SMV: Selective Multi-Versioning STM

Dmitri Perelman*, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar

Technion, Israel Institute of Technology

{dima39@tx,szaparka@t2,smalish@t2,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. It is able to do so 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 performs better than previous solutions. It has an up to $\times 7$ throughput improvement over a single-version STM and more than a two-fold improvement over an STM keeping a constant number of versions per object. We show that the memory consumption of algorithms keeping a constant number of versions per object might grow exponentially with the number of objects, while SMV operates successfully even in systems with stringent memory constraints.

1 Introduction

Software Transactional Memory (STM) [20, 32] is an increasingly popular paradigm for concurrent computing in multi-core architectures. STMs speculatively allow multiple transactions to proceed concurrently, which leads to aborting transactions in some cases. As system load increases, aborts may become frequent, especially in the presence of long-running transactions, and may have a devastating effect on performance [3]. Reducing the number of aborts is thus an important challenge for STMs.

Of particular interest in this context is reducing the abort rate of read-only transactions. Read-only transactions play a significant role in various types of applications, including linearizable data structures with a strong prevalence of read-only operations [21], or client-server applications where an STM infrastructure replaces a traditional DBMS approach (e.g., FenixEDU web application [9]). Particularly long read-only transactions are employed for taking consistent snapshots of dynamically updated systems, which are then used for checkpointing, process replication, monitoring program execution, gathering system statistics, etc.

Unfortunately, long read-only transactions in current leading STMs tend to be repeatedly aborted for arbitrarily long periods of time. As we show below, the time for completing such a transaction varies significantly under contention, to the point that some read-only transactions simply cannot be executed without “stopping the world”. As mentioned by Cliff Click [1], this kind of instability is one of the primary practical disadvantages of STM; Click mentions *multi-versioning* [5] (i.e., keeping multiple versions per object), as a promising way to make program performance more predictable.

* This work was partially supported by Hasso Plattner Institute.

Indeed, by keeping multiple versions it is possible to assure that each read-only transaction successfully commits by reading a *consistent snapshot* [4] of the objects it accesses, e.g., values that reflect updates by transactions that committed before it began and no partial updates of concurrent transactions. This way, multiple versions have the potential to improve the performance of single-versioned STMs [19, 16, 13, 11], which, as we show below, might waste as much as 80% of their time because of aborts in benchmarks with many read-only transactions.

However, using multiple versions introduces the challenge of their efficient garbage collection. As we show below, a simple approach of keeping a constant number of versions for each object does not provide enough of a performance benefit, and, even worse, can cause severe memory problems in long executions. We further demonstrate in Section 3 that the memory consumption of algorithms keeping k versions per object might grow exponentially with the number of objects. The challenge is, therefore, to devise an approach for efficient management of old object versions.

In Section 4, we present *Selective Multi-Versioning (SMV)*, a novel STM algorithm that addresses this challenge. SMV keeps old object versions that are still useful to potential readers, and removes ones that are obsolete. This way, read-only transactions can always complete – they neither block nor abort – while for infrequently-updated objects only one version is kept most of the time.

SMV achieves this while allowing read-only transactions to remain *invisible* [13], i.e., having no effect on shared memory. At first glance, combining invisible reads with effective garbage collection may seem impossible — if read-only transactions are invisible, then other transactions have no way of telling whether potential readers of an old version still exist! To circumvent this apparent paradox, we exploit separate GC threads, such as those available in managed memory systems. Such threads have access to all the threads' private memories, so that even operations that are invisible to other transactions are visible to the garbage collector. SMV ensures that old object versions become *garbage collectible* once there are no transactions that can safely read them.

In Section 5 we evaluate different aspects of SMV's performance. We implement SMV in Java and study its behavior for a number of benchmarks (red-black tree microbenchmark, STMBench7 [18] and Vacation [8]).

We find that SMV is extremely efficient for read-dominated workloads with long-running transactions. For example, in STMBench7 with 64 threads, the throughput of SMV is seven times higher than that of TL2 and more than double than those of 2- and 8-versioned STMs. Furthermore, in an application with one thread constantly taking snapshots and the others running update transactions, neither TL2 nor the k-versioned STM succeeds in taking a snapshot, even when only one concurrent updater is running. The performance of SMV remains stable for any number of concurrent updaters.

We compare the memory demands of the algorithms by limiting Java heap size. Whereas k-versioned STMs crash with a Java `OutOfMemoryException`, SMV continues to run, and its throughput is degraded by less than 25% even under stringent memory constraints.

In summary, we present the new approach for keeping multiple versions, which allows read-only transactions to stay invisible and delegates the cleanup task to the already existing GC mechanisms. Our conclusions appear in Section 6.

2 Related Work

As noted above, most existing STMs are single-versioned. Of these, SMV is most closely related to TL2 [11], 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 way, SMV can be seen as a multi-versioned extension of TL2.

The best-known representative of multi-versioned STMs is LSA [29]. LSA, as well as its snapshot-isolation variation [30], 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. 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.

Another multi-versioned STM, JVSTM [7], maintains a priority queue of all active transactions, sorted by their start time. A cleanup thread waits until the transaction at the head of the queue (the oldest transaction) is finished. When that happens, the cleanup process iterates over the objects overwritten by the committed transaction and discards their previous versions. Thus, while also keeping versions only as long as active transactions might read them, the GC mechanism of JVSTM imposes an additional overhead for transaction startup and termination (including both update and read-only transactions).

In a recent paper [15], the authors improved the GC mechanism of JVSTM by maintaining a global list of per-thread transactional contexts, each keeping information about the latest needed versions. A special cleanup thread iterates periodically over this list and thus finds the versions that can be discarded. This improvement, however, does not eliminate the need for a special cleanup thread, which should run in addition to Java GC threads. JVSTM read-only transactions still need to write to the global memory. In contrast, in this paper we present a simple algorithm with invisible read-only transactions, which exploits the automatic GC available in languages with managed memory.

Other previous suggestions for multi-versioned STMs [3, 25, 22, 6, 27] were based on cycle detection in the conflict graph, a data structure representing all data dependencies among transactions. This approach 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. Nevertheless, this approach can allow for more accurate garbage collection than practical systems like SMV, which do not maintain conflict graphs. For example, our earlier work [22, 27] specified GC rules based on precedence information as to when old versions can be removed. However, in addition to being too complex to be amenable to practical implementation, these earlier works did not specify when these GC rules ought to be checked. Aydonat and Abdelrahman [3] propose to keep each version for as long as transactions that were active at the time the version was created exist, 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 [25].

Instead of multi-versioning, STMs can avoid aborts by reading uncommitted values and then having the reader block until the writer commits [28], or by using read-write

locks to block in case of concurrency [12, 2]. These approaches differ from SMV, where transactions never block and may always progress independently. Moreover, reads, which are invisible in SMV, must be visible in these "blocking" approaches. In addition, reading the values of uncommitted transactions might lead to cascading aborts.

Transactional mutex locks (TML) [10], have been shown to be very efficient for read-dominated workloads due to their simplicity and low overhead. Unlike SMV, TML do not allow concurrency between update transactions and thus do not exploit the parallelism in read-write or write-dominated workloads.

Another technique for reducing the number of aborts is timestamp extension [29, 14]. This mechanism requires maintaining a read-set and therefore is usually not used by read-only transactions. Timestamp extension is applicable for SMV's update transactions as well, hence this improvement is orthogonal to the multi-versioning approach presented in this paper.

3 Exponential Memory Growth

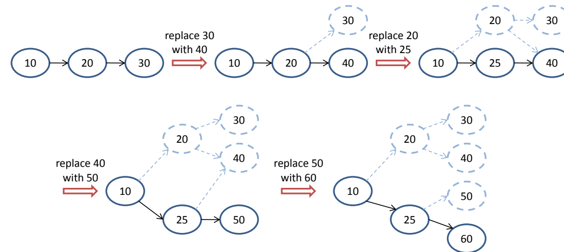


Fig. 1. Example demonstrating exponential memory growth even for an STM keeping only 2 versions of each object. A linked list causes a binary tree to be pinned in memory because previous node versions continue to keep references to already deleted nodes.

Before introducing SMV, we first describe an inherent memory consumption problem of algorithms keeping a constant number of object versions. A naïve assessment of the memory consumption of a k-versioned STM would probably estimate that it takes up to k times as much more memory as a single-versioned STM.

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

Consider, for example, a 2-versioned STM in the scenario depicted in Figure 1. The STM keeps a linked list of three nodes. When removing node 30 and inserting a new node 40 instead, node 30 is still kept as the previous version of 20.*next*. Next, when node 20 is replaced with node 25, node 30 is still pinned in memory, as it is referenced by node 20. After several additional node replacements, we see that there is a complete binary tree in memory, although only a linked list is used in the application.

More generally, with a k-versioned STM, a linked list of length n could lead to $\Omega(k^n)$ node versions being pinned in memory (though being still linear to the number of write operations). This demonstrates an inherent limitation of keeping a constant

number of versions per object. Our observation is confirmed by the empirical results shown in Section 5.5, where the algorithms keeping k versions cannot terminate in the runs with a limited heap size, while SMV does not suffer from any serious performance degradation.

4 SMV Algorithm

We present Selective Multi-Versioning, a new object-based STM. The data structures used by SMV are described in Section 4.1 and Section 4.2 depicts the algorithm.

4.1 Overview of Data Structures

SMV's main goal is to reduce aborts in workloads with read-only transactions, without introducing high space or computational overheads. SMV is based on the following design choices: 1) Read-only transactions do not affect the memory that can be accessed by other transactions. This property is important for performance in multi-core systems, as it avoids cache thrashing issues [13, 29]. 2) Read-only transactions always commit. A read-only transaction T_i observes a consistent snapshot corresponding to T_i's start time — when T_i reads object o_j, it finds the latest version of o_j that has been written before T_i's start. 3) Old object versions are removed once there are no live read-only transactions that can consistently read them. To achieve this with invisible reads, SMV relies on the omniscient GC mechanism available in managed memory systems.

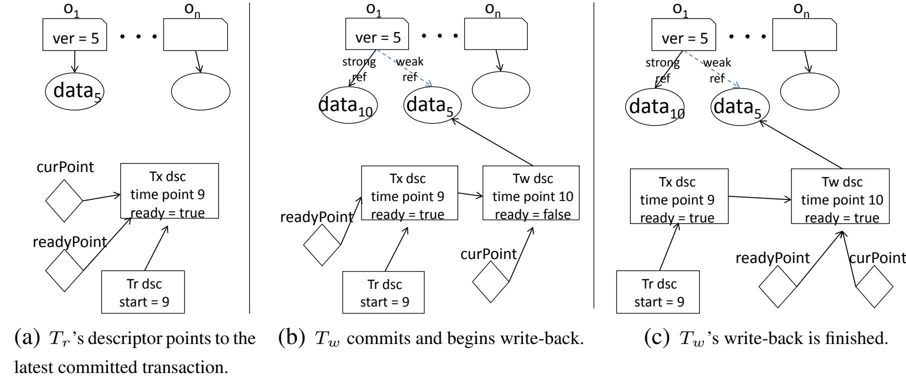
We now give a brief reminder of such a mechanism. An object can be reclaimed by the garbage collector once it becomes unreachable from the call stack or global variables. Reachability is a transitive closure over *strong* memory references: if a reachable object o_1 has a strong reference to o_2, then o_2 is reachable as well (strong references are the default ones in Java). In contrast, *weak references* [17] do not protect the referenced object from being GCed; an object referenced by weak references only is considered unreachable and may be removed.

As in other object-based STMs, transactional objects in SMV are accessed via *object handles*. An object handle includes a history of object values, where each value keeps a *versioned lock* [11] – data structure with a version number and a lock bit. In order to facilitate automatic garbage collection, object handles in SMV keep strong references only to the latest (current) versions of each object, and use weak references to point to other versions.

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

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

Version management is based on the idea that old object versions are pointed to by the descriptors of transactions that over-wrote these versions (see Figure 2). A committing transaction T_w includes in its transactional descriptor a strong reference to the



(a) T_r's descriptor points to the latest committed transaction. (b) T_w commits and begins write-back. (c) T_w's write-back is finished.

Fig. 2. Transactional descriptor of T_w references the over-written version of o_1 (data$_5$). This way, read-only transaction T_r keeps a reference chain to the versions that have been overwritten after T_r's start.

previous version of every object in its write set before diverting the respective object handle to the new version.

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

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

Figure 2 illustrates the commit of an update transaction T_w that writes to object o_1 (the use of *readyPoint* variable will be explained in Section 4.3). In this example, T_w and a read-only transaction T_r both start at time 9, and hence T_r references the transactional descriptor of time point 9. The previous update of o_1 was associated with version 5. When T_w commits, it inserts its transactional descriptor at the end of the time points list with value 10. T_w's descriptor references the previous value of o_1. This way, the algorithm creates a reference chain from T_r to the previous version of o_1 via T_w's descriptor, which ensures that the needed version will not be GCed as long as T_r is active.

4.2 Basic Algorithm

We now describe the SMV algorithm. For the sake of simplicity, we present the algorithm in this section using a global lock for treating concurrency on commit — in Section 4.3 we show how to remove this lock.

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

Handling update transactions. The protocol for update transaction T_i is depicted in Algorithm 1. The general idea is similar to the one used in TL2 [11]. An update trans-

Algorithm 1 SMV algorithm for update transaction T_i.

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

action T_i aborts if some object o_j read by T_i is over-written after T_i begins and before T_i commits. Upon starting, T_i saves the value of the latest time point in a local variable *startTime*, which holds the latest time at which an object in T_i's read-set is allowed to be over-written.

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

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

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

1. Lock the objects in the write set (line 20). Deadlocks can be detected using standard mechanisms (e.g., timeouts or Dreadlocks [24]), or may be avoided if acquired in the same order by every transaction.
2. Validate the read set (function *validateReadSet*).
3. Insert strong references to the over-written versions to T_i's descriptor (line 23). This way the algorithm guarantees that the over-written versions stay in the memory as long as T_i's descriptor is referenced by some read-only transaction.
4. Lock the time points list (line 24). Recall that this is a simplification; in Section 4.3 we show how to avoid such locking.
5. Set the commit time of T_i to one plus the value of the commit time of the descriptor referenced by *curPoint*.
6. Update and unlock the objects in the write set (lines 26–29). Set their new version numbers to the value of T_i.commitTime. Keep weak references to old versions.
7. Insert T_i's descriptor to the end of the time points list and unlock the list (line 30).

Handling read-only transactions. The pseudo-code for read-only transactions appears in Algorithm 2. Such transactions always commit without waiting for other transactions to invoke any operations. The general idea is to construct a consistent snapshot based

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 $\leq T_i$.startTP.commitTime) **then return** latestData
6: **return** the latest version *ver* in o_j.weak_references, s.t.
7: *ver*.version $\leq T_i$.startTP.commitTime

8: **Commit:**
9: T_i.startTP ← $\perp$

on the start time of T_i. At startup, T_i*.startTP* points to the latest installed transactional descriptor (line 2); we refer to the time value of startTP as T_i*'s start time*.

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

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

4.3 Allowing Concurrent Access to the Time Points List

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

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

To preserve consistency without locking the time points list, we add an additional boolean field *ready* to the descriptor's structure, which becomes *true* only after the committing transaction finishes updating all objects in its write-set. In addition to the global *curPoint* variable referencing the latest time point, we keep a global *readyPoint* variable, which references the latest time point in the *ready prefix* of the list (see Figure 2).

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

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

We have implemented two approaches to enforce real-time order: 1) An update transaction does not terminate until the ready point reaches its descriptor. A similar approach was previously used by RingSTM [33] and JVSTM [15]. 2) A new read-only transaction notes the time point of the latest terminated transaction and then waits until the *readyPoint* reaches this point before starting. Note that unlike the first alternative, read-only transactions in the second approach are not wait-free.

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

5 Implementation and Evaluation

5.1 Compared Algorithms

Our evaluation aims to check the aspect of keeping and garbage collecting multiple versions. Direct comparison was difficult because of different frameworks the algorithms are implemented in[1]. We implement the following algorithms:

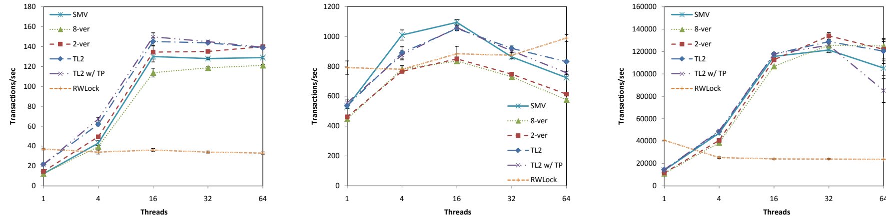
- **SMV**– The algorithm described in Section 4.
- **TL2**– Java implementation of TL2 [11] with a single central global version clock. We use a standard optimization of not keeping a read-set for read-only transactions. The code follows the style of TL2 implementation in Deuce framework [23].
- **TL2 with time points**– A variant of TL2, which implements the time points mechanism described in Section 4.1. This way, we check the influence of the use of time points on overall performance and separate it from the impact of multi-versioning techniques used in SMV.
- k**-versioned**– an STM based on a TL2-style's logic and code, in which each object keeps a constant k, number of versions (this approach resembles LSA [29]). Reads operate as in SMV, except that if no adequate version is found, the transaction aborts. Updates operate as in TL2.
- **Read-Write lock (RWLock)**– a simple global read-write lock. The lock is acquired at the beginning of an atomic section and is released at its end.

We use the Polite contention manager with exponential backoff [31] for all the algorithms: aborted transactions spin for a period of time proportional to 2^n, where n is the number of retries of the transaction.

5.2 Experiment Setup

All algorithms are implemented in Java. We use the following benchmarks for performance evaluation: 1) a red-black tree microbenchmark; 2) the Java version of STMBench7 [18]; and 3) Vacation, which is part of the STAMP [8] benchmark suite.

[1] DeuceSTM [23] framework comes with TL2 and LSA built-in, however, its LSA implementation is single-versioned.



(a) Throughput in red-black tree write-only workload. (b) Throughput in STMBench7's write-dominated workload. (c) Throughput in Vacation benchmark.

Fig. 3. In the absence of read-only transactions multi-versioning cannot be exploited. The overhead of SMV degrades throughput by up to 15%.

Red-black tree microbenchmark. The red-black tree supports insertion, deletion, query and range query operations. The initial size of the tree is 400000 nodes. It is checked both for read-dominated workloads (80/20 ratio of read-only to update operations) and for workloads with update operations only.

STMBench7. STMBench7 aims to simulate different behaviors of real-world programs by invoking both read-only and update transactions of different lengths over large data structures, typically graphs. Workload types differ in their ratio of read-only to update transactions: 90/10 for *read-dominated* workloads, 60/40 for *read-write* workloads, and 10/90 for *write-dominated* workloads. When running STMBench7 workloads, we bound the length of each benchmark by the number of transactions performed by each thread (2000 transactions per thread unless stated otherwise). We manually disabled long update traversals because they inherently eliminate any potential for scalability.

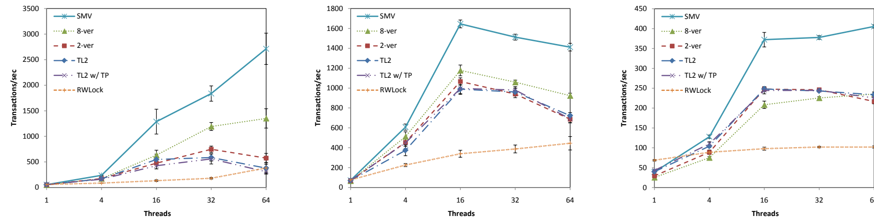
Vacation (Java port). Vacation [8], emulates a travel reservation system, which is implemented as a set of trees. In our experiments it is run with the standard parameters corresponding to `vacation-high++`. Note that STAMP benchmarks are not suitable for evaluating techniques that optimize read-only transactions, because these benchmarks do not have read-only transactions at all. We use Vacation as one exemplary STAMP application to evaluate SMV's overhead.

Setup. The benchmarks are run on a dedicated shared-memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor. The system runs Linux 2.6.22.5-31 with swap turned off. For all tests but those with limited memory, JVM is run with the `AggressiveHeap` flag on. Thread scheduling is left entirely to the OS. We run up to 64 threads on the 32 cores.

Our evaluation study is organized as follows: in Section 5.3, we show system performance measurements. Section 5.4 considers the latency and predictability of long read-only operations, and in Section 5.5, we analyze the memory demands of the algorithms.

5.3 Performance Measurements

SMV overhead. As we mentioned earlier, the use of multiple versions in our algorithm can be exploited by read-only transactions only. However, before evaluating the performance of SMV with read-only transactions, we first want to understand its behavior



(a) Throughput in STMBench7's read-dominated workload.

(b) Throughput in STMBench7's read-write workload.

(c) Throughput in the RBTree read-dominated workload.

Fig. 4. By reducing aborts of read-only transactions, SMV presents a substantially higher throughput than TL2 and the k-versioned STM. In read-dominated workloads, its throughput is $\times 7$ higher than that of TL2 and more than twice those of the k-versioned STM with $k = 2$ or $k = 8$. In read-write workloads its advantage decreases because of update transactions, but SMV still clearly outperforms its competitors.

in programs with update transactions only. In these programs, SMV can hardly be expected to outperform its single-versioned counterparts. For update transactions, SMV resembles the behavior of TL2, with the additional overhead of maintaining previous object versions. Thus, measuring throughput in programs without read-only transactions quantifies the cost of this additional overhead.

In Figure 3, we show throughput measurements for write-dominated benchmarks: Red-black tree (Figure 3(a)) and Vacation (Figure 3(c)) do not contain read-only transactions at all. The write-dominated STMBench7 workload shown in Figure 3(b) runs 90% of its operations as update transactions, and therefore the influence of read-only ones is negligible.

All compared STM algorithms show similar behavior in all three benchmarks. This emphasizes the fact that the algorithms take the same approach when executing update transactions and that they all have a common underlying code platform. The differences in the behavior of RWLock are explained by different contention levels of the benchmarks. While the contention level in Vacation remains moderate even for 64 threads, contention in the write-dominated STMBench7 is extremely high, so that RWLock outperforms the other alternatives.

Figure 3 demonstrates low overhead of SMV when the number of threads does not exceed 32; for 64 threads this overhead causes a 15% throughput drop. This is the cost we pay for maintaining multiple versions when these versions are not actually used.

Throughput. We next run workloads that include read-only transactions, in order to assess whether the overhead of SMV is offset by its smaller abort rate. In Figure 4 we depict throughput measurements of the algorithms in STMBench7's read-dominated and read-write workloads, as well as the throughput of the red-black tree. We see that in the read-dominated STMBench7 workload, SMV's throughput is seven times higher than that of TL2. Despite keeping as many as 8 versions, the k-versioned STM cannot keep up, and SMV outperforms it by more than twice.

What is the reason for 8 versions not being enough? In the full version of the paper [26] we show the following two results that explain this: First, the probability of

accessing an old object version is extremely small (less than 2.5% even for the second version). Therefore, keeping k versions for *each object* can be wasteful. Second, the amount of work lost because the kth version is absent, is surprisingly high even for large k values. Intuitively, this occurs since a transaction that needs to access the kth version of an object must have been running for a long time, and the price of aborting such a transaction is high. Hence, keeping previous versions is important despite the low frequency of accessing them; keeping a constant number of versions per object will typically not be enough for reducing the amount of wasted work.

We further note that SMV is scalable, and its advantage over a single-version STM becomes more pronounced as the number of threads rises. In the read-write workload, the number of read-only transactions that can use multiple versions decreases, and the throughput gain becomes 95% over TL2 and 52% over the 8-versioned STM.

We conclude that in the presence of read-only transactions the benefit of SMV significantly outweighs its overhead. In the full version of the paper [26] we explain this benefit by looking at the amount of work wasted due to aborts. We show that in the read-dominated workload, TL2 spends more than 80% of its time running aborted transactions! Interestingly, k-versioned STMs cannot fully alleviate this effect, reducing the amount of wasted time only to 36%. In contrast, SMV's waste does not rise above 3%.

It is possible to employ timestamp extension [29, 14] to reduce the amount of wasted work in both TL2 and SMV. However, this approach requires read-only transactions to maintain read-sets. The overhead of keeping a read-set is significant for long read-only transactions. We implemented timestamp extension in both TL2 and SMV, and our experiments showed that it did not improve the performance of either algorithm, although it did reduce the amount of wasted work. For space imitations, we omit these results.

5.4 Latency and Predictability of Long Read-Only Operations

In the previous section we concentrated on overall system performance without considering specific transactions. However, in real-life applications the completion time of individual operations is important as well. In this section we consider two examples: taking system snapshots of a running application and STMBench7's long traversals.

Taking a full-system snapshot is important in various fields: it is used in client-server finance applications to provide clients with consistent views of the state, for checkpointing in high-performance computing, for creating new replicas, for application monitoring and gathering statistics, etc. Predictability of the time it takes to complete the snapshot is important, both for program stability and for usability.

We first show the maximum time for completing a long read-only traversal, which is already built-in in STMBench7 (see Table 1(a)). As we can see from the table, this operation takes only several seconds when run without contention. However, when the number of threads increases, completing the traversal might take more than 100 seconds in TL2 and k-versioned STMs. Unlike those algorithms, SMV is less impacted by the level of contention and it always succeeds to complete the traversal in several seconds.

Next, we added the option of taking a system snapshot in Vacation. In addition to the original application threads, we run a special thread that repeatedly tries to take a snapshot. We are interested in the maximum time it takes to complete the snapshot operation. The results appear in Table 1(b). We see that neither TL2 nor the k-versioned

(a) Maximum time (sec) for completing a long read-only operation in STMBench7.

	Number of threads				
	1	4	8	16	32
TL2	1.3	21.6	68.5	103.6	358.5
SMV	1.3	1.4	2.4	3.6	11.9
2-versioned	1.3	4.1	22.9	45.2	204.5
8-versioned	1.3	6.8	10.6	22.2	79.4

(b) Maximum time (sec) to take a snapshot in Vacation benchmark.

	Number of threads				
	1	4	8	16	32
TL2	—	—	—	—	—
SMV	1.4	1.3	1.2	1.4	1.5
2-versioned	—	—	—	—	—
8-versioned	—	—	—	—	—

Table 1. Maximum time for completing long read-only operations. Long read-only traversals in STMBench7 can be hardly predictable for TL2 and k-versioned STMs: they might take hundreds of seconds under high loads. Vacation snapshot operation run by TL2 or k-versioned algorithms cannot terminate even when there is only a single application thread. SMV presents stable performance unaffected by the level of contention both for STMBench7 traversals and Vacation snapshots.

STM can successfully take a snapshot even when only a single application thread runs updates in parallel with the snapshot operation. Surprisingly, even 8 versions do not suffice to allow snapshots to complete, this is because within the one and a half seconds it takes the snapshot to complete some objects are overwritten more than 8 times.

On the other hand, the performance of SMV remains stable and unaffected by the number of application threads in the system. We conclude that SMV successfully keeps the needed versions. In Section 5.5, we show that it does so with smaller memory requirements than the k-versioned STM.

We would like to note that while taking a snapshot is also possible by pausing mutator threads, this approach is much less efficient, as it requires quiescence periods and thus reduces the overall throughput.

5.5 Memory Demands

One of the potential issues with multi-versioned STMs is their high memory consumption. In this section we compare memory demands of the different algorithms. To this end, we execute long-running write-dominated STMBench7 benchmarks (64 threads, each thread running 40000 operations) with different limitations on the Java memory heap. Such runs present a challenge for the multi-versioned STMs because of their high update rate and limited memory resources. As we recall from Section 5.3, multi-versioned STMs cannot outperform TL2 in a write-dominated workload. Hence, the goal of the current experiment is to study the impact of the limited memory availability on the algorithms' behaviors.

	Memory limit				
	2GB	4GB	8GB	12GB	16GB
TL2	606.89	631.56	630.3	674.96	647.17
SMV	450.12	543.04	563.74	595.78	602.01
2-versioned	—	515.32	532.7	550.61	533.01
4-versioned	—	—	—	—	281.98
8-versioned	—	—	—	—	—

Fig. 5. Throughput (txn/sec) in limited memory systems: k-versioned STMs do not succeed to complete the benchmark.

Figure 5 shows how the algorithms' throughput depends on the Java heap size. A "—" sign corresponds to runs in which the algorithm did not succeed to complete the benchmark due to a Java `OutOfMemoryException`. Notice that the 8-versioned STM is unable to successfully complete a run even given a 16GB Java heap size. Decreasing k to 4, and then 2, makes it possible to finish the runs under stricter constraints. However, none of the k-versioned STMs succeed under the limitation of 2GB. Unlike k-versioned STMs, SMV continues to function under

these constraints. Furthermore, SMV's throughput does not change drastically — the maximum decrease is 25% when Java heap size shrinks 8-fold.

The collapse of the k-versioned STM confirms the observation from Section 3, where we have illustrated that its memory consumption can become exponential rather than linear in the number of transactional objects.

6 Conclusions

Many real-world applications invoke a high rate of read-only transactions, including ones executing long traversals or obtaining atomic snapshots. For such workloads, multi-versioning is essential: it bears the promise of high performance, reduced abort rates, and less wasted work.

We presented Selective Multi-Versioning, a new STM that achieves high performance (high throughput, low and predictable latency, and little wasted work) in the presence of long read-only transactions. Despite keeping multiple versions, SMV can work well in memory constrained environments.

SMV keeps old object versions as long as they might be useful for some transaction to read. We do so while allowing read-only transactions to remain invisible by relying on automatic garbage collection to dispose of obsolete versions.

References

1. `http://www.azulsystems.com/blog/cliff-click/2008-05-27-clojure-stms-vs-locks`.
2. H. Attiya and E. Hillel. Brief announcement: Single-Version STMs can be Multi-Version Permissive. In *Proceedings of the 29th symposium on Principles of Distributed Computing*, 2010.
3. U. Aydonat and T. Abdelrahman. Serializability of transactions in software transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2008.
4. 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.
5. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
6. A. Bieniusa and T. Fuhrmann. Consistency in hindsight, a fully decentralized stm algorithm. In *IPDPS 2010: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.
7. J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.
8. C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.
9. N. Carvalho, J. Cachopo, L. Rodrigues, and A. Rito-Silva. Versioned transactional shared memory for the FenixEDU web application. In *Proceedings of the 2nd workshop on Dependable distributed data management*, pages 15–18, 2008.
10. L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional mutex locks. In *Euro-Par'10*.
11. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.

12. D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.
13. R. Ennals. Cache sensitive software transactional memory. Technical report.
14. P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08*, pages 237–246, 2008.
15. S. M. Fernandes and J. a. Cachopo. Lock-free and Scalable Multi-Version Software Transactional Memory. In *PPoPP '11*, pages 179–188, 2011.
16. K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003.
17. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, 2005.
18. R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the Second European Systems Conference*, 2007.
19. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC'03*, pages 92–101, 2003.
20. 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.
21. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
22. I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. In *SPAA'09*, pages 59–68, 2009.
23. G. Korland, N. Shavit, and P. Felber. Noninvasive Java concurrency with Deuce STM (poster). In *SYSTOR '09*, 2009. Further details at `http://www.deucestm.org/`.
24. 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.
25. J. Napper and L. Alvisi. Lock-free serializable transactions. Technical report, The University of Texas at Austin, 2005.
26. D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: Selective Multi-Versioning STM. Technical report, Technion, 2011.
27. D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in transactional memory. In *PODC'10*.
28. H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. *SIGPLAN Not.*, 44(4):163–172, 2009.
29. 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.
30. T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2006.
31. W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC'05*, pages 240–248, 2005.
32. 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.
33. M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08*, pages 275–284, 2008.