SMV: Selective Multi-Versioning STM

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

January 10, 2011

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 achieves much better performance 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.

*Department of Electrical Engineering, Technion, Haifa, Israel {dima39@tx,szaparka@t2,smalish@t2,idish@ee}.technion.ac.il

1 Introduction

Transactional memory [19, 31] is an increasingly popular paradigm for concurrent computing in multi-core architectures. Most transactional memory implementations today are software toolkits, or *STMs* for short. 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 [4]. 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 [20], or client-server applications where an STM infrastructure replaces a traditional DBMS approach (e.g., FenixEDU web application [11]). 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* [6] (i.e., keeping multiple versions per object), as a promising way to make program performance more predictable.

Indeed, by keeping multiple versions it is possible to assure that each read-only transaction successfully commits by reading a *consistent snapshot* [5] 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 [18, 15, 14, 12], which, as we show below, might waste as much as 80% of their time because of aborts in benchmarks with many read-only transactions.

Nevertheless, previously suggested multi-versioned STMs did not fully realize this potential. As we show in this paper, this happened mainly because of an inefficient management of old object versions (see Section 2). Instead of keeping a variable number of versions based on demand, multi-versioned STMs existing today keep a constant number of versions for each object [27, 9, 28]. As we show below, this approach does not provide enough of a performance benefit for read-only transactions, and, even worse, it causes 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* [14], 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 use 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 (in an unmanaged system, one would need to explicitly implement the GC threads). SMV ensures that old object versions become *garbage collectible (GCable)* 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[1] and study its behavior for a number of benchmarks (red-black tree microbenchmark, STMBench7 [17] and Vacation [10]). We compare SMV to a TL2-style single-versioned STM [12], to a k-versioned variant of the same algorithm, which keeps k versions per object similarly to LSA [27], and to a simple global read-write lock approach.

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 a TL2-style algorithm 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. In contrast, 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 first STM that can successfully execute long read-only transactions concurrently with frequent updates. We do so by introducing a new approach for maintaining multiple versions. 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 [12], 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.

Among multi-versioned STMs, the closest to SMV is LSA [27]. LSA, as well as its snapshot-isolation variation [28], 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.

Other previous suggestions for multi-versioned STMs [4, 24, 22, 7, 25] 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. Earlier work [22, 25] specified GC rules based on precedence information as to when old versions can be removed. However, these algorithms were too complex to be amenable to practical implementation, and did not specify when these GC rules ought to be checked. Aydonat and Abdelrahman [4] 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 [24]. 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.

[1] The code is publicly available in `http://tx.technion.ac.il/~dima39/publications.html`

The idea of keeping information as long is it might be needed by on-going transactions was recently used by Bronson et al. [8] for implementing concurrent collections. However, that paper deals with specific data structures rather than general purpose STMs.

Instead of multi-versioning, STMs can avoid aborts by reading uncommitted values and then having the reader block until the writer commits [26], or by using read-write locks to block in case of concurrency [13, 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.

3 Exponential Memory Growth

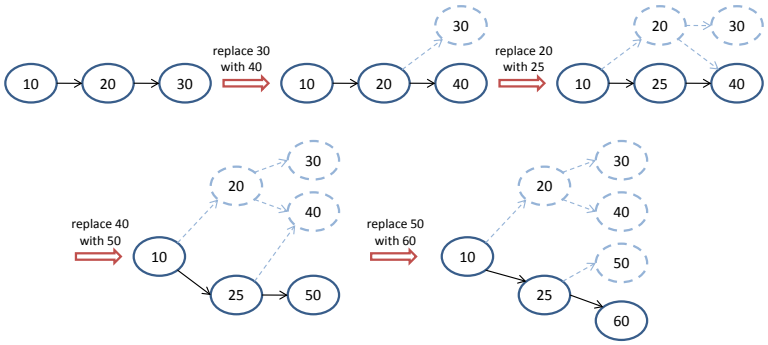


Figure 1: Example demonstrating exponential memory growth even for an STM keeping only 2 versions of each object. A linked list causes a binary tree to be pinned in memory because previous node versions continue to keep references to already deleted nodes.

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. This demonstrates an inherent limitation of any algorithm that keeps a constant number of versions for each object. SMV overcomes this limitation by keeping old object versions only as long as they might be needed by live read-only transactions. Our observation is confirmed by the empirical results shown in Section 5.6, 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 Selective Multi-Versioning Algorithm

We present Selective Multi-Versioning, a new object-based STM algorithm. Section 4.1 presents the principles underlying SMV's design. The data structures used by SMV are described in Section 4.2. Section 4.3 presents the basic idea of the algorithm, and Section 4.4 discusses some practical optimizations.

4.1 Design Principles

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:

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 [14, 27].

Multi-versioning for reads. Read-only transactions always commit. We achieve this by allowing read-only transaction T_i to observe 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.

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 ensure that unneeded data is GCable, i.e., once the version of the object cannot be read by any transaction, this version is not strongly referenced by any live memory object.

Global version clock. Like TL2 [12] and LSA [27], 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 [12, 29]. Such optimizations are orthogonal to our work, and are therefore beyond the scope of this paper. Perelman et al. [25] show that such global contention is essential for any multi-versioned STM that allows all read-only transactions to commit (more formally, such an STM cannot be disjoint access parallel [21]).

4.2 Overview of Data Structures

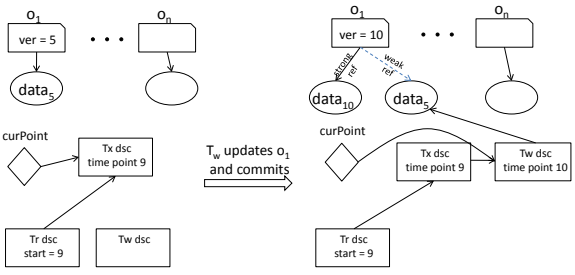


Figure 2: Transactional descriptor of T_w with time point 10 references the over-written version of o_1.

Before introducing the data structures of SMV, we give a brief reminder of the garbage collection mechanism in managed memory systems. 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* [16] 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* [12] – 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. However, unlike previous implementations of such clocks [12, 27], SMV's version clock consists of a list of transactional descriptors, rather than a scalar variable. 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 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. 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. 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 greater than T_r's start time. Hence, T_r reads the previous object version, whose version (5) is earlier than its start time (10).

4.3 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 4.4 we will remove this lock and show how to make commit operations lock-free.

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 [12] and LSA [27]. 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 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 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 34–36). The validation procedure checks that o_j is not locked and that its version is not greater than T_i.startTime. If the validation fails, the transaction is aborted.

A write operation (lines 10–13) is 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 adds it to T_i's local write set.

Algorithm 1 SMV algorithm for update transaction T_i.

```
1:  Upon Startup:
2:      T_i.startTime ← curPoint.commitTime
3:  Read o_j:
4:      if (o_j ∈ T_i.writeSet) 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)
8:      return data
9:  Write to o_j:
10:     if (o_j ∈ 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: Function validateReadSet() ▷ verify that none of the objects in the
    read-set has been over-written after being read by T_i
15:     foreach o_j ∈ T_i.readSet do:
16:         if (o_j.isLocked() ∨ o_j.version > T_i.startTime) then return
            false
17:     return true
18: Commit:
19:     foreach o_j ∈ T_i.writeSet do: o_j.lock()
20:     if ¬validateReadSet() then abort
        ▷ txn dsc should reference the over-written data
21:     foreach o_j ∈ T_i.writeSet do:
22:         T_i.prevVersions.put(⟨o_j, o_j.latest⟩)
23:     timeLock.lock()
24:     T_i.commitTime ← curPoint.commitTime + 1
        ▷ update and unlock the objects
25:     foreach ⟨o_j, data⟩ ∈ T_i.writeSet do:
26:         o_j.version ← T_i.commitTime
27:         o_j.weak_references ← o_j.weak_references ∪ o_j.latest
28:         o_j.latest ← data
29:         o_j.unlock()
30:     curPoint.next ← T_i
31:     curPoint ← T_i
32:     timeLock.unlock()
33: Function validateRead(Object o_j)
34:     if (o_j.isLocked()) then return false
35:     if (o_j.version > T_i.startTime) then return false ▷ o_j has been
        over-written
36:     return true
```

Commit (lines 19–32) consists of the following steps:

1. Lock the objects in the write set (line 19). Deadlocks can be detected using standard mechanisms (e.g., timeouts or Dreadlocks [23]), or may be avoided if acquired in the same order by every transaction.

2. Validate the read set (function *validateReadSet*, lines 15–17).

3. Insert strong references to the over-written versions to T_i's descriptor's *prevVersion* structure (line 22). In 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 23). Recall that this is a simplification; we show in Section 4.4 how we 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 25–29). 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 (lines 30–32).

Algorithm 2 SMV algorithm for read-only transaction T_i.

```
1:  Upon Startup:
2:      T_i.startTP ← curPoint
3:  Read o_j:
4:      latestData ← o_j.latest
5:      if (o_j.version ≤ T_i.startTP.commitTime) then return latestData
6:      return the latest version ver in o_j.weak_references, s.t.
7:          ver.version ≤ T_i.startTP.commitTime
8:  Commit:
9:      T_i.startTP ← ⊥
```

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 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.

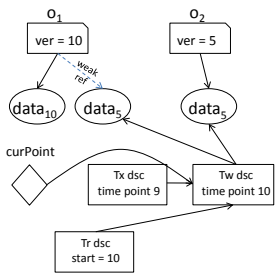
The commit procedure for read-only transactions merely removes the pointer to the starting time point, in order to make it GCable, and always commits.

4.4 Allowing Concurrent Access to the Time Points List

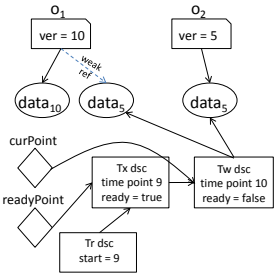
We show now how to avoid locking the time points list (lines 23, 32), so that update transactions with disjoint write-sets may commit concurrently. We first explain the reason for using the lock. In order to update the objects in the write-set, the updating transaction has to know the new version number to use. Choosing the version number and adding the descriptor to the list must be part of the same atomic action, so that two different transactions do not get the same number.

But if a transaction exposes its descriptor before it finishes updating the write-set, then some read-only transaction might observe an inconsistent state, as exemplified by the scenario depicted in Figure 3(a). Consider 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 $curPoint \rightarrow commitTime = 10$. 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, thus obtaining an inconsistent snapshot. 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.



(a) T_r's start time is 10, which causes inconsistent snapshot.



(b) T_r's start time is 9 according to the latest ready time point.

Figure 3: Transaction T_w updates o_1 and o_2 with version 10. Without a proper synchronization, transaction T_r might read a new version of o_1 and an old version of o_2.

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. When a new read-only transaction starts, its *startTP* variable references *readyPoint* (instead of *curPoint* as in the original algorithm). In Figure 3(b), a read-only transaction T_r has a start time of 9, because the new time point with value 10 is still not ready. Hence, T_r does not return the new value of o_1. Generally, the use of *readyPoint* guarantees that if a read-only transaction T_r 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 make sure we do not violate the real time order. That is, we should prevent a situation in which a transaction T_r that starts after T_w terminates has a start time value smaller than T_w's commit time, because in this case, T_r might not see the values written by T_w.

This situation might arise 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. To enforce real-time order, a new read-only transaction first notes the time point of the latest ready transaction. If *readyPoint* does not point to this transaction (because of a "gap" in ready transactions as explained above), it waits for the ready point to reach this point before starting.

5 Implementation and Evaluation

5.1 Compared Algorithms

Our evaluation aims to check the specific novel aspect introduced by SMV, that is to say, keeping and garbage collecting multiple versions. We compare SMV to the closest well-known algorithm, namely TL2 and a multiple-versioned variant thereof. We implement the following algorithms:

SMV – The algorithm described in Section 4.

TL2-style – A single-versioned STM mimicking the basic operation of TL2 [12] with a single central global version clock.

TL2 with time points – A variant of TL2, which implements the time points mechanism described in Section 4.2. This way, we check the influence of the use of time points on the 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 [27]). Reads operate as in SMV, except that if no adequate version is found, the transaction aborts. Update transactions have a behavior similar to that of 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 [30] 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

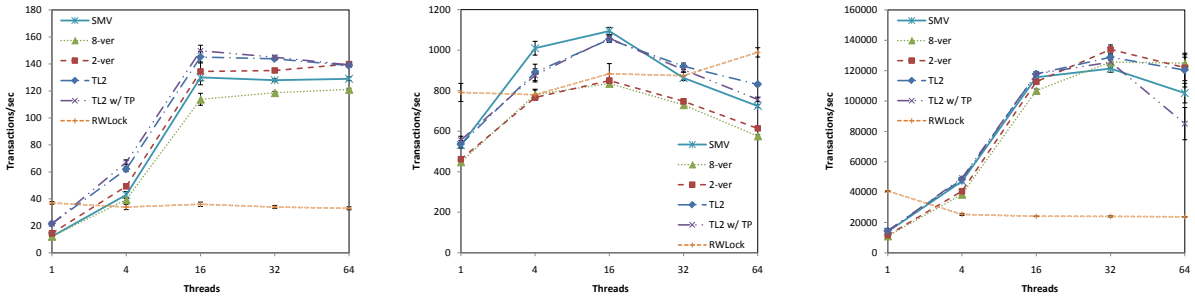
All algorithms are implemented in Java. We use the following benchmarks for performance evaluation: 1) a red-black tree microbenchmark; 2) the Java version of STMBench7 [17]; and 3) Vacation, which is part of the STAMP [10] benchmark suite.

Red-black tree microbenchmark. The red-black tree data structure supports insertion, deletion, query and range query operations. The initial size of the tree is 400000 nodes. Its behavior 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. Operation types for both read-only and update transactions include graph traversals of different lengths, structural modifications, and single access operations. 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 emulates a travel reservation system, which is implemented as a set of trees. It supports changing the size of the database, length of the benchmark, and other parameters that give us control over the level of contention among threads; in our experiments we use 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 one exemplary STAMP application (Vacation) to compare SMV's overhead with the overhead of a single-versioned STM in cases without read-only transactions. Other STAMP benchmarks do not add any additional insight into this question, and are therefore not included in the paper.



(a) Throughput in red-black tree write-only workload.

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

(c) Throughput in Vacation benchmark.

Figure 4: In the absence of read-only transactions multi-versioning cannot be exploited. The overhead of SMV degrades throughput by up to 15%.

Setup. In all our benchmarks, we defined transactional objects at the granularity of graph/data structure nodes. This provides a reasonable compromise between the cost of copy-on-write and the overhead of algorithmic bookkeeping. To support this, we re-implemented collections based on `java.util`.

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. In Section 5.5, we examine how many versions should be used, and in Section 5.6, we analyze the memory demands of the algorithms.

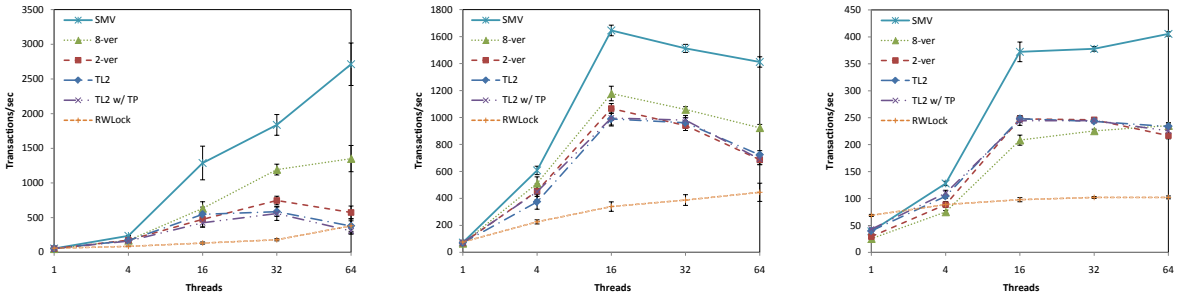
5.3 Performance Measurements

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 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 4, we show throughput measurements for write-dominated benchmarks: Red-black tree (Figure 4(a)) and Vacation (Figure 4(c)) do not contain read-only transactions at all. The write-dominated STMBench7 workload shown in Figure 4(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 4 demonstrates low overhead of SMV when the number of threads does not exceed 32; for 64



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

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

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

Figure 5: 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.

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 5 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. 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.

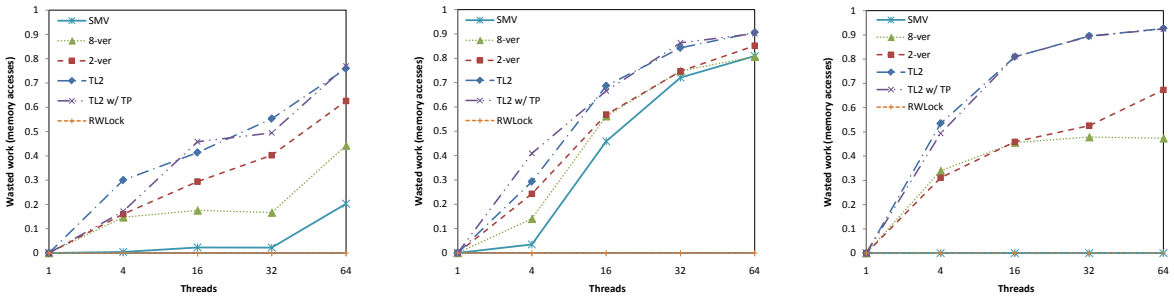
We conclude that in the presence of read-only transactions the benefit of SMV significantly outweighs its overhead. To explain this benefit, we next look at the amount of work wasted by the STMs due to aborts.

Amount of wasted work. As explained above, keeping multiple versions can potentially improve performance by decreasing the abort rate. However, looking at the abort rate is not enough: there is a substantial difference between a transaction that is aborted at the very beginning and a transaction aborted after running for a long time. Hence, what we want to measure now is the "amount of wasted work" done by transactions before they abort.

There is no strict way for defining wasted work. We therefore analyze two parameters: 1) number of accesses to transactional objects during aborted transactions; and 2) time spent by the program inside aborted transactions.

In Figure 6 we show the number of memory accesses performed by aborted transactions. Wasted memory accesses are undesirable mainly because of their disruptive influence on cache performance and their negative role in power consumption. In addition, they give a good intuition for the overall efficiency of the run.

We see that in the read-dominated workload of STMBench7 (Figure 6(a)), with 64 threads, more than 70% of the memory accesses in TL2 occur in aborted transactions. This occurs not only because of a high abort rate, but also because the probability for a read operation by a long read-only transaction to require an

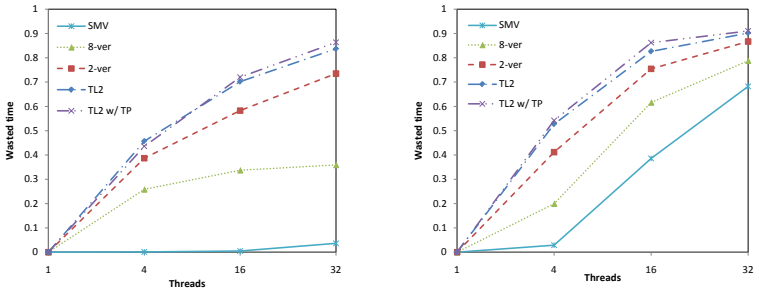


(a) Wasted memory accesses in STMBench7's read-dominated workload.

(b) Wasted memory accesses in STMBench7's read-write workload.

(c) Wasted memory accesses in the RBTree read-dominated workload.

Figure 6: Ratio of memory accesses in aborted transactions. In the read-dominated STMBench7 workload the use of selective multi-versioning reduces the amount of wasted work by a factor of 3, outperforming k-versioned STMs by more than 30%. In the RB tree read-dominated benchmark SMV succeeds to run most of the transactions, such that its wasted work is close to zero.



(a) Time spent in aborted transactions in STMBench7's read-dominated workload.

(b) Time spent in aborted transactions in STMBench7's read-write workload.

Figure 7: The share of time spent in aborted transactions for read-dominated and read-write workloads. In read-dominated workloads, SMV wastes less than 4% of time, while TL2 might waste more than 80% of the time inside aborted transactions. In read-write workloads, all STMs lose more time because of the larger share of update transactions. SMV is still better than TL2 and the k-versioned STMs by more than 20%.

old version increases over time. In SMV, wasted accesses amount to roughly 25% — a result that cannot be achieved by the k-versioned STMs either. Somewhat surprisingly, even 8 versions do not suffice, and circa 45% of the memory accesses are wasted. We will show why this occurs in Section 5.5.

In read-write workloads most of the wasted accesses occur due to aborts of update transactions, hence the differences between the algorithms are less significant — all the multi-versioned STMs show 10% less wasted accesses than TL2.

In Figure 7, we show the amount of time wasted on eventually aborted transactions. This approach approximates net CPU utilization and hence explains throughput results. We note that this approximation works well only if the number of threads is less than or equal to the number of available cores (otherwise, time measurements also count intervals in which the threads are suspended).

Here, the benefit of SMV is even more pronounced. We see 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 either, succeeding to reduce the amount of wasted time to 36% only. In contrast, SMV's wastage does not rise above 3%. In the read-write workload, the differences between the algorithms become less obvious, but the same tendency remains.

	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

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

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

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

Figure 8: 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.

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. As for any other operation, 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 8(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.

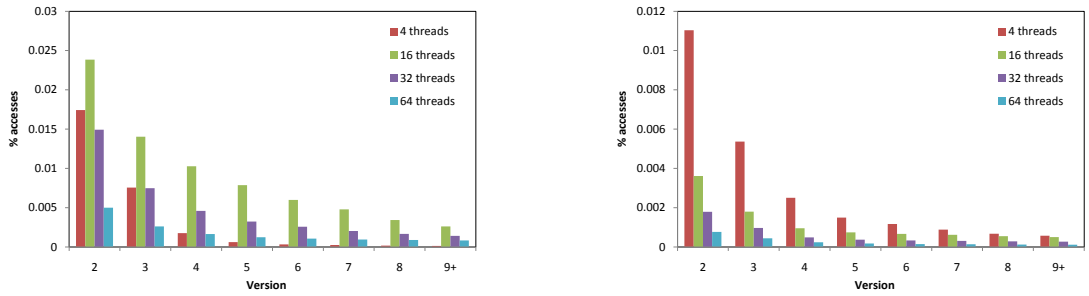
Next, we added the option of taking a system snapshot in Vacation. In addition to the original application threads, we run a special thread that repeatedly tries to take a snapshot. We are interested in the maximum time it takes to complete the snapshot operation. The results appear in Table 8(b). We see that neither TL2 nor the k-versioned 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 5.6, 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 could hardly be efficient. Pausing mutators is actually a privatization problem which has been shown to be inherently costly [3].

5.5 How Many Versions to Keep

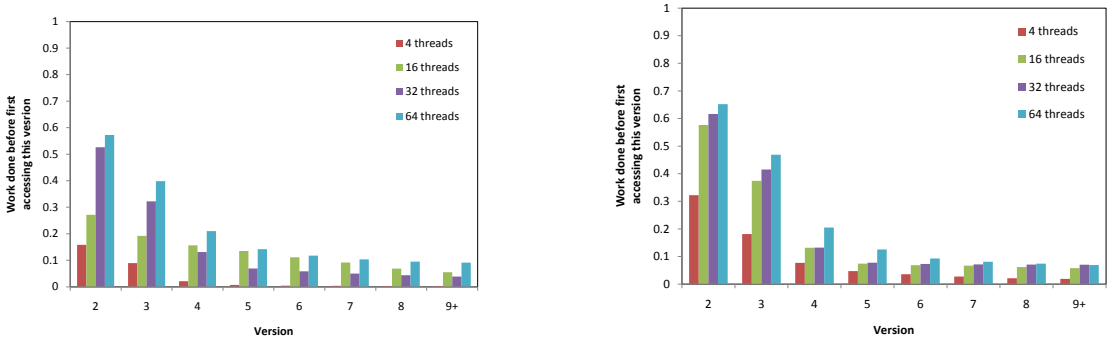
In the previous sections we have shown the benefits of keeping multiple versions for system throughput and operation latency in benchmarks with many read-only transactions. We now consider the following questions: how many versions do we need to keep in order to improve the performance?



(a) Version access histogram for STMBench7's read-dominated workload.

(b) Version access histogram for STMBench7's read-write workload.

Figure 9: Version access histograms for STMBench7. The 2^{nd} version is accessed less than 2.5% of the time, and it drops even further for older versions. The histogram emphasizes the fact that in most cases, keeping a single version of an object is enough.



(a) The share of memory accesses done by transactions before first accessing a k^{th} object version in STMBench7's read-dominated workload.

(b) The share of memory accesses done by transactions before first accessing a k^{th} object version in STMBench7's read-write workload.

Figure 10: The share of memory accesses done by transactions before first accessing a k^{th} object version. Surprisingly, despite extremely low access probabilities for version k in general (see Figure 9), the price of not keeping this version is high. For example, at least 60% of memory accesses have to be redone at least once if an STM keeps a single object version. Furthermore, the "heavy tail" in the graphs shows that keeping any constant number of versions per object does not prevent a high percentage of wasted work.

Histogram of version accesses. In Figure 9 we show a histogram of previous version accesses in STMBench7's read-dominated and read-write workloads. At any point in time, we number object versions in ascending order, starting from the latest one (the latest version is number 1, the one before it is number 2, etc.). For each number k, we count the percentage of accesses to version k out of the total number of memory accesses.

We see from the graph that the percentage of accesses to an arbitrary old object version is extremely small – it is less than 2.5% even for version 2. This observation intuitively suggests that keeping a constant number of versions per object is wasteful, because for most of the objects, a single version is enough most of the time.

But in the previous sections, we saw that even 8 versions do not suffice to allow long read-only transactions to complete, and even with this many versions, aborts lead to a substantial percentage of wasted work. To explain this apparent contradiction, we now examine how the number of versions can impact wasted work.

Wasted work lower bound. We now conduct the following experiment: while running SMV, for each version number k, we keep the number of memory accesses executed by transactions before they first access version k (if a transaction never accesses version k, the number is zero). In an STM that keeps $k-1$ versions per object, the transactions would then abort, and these memory accesses would need to be redone at least once. Thus, we obtain a lower bound on wasted memory accesses in a k-versioned STM.

Figure 10 shows the number of memory accesses done before first accessing some k^{th} object version. Surprisingly, this amount is relatively high: it starts from 60% for the second version and does not fall below 10% even for the 9^{th} version. According to the histogram in Figure 9, the likelihood of accessing these versions is extremely low. So how can we get such a high number of memory accesses? The insight is that if a transaction accesses the k^{th} version of an object, it means that this object has been updated at least $k-1$ times since this transaction began. This usually happens if a transaction has been already running for a long period of time and has already accessed a large number of transactional objects. The older the version, the less it is accessed, and thus the higher the amount of work executed before the access.

The direct conclusion from Figure 10 is that keeping previous versions is important despite the low frequency of accessing them. Furthermore, the "heavy tail" in the graphs indicates that keeping a constant number of versions per object will typically not be enough for reducing the amount of wasted work. This observation is in line with the results in the previous sections.

5.6 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	—	—	—	—	—

Table 1: Throughput (txn/sec) in limited memory systems: k-versioned STMs do not succeed to complete the benchmark, while SMV performs well even with a 2GB memory limitation.

Table 1 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 — there is a maximum decrease of 25% in throughput when Java heap size is shrinked 8-fold.

We explain the collapse of the k-versioned STM in Section 3, where we illustrate 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.

Nevertheless, previously suggested multi-versioned STMs did not fully deliver on these promises. In this paper, we have illustrated that the main reason for their limited success is due to keeping a *constant* number of versions for each object. We saw that because of the heavy-tailed nature of memory accesses, this constant number does not suffice for long traversals to complete. On the other hand, we illustrated that the memory consumption of this approach may grow exponentially with the amount of transactional data.

We presented Selective Multi-Versioning, a new STM that delivers on the promises of multi-versioning. It 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. More generally, we presented the idea of keeping old data as long as it might be useful for some processes in the future by having such potential future users of the data keep references that prevent the data's garbage collection. We believe that this approach can be the key to achieving good performance not only in STMs, but also in a range of concurrent data structures. SMV can be implemented in unmanaged memory systems by introducing special GC threads for periodical inspection of stale data — these threads can be fine-tuned to be more efficient that the Java alternative.

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] H. Attiya and E. Hillel. The cost of privatization. In *DISC'10*, pages 35–49, 2010.

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

[5] 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.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] 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.

[8] N. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional Predication: High-Performance Concurrent Sets and Maps for STM. In *Proceedings of the 29th symposium on Principles of Distributed Computing*, 2010.

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

[10] 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.

[11] 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.

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

[13] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

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

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

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, 2005.

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

[18] 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.

[19] 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.

[20] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[21] 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.

[22] I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. In *SPAA'09*, pages 59–68, 2009.

[23] 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.

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

[25] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in transactional memory. In *PODC'10*.

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

[27] 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.

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

[29] 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.

[30] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC'05*, pages 240–248, 2005.

[31] 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.