible reads with effective garbage collection is impossible.

To circumvent this apparent paradox, we use the automatic GC facility available in managed memory systems. Such systems use dedicated GC threads, which have access to all the threads' private memories. Thus, even operations that are invisible to other transactions are visible to the garbage collector.

In Section 3, we present *Selective Multi-Versioning (SMV)*, a new STM algorithm with invisible read operations that uses old object versions to reduce aborts. SMV provides wait-free termination for (almost) all read-only transactions – they neither block nor abort. SMV keeps old object versions as long as there are transactions that can safely read them, while ensuring that all obsolete object versions become *garbage collectible (GCable)*, i.e., can be reclaimed by the automatic GC service. For infrequently-updated objects, SMV typically holds exactly one version.

In Section 4 we present a preliminary evaluation of SMV. We implement SMV in Java within the STMBench7 [8] benchmark suite. We compare SMV to a TL2-style algorithm – a single-versioned STM whose basic operation is like that of TL2 [5]; and to a k-versioned variant of the same algorithm, which keeps k previous versions per object, similarly to LSA [17]. We evaluate the algorithms on a 32-core machine by running up to 32 threads. Our findings show that SMV is most suitable for read-dominated workloads with long-running transactions. For example, with 32 threads, the throughput of SMV is more than double that of the TL2-style algorithm, and a 50% higher than those of 4- and 8-versioned algorithms. This is thanks to the much higher commit ratio of SMV: 97% compared to 54% and 76% with the other algorithms. At the same time, SMV may consume as much as 30% less memory than the studied multiversioned algorithm. Moreover, SMV scales better than the other algorithms, and its advantage becomes more pronounced when we increase the number of concurrent threads, as conflicts become more likely.

Indeed, in read-dominated workloads, SMV can reap substantial benefits from avoiding aborts of long-running transactions. As explained above, aborts are avoided thanks to keeping old object versions. However, keeping this information may incur a large overhead in workloads with many (short) update transactions. In order to evaluate this overhead, we examine the performance of SMV in read-write workloads with frequent updates. Although in this scenario SMV still reduces the abort rate compared to the other algorithms, its throughput is inferior to theirs.

In Section 5, we discuss some future research directions, including possible extensions that may reduce SMV's overhead in workloads with frequent short writes. These ideas were not yet evaluated; we defer their detailed study to a later (and more complete) version of this work.

In summary, SMV reduces the number of aborts in STMs by storing object versions as long as they might be useful. It achieves a simple design by relying on existing automated GC mechanisms. Though SMV reduces the abort rate in all workloads, this translates to performance benefits only in read-dominated ones, where many aborts are avoided and the overhead for keeping multiple versions is low. The question of whether this tradeoff is inherent or "the best of both worlds" is attainable remains open. We outline some ideas for further investigation of this question.

2. Related Work

As noted above, most existing STMs are single-versioned. Of these, SMV is most closely related to TL2 [5], from which we borrow the ideas of invisible reads, commit-time locking of updated objects, and a global version clock for consistency checking. In a sense, SMV may be seen as a multi-versioned extension of TL2.

Among multi-versioned STMs, the closest to SMV is LSA [17]. LSA, as well as its snapshot-isolation variation [18], implements a simple solution to garbage collection: it keeps a constant number of versions for each object. However, this approach leads to storing versions that are too old to be of use to any transaction on the one hand, and to aborting transactions because they need older versions than the ones stored on the other. The number of versions kept defines a tradeoff between the two; the authors of [17] use 8. In contrast, SMV keeps versions as long as they might be useful for ongoing transactions, and makes them GCable by an automatic garbage collector as soon as they are not. For infrequently updated objects, SMV typically keeps a single version.

Other previous suggestions for multi-versioned STMs (including our own) [2, 12, 14] were based on cycle detection in the conflict graph, a data structure representing all data dependencies among transactions. Cycle detection incurs a high cost (quadratic in the number of transactions), which is clearly not practical. Moreover, it requires reads to be visible in order to detect future conflicts, which can be detrimental to performance. Our earlier work [12] specified complex GC rules as to when old versions can be removed. However, the algorithm was too complex to be amenable to practical implementation, and did not specify when these GC rules ought to be checked. Aydonat and Abdelrahman [2] propose to keep each version as long as there exist transactions that were active at the time it was created, but the authors do not specify how this rule can be implemented efficiently. Other theoretical suggestions for multi-versioned STMs ignored the issue of GC altogether [14]. In contrast, in this paper we present a simple algorithm, which implements invisible reads, and exploits the automatic GC available in languages with managed memory.

Another approach for increasing concurrency of conflicting transactions was presented by Ramadan et al. [16]. This work presents an implementation of the dependence-aware transactional memory (DASTM), in which the conflicting transactions commit according to their conflict serialization order. For example, if a transaction T_r reads an object written by an active transaction T_w (W$\rightarrow$R conflict), then T_r should read the value written by T_w and the commit/abort of T_r is postponed till the commit/abort of T_w. DASTM accepts every conflict-serializable transactional interleaving and in this way DASTM, similarly to SMV, has the goal of increasing concurrency. The main difference is that in DASTM some transactions are actually paused till the termination of conflicting transactions, while in SMV the transactions progress independently. DASTM technique of reading the values of uncommitted transactions might cause cascading aborts. Additionally, reads, which are invisible in SMV must be visible in DASTM because the transaction should know all its conflicting predecessors.

3. Selective Multi-Versioning Algorithm

We present Selective Multi-Versioning, a new object-based STM algorithm. Section 3.1 presents the principle underlying SMV's design. The data structures used by SMV are described in Section 3.2. Section 3.3 presents the basic algorithm, and Section 3.4 discusses some practical optimizations.

3.1 Design Principles

SMV's main goal is to reduce aborts in read-dominated workloads, especially ones with long transactions. SMV is based on the following design choices:

Invisible reads. Read operations can only modify the reading transaction's private memory, and do not affect global memory. Invisible reads have been argued to be important for performance, especially in multi-core systems, where updates to global memory cause caches to thrash [6, 17].

Multi-versioning for reads. Read-only transactions usually commit in a wait-free manner. We achieve this by allowing read-only transactions T_i to observe a consistent snapshot corresponding to T_i's start time. The optimization we present in Section 3.4 may "give up" on some long running transactions, hence this property only "usually" holds.

Managed garbage collection based on real-time order. Old object versions are removed once there are no longer live read-only transactions that can consistently read them. To achieve this with invisible reads, SMV relies on the omniscient garbage collection mechanism available in managed memory systems. Thus, SMV needs only to ensure that unneeded data is GCable, i.e., it is not referenced by any live memory object. Note that this approach mandates that object handles will not point to old versions, in order to render old versions GCable.

Global version clock. Like TL2 [5] and LSA [17], SMV uses a global version clock to detect conflicts. Each transaction reads the clock when it begins, and update transactions increment the clock upon commit. Each object is tagged with the version clock of the transaction that wrote it. Though the global version clock is a contention-point, many practical optimizations were introduced to reduce the overhead associated with it [5, 19]. Such optimizations are orthogonal to our work, and are therefore beyond the scope of this paper. In a companion paper [15], we prove that global contention is essential for any multi-versioned STM that performs garbage collection based on real-time order (more formally, such an STM cannot be disjoint access parallel [11]).

3.2 Overview of Data Structures

As usual in object-based STMs, objects are accessed via *object handles*. An object handle includes a pointer to the memory location of the current version of the data, as well as a *versioned lock* [5], i.e., a variable holding the global version number associated with the transaction that wrote the data and a lock bit.

In order to facilitate garbage collection, object handles in SMV point only to the latest (current) version of each object. Pointers to older object versions are kept in a different data structure, namely a list of *time points*, as we explain below.

Global version numbers are generated using a global version clock. However, unlike previous implementations of such clocks [5, 17], SMV maintains the version clock as a linked list of time points, rather than a scalar variable. Upon commit, an update transaction adds a new time point (succeeding the current time point) at the end of the list. For example, when the current global version is 100, a committing update transaction adds a time point holding 101, and adds a pointer to it from the time point holding 100.

The variable *curPoint* points to the time point added by the latest committed update transaction. When a read-only transaction begins, it keeps, (in its local variable *startTP*), a pointer to the current time point. The pointer is cleared upon commit, making old time points at the head of the list GCable.

Old object versions are pointed to by the time points associated with the transactions that over-wrote them: A committing transaction T_w creates a new time point and a pointer from this time point to the previous version of every object in its write set before diverting the object handle to the new version. Reading transactions, in turn, traverse the time points, starting from *StartTP*, to find old versions. Old versions become GCable once the time points pointing to them do, i.e., when all active transactions are newer than the transaction that over-wrote them.

Figure 1 illustrates the commit of an update transaction T_w that writes to object o_1. In this example, T_w and a read-only transaction T_r both start at time 9, and hence T_r points to this time point. The previous update of o_1 was associated with version 5. When T_w commits, it adds time point 10, which points to version 5 of o_1, and changes o_1's object handle to point to the new data, which is associated with version 10. Assume T_r then wants to read object o_1. T_r determines that it cannot read the latest version because its version (10) is later than T_r's start time. Hence, T_r traverses the time points from time point 9, until it finds the *first* time point that holds some version of o_1 (time point 10 in our case).

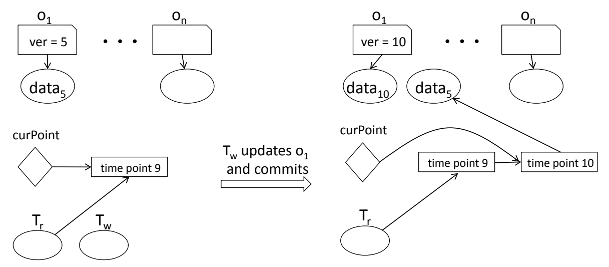


Figure 1. Time point 10 installed by T_w references the over-written version of o_1.

3.3 Basic Algorithm

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 [5] and LSA [17]. 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. Conflicts are detected using the global time points list. Upon starting, T_i saves the value of the latest time point in the local variable *upperBound*. Roughly speaking, *upperBound* holds the latest time at which an object in T_i's read-set is allowed to have been over-written.

A read operation of object o_j first reads the latest version of o_j via the object handle, and then checks whether this version is valid for T_i (function *validateRead*, lines 30–38). The validation procedure first verifies that o_j is not locked. Then, it tries to validate that o_j's version is less than or equal to T_i.upperBound. If it is not, then T_i tries to expand its upperBound to o_j's version by checking that none of the objects in its read-set has been over-written (function *validateReadSet*). If this attempt fails, the transaction is aborted.

A write operation (lines 10–13) is also invisible to other transactions, as it postpones the actual work until the time of commit. Write creates a copy of the object's latest version for local updates, and puts it in T_i's local write set.

Algorithm 1 SMV algorithm for update transaction T_i.

```
1:  Upon Startup:
2:      $T_i$.upperBound ← curPoint→time

3:  Read $o_j$:
4:      if ($o_j \in T_i$.writeSet) then return $T_i$.writeSet.get($o_j$)
5:      data ← $o_j$.latest
6:      if ¬validateRead($o_j$) then abort
7:      readSet.put(⟨$o_j$, data⟩)
8:      return data

9:  Write to $o_j$:
10:     if ($o_j \in T_i$.writeSet) then update $T_i$.writeSet.get($o_j$); return
11:     localCopy ← $o_j$.latest.clone()
12:     writeSet.put(⟨$o_j$, localCopy⟩)
13:     update localCopy

14: Commit:
15:     foreach $o_j \in T_i$.writeSet do: $o_j$.lock
16:     if ¬validateReadSet() then abort
        ▷ newTP will reference the overwritten data
17:     newTP ← new time point
18:     foreach $o_j \in T_i$.writeSet do:
19:         newTP.prevVersions.put(⟨$o_j$, $o_j$.latest⟩)
20:     timeLock.lock()
21:     newTP.time ← curPoint → time + 1
        ▷ update and unlock the objects
22:     foreach ⟨$o_j$, data⟩ ∈ $T_i$.writeSet do:
23:         $o_j$.version ← newTP.time
24:         $o_j$.latest ← data
25:         $o_j$.unlock()
26:     curPoint.next ← newTP
27:     curPoint ← newTP
28:     timeLock.unlock()

29: Function validateRead(Object $o_j$)
30:     if $o_j$.isLocked() then return false
31:     if ($o_j$.version > $T_i$.upperBound) then ▷ $o_j$ has been over-written
32:         if ¬validateReadSet() return false
33:         upperBound ← $o_j$.version
34:     return true

35: Function validateReadSet() ▷ verify that none of the objects in the
        read-set has been over-written after being read by $T_i$
36:     foreach ⟨$o_j$, data⟩ ∈ $T_i$.readSet do:
37:         if $o_j$.isLocked() ∨ $o_j$.latest ≠ data then return false
38:     return true
```

Commit (lines 15–28) consists of the following steps:

1. Lock the objects in the write set (line 15). Deadlocks may be detected by standard mechanisms (e.g., timeouts or Dreadlocks [13]), or may be avoided if acquired according to the same order by every transaction.
2. Validate the read set.
3. Create a new time point *newTP* (lines 17–19), referencing all the versions over-written by T_i (from T_i's write set). For the sake of simplicity, we first assume that commit locks the time

points list (line 20). We will show in Section 3.4 how we avoid such locking.

4. Set the value of *newTP* to the successor of *curTP*.
5. Update and unlock the objects in the write set (lines 22–25). Set their new version numbers to the value of *curTP*.
6. Install the new time point at the end of the list and unlock the time points list (lines 26–28).

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.time) then return latestData
6:      TP ← startTP.next
7:      while(TP ≠ ⊥) do:
8:          if $o_j$ ∈ TP.prevVersions then return TP.prevVersions.get($o_j$)
9:          TP ← TP.next
10: Commit:
11:     $T_i$.startTP ← ⊥
```

Handling read-only transactions. The pseudo-code for read-only transactions appears in Algorithm 2. The basic SMV algorithm guarantees that every read-only transaction T_i commits in a wait-free manner. 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 time point (line 2); we refer to the time value of startTP as T_i*'s start time*. For each object o_j, the latest versions written to o_j before T_i's start time should be read. When T_i reads an object o_j whose latest version is greater than its start time, it traverses the list of time points until it finds any version of o_j. Some version is guaranteed to be found, because the updating transaction that over-wrote o_j has added a time point referencing the over-written version. 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.

3.4 Optimizations

Allowing concurrent access to the time points list. We show how to avoid locking the time points list (lines 20, 28), so that update transactions with disjoint write-sets are able to commit concurrently. We first explain the reason for using the lock. In order to update the objects in the write-set, the updating transaction must know the new version number. The version number must be chosen and added to the list as part of the same atomic action, so that other transactions will not get the same number. But if a transaction makes the new time point available before it updates its write-set, then some read-only transaction might observe an inconsistent state, as exemplified by the following scenario: transaction T_w 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 a new time point 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* $\rightarrow$ *time* $= 10$. A new read-only transaction starts with time 10 and successfully reads the new value of o_1 and the old value of o_2, because they are both less than or equal to 10, obtaining an inconsistent snapshot. We conclude that the new time point should not become available to reading transactions before the objects are updated.

To preserve consistency without locking the time points list, we add an additional boolean field *ready* to the time point data 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 *curReadyPoint* variable, which references the latest time point in the ready prefix of the list. When a new read-only transaction starts, its *startTP* variable references *curReadyPoint* (instead of *curPoint* as in the original algorithm). In the scenario above, the new read-only transaction has a start time of 9, because the new time point with value 10 is still not ready. Hence, the new transaction does not return the new value of o_1 before the new value of o_2 is written. Generally, the use of *curReadyPoint* guarantees that if a read-only transaction T_r reads an object version written by T_w, then T_w has finished writing all the objects in its write-set.

Limiting the length of read operation traversals. In the basic algorithm described above, a read-only transaction T_r might need to traverse an unbounded number of time points from its start point until it finds a time point referencing a version of some object o_j. In case many (short) writes overlap T_r, the overhead of traversing the list will degrade the performance of T_r to the point that it will not justify the benefit of avoiding the abort.

In order to avoid this undesired behavior, we define a system parameter, *WindowSize*, which limits the number of time points a read operation may traverse. If an appropriate version is not found after traversing WindowSize time points, the transaction aborts. Such aborts of long-running transactions also allow GC of old time points. Using a limited WindowSize compromises the guarantee of unabortable read-only transactions, but significantly improves the algorithm's performance in some workloads (see Section 4).

4. Preliminary Evaluation

4.1 Experiment Setup

In order to evaluate the performance of our algorithm we use the Java version of the STMBench7 [8] evaluation framework. This framework includes a benchmark suite that aims to simulate different behaviors of real-world programs by involving read-only and update transactions of different lengths over the large data structures. It supports various workload types, including *read-dominated workloads*, where the share of read-only transactions is higher than that of update transactions, and *read-write workloads*, where there is no clear domination of either read-only or update transactions. There is no straightforward way to plug-in current STM toolkits into STMBench7, and therefore one needs to implement the algorithm directly in this framework.

Our evaluation aims to check the specific novel aspect introduced by SMV, namely, keeping and garbage collecting multiple versions using time points. Hence, we compare SMV to the closest well-known algorithm, namely TL2. Specifically, we implement the following algorithms in STMBench7:

SMV with the WindowSize set to 100.

SMVUnlimited – an unoptimized version of SMV, in which the WindowSize parameter is ∞, i.e., there is no limit on the number of time points traversed during the read operation.

TL2-style – A single-versioned STM mimicking the basic operation of TL2 [5] with a single central global version clock.

k-versioned – an STM similar to TL2, in which each object keeps k versions, (like in LSA). When a read-only transaction cannot read the latest version of an object, it checks if it may read any of the previous k versions. If all are too old or if the object is locked, the transaction aborts.

The main difference between our k-versioned algorithm and LSA is in the treatment of concurrent writers: we use commit-time locking as in TL2, whereas writers in LSA are visible and use an additional level of indirection for atomic commit. The authors of LSA have chosen to keep 8 versions. In this section, we experiment with 4- and 8-versioned algorithms.

It is worth noting that our implementation of the TL2-style algorithm does not use many of the software optimizations used in the original one. Instead, we use a common code platform to compare the algorithmic issues and not the engineering optimizations, which we believe may be applicable to each of the compared alternatives.

The benchmarks are run on a shared-memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor. The server was not used by other applications during our experiments. To ensure that the only limiting factor for scalability is memory contention, we ran at most 32 threads for each workload, corresponding to the maximal true concurrency provided by the machine.

In each test, each thread executes 2000 random transactions on the data structures of STMBench7 distributed according to the specific benchmark's characteristics. Each data point in the graphs shows an average taken over all threads. Each test is repeated multiple times to ensure that the results are not spurious. The same seeds are used for the random number generators for all algorithms, so that they all execute exactly the same workload in every test. STMBench7 starts measuring performance after an initialization stage, in which it loads all the classes and builds an initial data structure, thus eliminating any "cold start" effect.

Memory usage is evaluated by measuring JVM's allocated heap space. We note that this approach can be inaccurate, and might not give a clear picture of the memory demands of transactional objects, due to a large number of non-transactional immutable objects kept by the benchmark. In the future, we would like to design a more precise way for evaluating the memory usage of different TM algorithms.

4.2 Results

Read-dominated workloads. We first study read-dominated workloads, i.e., runs in which the rate of long running read-only transactions is much higher than the rate of update transactions. Read-dominated workloads emphasize the strong sides of SMV:

1. Since this benchmark includes long read-only transactions, the old versions SMV keeps are used intensively, leading to a high commit ratio.
2. The rate of update transaction commits is relatively low. Therefore, time points are added to the global list infrequently. Thus, read-only transactions do not need to traverse many time points when looking for old object versions, and SMV's overhead is low.

In Figure 2, we depict the commit ratio and the throughput of the studied algorithms in read-dominated workloads. In Figure 2(a) we can clearly see the deterioration of the commit ratio for the other algorithms. We see that keeping a limited number of previous versions increases the commit ratio from 0.55 to 0.75 in the k-versioned algorithm. However, there is no significant difference between keeping four or eight versions. We believe this might be because many of the aborts are due to objects being locked rather than no appropriate version being available. In contrast, SMV's commit ratio remains above 0.97 for both the optimized and unoptimized versions of SMV.

Despite the fact that SMV provides a better commit ratio than k-versioned algorithms, it consumes less memory. This is because our algorithm strives to keep only versions with potential readers. For example, SMV may keep a lot of versions for frequently updated "hotspot" objects, while keeping only one (the latest) version for an object with a low update rate. In Figure 3, we see the average amount of heap-allocated memory space of the different algorithms during the benchmark run. We can see that the memory demands of SMV are smaller than those of 4- and 8-versioned algorithms.

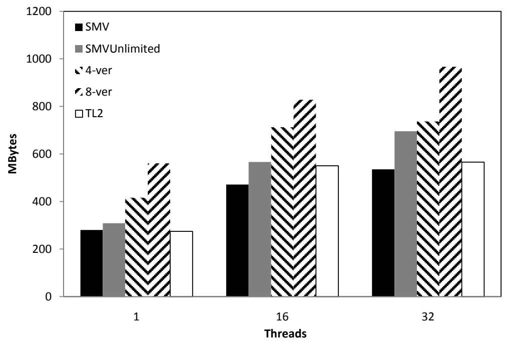


Figure 3. Average memory consumption in an STMBench7 read-dominated workload. SMV consumes less memory than k-versioned algorithms.

The high commit ratio achieved in this benchmark, along with the low overhead for reads, make SMV highly efficient. In Figure 2(b) we can see that the throughput of SMV grows as the number of threads increases (up to 32 threads), and it is substantially higher than achieved by other algorithms, which suffer from a high abort rate with a large number of threads: the throughput of the TL2-style algorithm is 308 txs/sec, compared to 623 txs/sec for SMV for 32 threads.

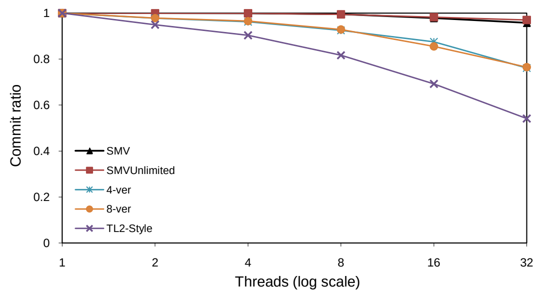
Read-write workloads. We now compare the performance of the algorithms in read-write workloads. Read-write workloads present the worst-case scenario for SMV, due to the following reasons:

1. Update transactions cannot leverage the multiple versions stored by SMV. Therefore, their commit ratio is the same as in the other algorithms.
2. The rate of update transaction commits is relatively high, and the overhead of adding many new time points to the global time points list is significant. Moreover, read operations that need to look for old object versions have to traverse long list suffixes, imposing a high computational overhead on read-only transactions.

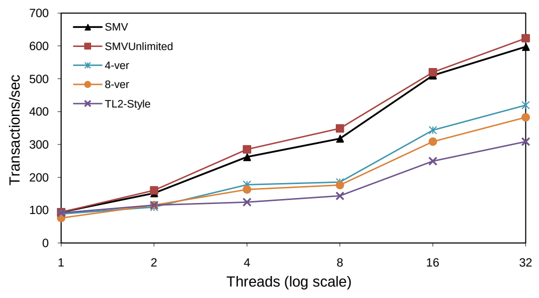
In Figure 4, we depict the commit ratio and the throughput of the algorithms in the read-write workload. The commit ratio (Figure 4(a)) of SMV, like those of the other algorithms, drops significantly as the number of threads grows. This decrease is due to conflicting update transactions. However, SMV's commit ratio is still much better than those of TL2-style and k-versioned algorithms because of the reduced abort rate of read-only transactions.

In Figure 5, we see the average amount of heap-allocated space during the benchmark run. We can see that the memory consumption of SMVUnlimited becomes much worse when the number of threads increases. This happens because long read-only transactions progress slowly, and therefore do not allow for efficient GC of the time points. The memory consumption of SMV, however, remains lower than that of the k-versioned algorithm, even for a large number of threads.

In Figure 4(b) we see that the overhead of traversing long suffixes of the time points list becomes excessive for high rates of committing transactions: indeed, the throughput of SMVUnlimited drops for a large numbers of threads. This problem is mitigated when we limit the number of traversed time points, emphasizing the importance of the *WindowSize* parameter. Nevertheless, SMV is still inferior to the other algorithms in this workload. The TL2-style algorithm, which has the lowest overhead, achieves the best performance.

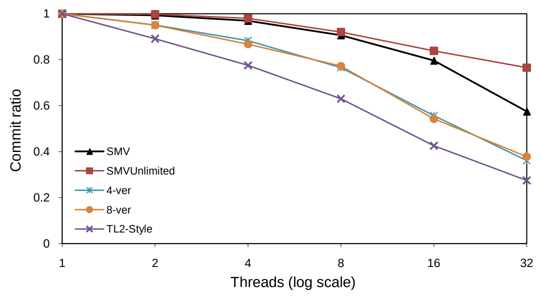


(a) Commit ratio in a read dominated workload.

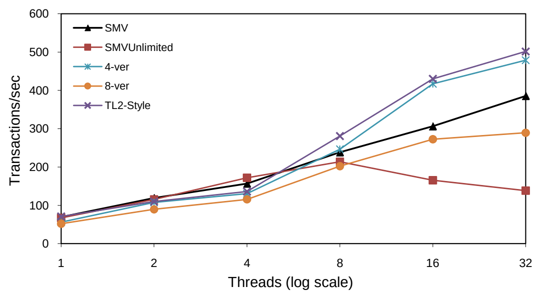


(b) Throughput in a read dominated workload.

Figure 2. Throughput and commit ratio for STMBench7 read-dominated workload. SMV gives the best throughput and commit ratio.



(a) Commit ratio in a read-write workload.



(b) Throughput in a read-write workload.

Figure 4. Throughput and commit ratio for STMBench7 read-write workload. Though giving a best commit-abort ratio, the throughput of SMVUnlimited drops for the large number of threads. The throughput of SMV is similar to those of TL2-style algorithms for small number of threads and it becomes 20% worse when the number of threads increases to 32.

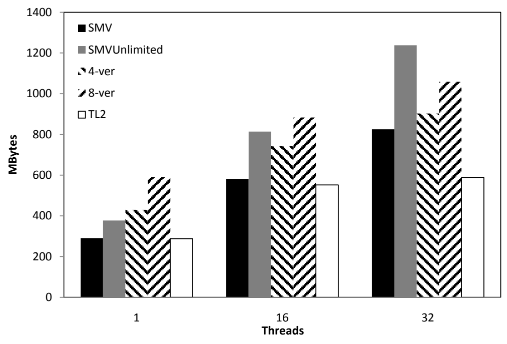


Figure 5. Average memory consumption in STMBench7 read-write workload. The memory consumption of SMVUnlimited does not scale well as the number of threads grows, whereas SMV's memory consumption is lower than that of the k versioned algorithms.

5. Discussion and Future Research Directions

We presented Selective Multi-Versioning, an STM algorithm that reduces the number of aborts, and is especially effective in read-dominated workloads. Beyond the SMV algorithm, there are two general take-away points from this paper: (1) the tradeoff between commit rates and overhead; and (2) the use of auxiliary threads that have access to private memories to complement invisible transactional operations. We now discuss these two points, as well as future research directions related to both of them.

5.1 Commit rates vs. overhead

There is an inherent tradeoff between improved commit rates on the one hand, and the overhead associated with keeping, finding, and removing old versions on the other. There are two complementary approaches to studying this tradeoff.

First, formal analysis can explore the *theoretical* limitations of multi-versioning. We are currently working on a (theoretical) companion paper [15] that proves inherent limitations on STMs in terms of the relationships between three factors: (1) the conditions under which aborts can be ensured to be avoided (*permissiveness* [12]), (2) the visibility level of transactional operations, and (3) the ability to garbage collect object versions that are no longer useful to any live transaction.

The second approach is to try to push the envelope of *practical* STM implementations further along. We now discuss our initial thoughts on how to reduce the overhead without forfeiting low abort rate. We observe that SMV's overhead directly depends on the frequency of update transactions: First, every update transaction in SMV does more work than in a single-versioned STM like TL2– it allocates a new time point data structure instead of simply updating an existing version clock. Second, the overhead of read-only transactions grows with the number of update transactions they overlap, because every concurrently committing update transaction adds to the time points list they might need to traverse. Third, the memory allocation of SMV, and with it the cost of automatic garbage collection, also increase with the frequency of update transactions.

Thus, it appears that the main limiting factor for SMV's performance is the frequency of creating new time points. Our first idea therefore focuses on allocating fewer time points.

Allocating a new time point serves two purposes. The first goal is to distinguish between two versions of the same object, in case this object is referenced by both the new and the previous time points. In this case, creating a new time point is essential for correctness. The second goal, which is not required for correctness, is allowing a more aggressive garbage collection— object versions stored in a previous time point may be garbage collected while there are still live read-only transactions pointing the new time point.

In case the write set of a transaction T does not intersect the set of objects stored in the latest time point, T can safely add its objects to the latest time point, and increment the version of that time point. Doing so reduces the overhead, both for T and for subsequent read-only transactions, but it incurs a memory cost by delaying garbage collection. We plan to experiment with different policies for choosing when to create a new time point.

SMV's overhead can be further reduced via clever use of auxiliary threads, as discussed in the next section.

5.2 Auxiliary threads

Implementing transactional operations in an invisible way has been argued to be important for performance. On the other hand, it limits the information available to transactions, which poses some challenges for STM designers.

In this paper, we used an auxiliary GC thread in order to circumvent one of the limitations of invisibility, namely, the inability to garbage collect old object versions once they are no longer useful for any live transaction. The auxiliary thread has access to all memory, including the "invisible" private memory of read-only transactions. Although, from a theoretical standpoint, this means that operations are not strictly invisible, from a practical perspective, they are "invisible enough", since the GC thread is only run infrequently, and the vast majority of transactional operations can continue to be served from local caches without interference.

While in SMV we relied upon one particular pre-existing auxiliary thread, we believe that the same approach can have many additional uses, as STMs can design their own auxiliary threads for a variety of purposes.

In the context of SMV, we plan to develop a custom GC thread as part of the application, to replace the system's automated one. This will allow SMV to be used in environments without managed memory, similarly to Boehm's garbage collector [1]. The custom GC can also work more efficiently than the automated one, by explicitly looking for time points with no pointers from startTP variables. In addition, this approach will allow for a number of optimizations to improve SMV's performance.

For example, there is a high overhead of traversing long time points lists in mixed workloads because automatic GC prevents object handles from directly referencing old object versions. But if we use our own custom GC thread, we can keep "soft" pointers to the old version: the custom GC thread can identify versions for removal using startTP pointers only, and upon deciding to remove a version, it can disconnect the pointer to it from the object handle.

We have only scratched the surface of possible uses of auxiliary threads. We hope that future work will find many additional creative ways to exploit them, in the context of STM and beyond.

Acknowledgments

We would like to thank Michal Kapalka for his help in using STMBench7. We thank Nir Shavit and the members of our group (Liat Atsmon, Nathaniel Azuelos, Dmitri Basin, Ittay Eyal, Rui Fan, Max Gurevich, Zvika Guz and Oved Itzhak) for fruitful discussions. This work was partially supported by the Israeli Ministry of Science Knowledge Center on Chip Multi-Processors, and by the Hasso Plattner Institute.

References

[1] http://www.hpl.hp.com/personal/hans_boehm/gc/.

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

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, 1995.

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

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

[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, M. Kapalka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the Second European Systems Conference*, 2007.

[9] 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.

[10] 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.

[11] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160, 1994.

[12] 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.

[13] E. Koskinen and M. Herlihy. Dreadlocks: efficient deadlock detection. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 297–303, 2008.

[14] J. Napper and L. Alvisi. Lock-free serializable transactions. Technical report, The University of Texas at Austin, 2005.

[15] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in transactional memory (submitted). Technical report, Technion, 2010.

[16] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. *SIGPLAN Not.*, 44(4):163–172, 2009.

[17] 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.

[18] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2006.

[19] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 221–228, 2007.

[20] 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.