A Lazy Snapshot Algorithm with Eager Validation

Torvald Riegel[1] and Pascal Felber[2] and Christof Fetzer[1]

[1] Dresden University of Technology, Dresden, Germany,
`torvald.riegel@inf.tu-dresden.de`, `christof.fetzer@tu-dresden.de`
[2] University of Neuchâtel, Neuchâtel, Switzerland,
`pascal.felber@unine.ch`

Abstract. Most high-performance software transactional memories (STM) use optimistic invisible reads. Consequently, a transaction might have an inconsistent view of the objects it accesses unless the consistency of the view is validated whenever the view changes. Although all STMs usually detect inconsistencies at commit time, a transaction might never reach this point because an inconsistent view can provoke arbitrary behavior in the application (e.g., enter an infinite loop). In this paper, we formally introduce a lazy snapshot algorithm that verifies at each object access that the view observed by a transaction is consistent. Validating previously accessed objects is not necessary for that, however, it can be used on-demand to prolong the view's validity. We demonstrate both formally and by measurements that the performance of our approach is quite competitive by comparing other STMs with an STM that uses our algorithm.

1 Introduction

The recent move to multi-core processors has resulted in an increased research interest in *software transactional memory* (STM) [1]. STMs have been introduced as a mean to support lightweight transactions in concurrent applications. Transactions execute concurrently and those that fail to commit automatically roll back and restart their execution.

In STMs there is currently a tradeoff between consistency and performance. Several high-performance STM implementations [2–4] use optimistic reads in the sense that the set of objects read by a transaction might not be consistent. Consistency is only checked at commit time, i.e., commit *validates* the transaction. However, having an inconsistent view of the state of the objects during the transactions might, for example, result in infinite loops or the throwing of exceptions. These failures must then be detected and masked by the STM or the program's runtime environment, which is often both difficult and costly. Explicit, programmer-controlled validation is usually considered too difficult for programmers.

Example 1. Consider a linked list and two operations: (1) *search* iterates through the list until it finds a specific element while (2) *sort* re-orders the elements.

Consider that the list contains elements o_2 and o_1 in that order. Transaction T_1 sorts the list, which leads to re-ordering o_1 before o_2. If transaction T_2 iterates through the list but reads o_2 before the execution of T_1 (o_2 was the first object of the list) and o_1 after the sort operation has completed, it will experience a cycle and will loop forever.

Validation, on the other hand, can be costly (see Section 4) if it is performed in the obvious way, i.e., checking every object previously read. Typically, the validation overhead grows linearly with the number of objects a transaction has accessed so far. When one is forced to validate after each step, this could result in a validation overhead that grows quadratically with the number of objects accessed by a transaction.

In this paper, we introduce a lazy snapshot algorithm (LSA) that efficiently constructs an always consistent snapshot for transactions. Reads of a transaction are invisible to other transactions; the consistency of a transaction is verified by maintaining a validity interval for snapshots. In this way, an STM can efficiently verify during each object access that the snapshot of the objects that a transaction has seen so far is consistent.

We have built an object-based STM using LSA, to which we will refer as LSA-STM in what follows. It ensures linearizability [5] for read-only and update transactions. Our performance measurements demonstrate that the performance of LSA is very competitive with other STM implementations even when ensuring linearizability and always providing transactions with a consistent view.

In earlier work [6], we showed how to use LSA to build STMs that provide snapshot isolation [7]. The key idea of snapshot isolation (SI) is to provide each transaction T with a consistent snapshot of all objects at a given time. Writes of T occur atomically but possibly at a later time than that of the snapshot. This decoupling of the reads and the writes has the potential of increasing the transaction throughput but also gives application developers less ideal semantics than, say, STMs that guarantee linearizability [5]. However, SI always provides a transaction with a consistent view which avoids the programming anomalies that we described above.

In what follows, we first give a brief overview of the related work (Section 2). We then introduce LSA and demonstrate some of its properties (Section 3). Finally, we describe the results of our performance evaluation (Section 4) and conclude the paper (Section 5).

2 Related Work

Software Transaction Memory is not a new concept [1] but it recently attracted much attention because of the rise of multi-processor and multi-core systems. There are word-based [8] and object-based [9] STM implementations. The design of the latter, Herlihy's DSTM, is used by several current STM implementations. Most STM implementations are obstruction-free [10] and use contention managers [9] to deal with conflicts and ensure progress. Our LSA-STM is object-based and obstruction-free [10] and thus, uses some of DSTM's concepts. However, LSA-STM is a multi-version STM, whereas DSTM keeps at most two versions of an object but only uses the most recent version. We previously presented SI-STM [6], which uses LSA but provides, in addition to strict transactional consistency, support for snapshot isolation, which can increase the performance of suitable applications.

In DSTM and most of the high-performance STMs in general, reads by a transaction are invisible to other transactions: to ensure that consistent data is read, one must validate that all previously read objects have not been updated in the meantime. If reads are to be visible, transactions must add themselves to a list of readers at every transactional object they read from. Reader lists enable update transactions to detect conflicts with read transactions. However, the respective checks can be costly because readers on other CPUs update the list, which in turn increases the contention of the memory interconnect. Scherer and Scott [11, 12] investigated the trade-off between invisible and visible reads. They showed that visible reads perform much better in several benchmarks but, ultimately, the decision remains application-specific. Marathe *et al.* [13] present an STM implementation that adapts between eager and lazy acquisition of objects (i.e., at access or commit time) based on the execution of previous transactions. However, they do not explore the trade-off between visible and invisible reads but suggest that adaptation in this dimension could increase performance. Cole and Herlihy propose a snapshot access mode [14] that can be roughly described as application-controlled invisible reads for selected transactional objects with explicit validation by the application. Dice *et al.* show in [15], a recent improvement of earlier work [4], how to use a global version clock to improve the performance of low-overhead STMs. However, the validity of snapshots is fixed to the start time of a transaction and is not extended on demand. The only other multi-version STM that we are aware of is [16], although snapshots are not computed dynamically and conflict detection of update transactions only occurs at commit time. Furthermore, in that STM design, every commit operation, including the upgrade of transaction-private data to data accessible by other threads, synchronizes on a single global lock. No performance benchmark results were provided.

Read accesses in our LSA-STM are invisible to other transactions but do not require revalidation of previously read objects on every new read access. We show that our LSA facilitates inexpensive validation by maintaining a validity range in which a transaction is valid. In this way we get most of the benefits of visible and invisible reads but at a much lower cost.

3 Lazy Snapshot Algorithm

Before we can describe our lazy snapshot algorithm in Section 3.2, we first need to introduce some notations in Section 3.1. We show the correctness of LSA in Section 3.4.

3.1 Notations

A transactional memory consists of a set of shared objects $o_1, \ldots, o_n \in O$. Transactions are either *read-only*, i.e., they do not write to any object, or are *update* transactions, i.e., write to one or more objects.

Our transactional memory has a global counter, CT, that counts the number of update transactions that have committed so far. When an update transaction commits, it acquires a unique CT timestamp and creates a new version of the state of the transactional memory with this timestamp. Unlike in many other systems, this counter is not incremented when a read-only transaction commits. The goal is to improve the caching hit rate for this counter. We use CT as our time base, that is, all times given in the following are given with respect to this CT counter. For example, we denote the content of object o_i at (commit) time t, by o_i^t. Note that CT is a simple integer counter.

A transaction T accesses a finite set of objects $O_T \subseteq O$. We assume that objects are only accessed and modified within transactions. Hence, we can describe a history of an object with respect to the global commit time CT. The sequence $H_i = (v_{i,1}, \ldots, v_{i,j}, \ldots)$ denotes all the times at which updates to object o_i are committed by some update transactions. $v_{i,1}$ is the time the object is created. Sequence H_i is strictly monotonically increasing, i.e., $\forall j < |H_i| : v_{i,j} < v_{i,j+1}$. To simplify our equations, we assume that the first element of H_i, i.e., $v_{j,1}$ is 0 (all objects are created at time zero) and the last element is ∞ if $|H_i|$ is finite.

We say that the version j of object o_i ($j < |H_i|$) is valid from $v_{i,j}$ to $v_{i,j+1}-1$. We call this the *validity range* and denote this by

$$R_{i,j} := [v_{i,j}, v_{i,j+1} - 1].$$

A transaction T might not use the most recent version o_i^t of an object o_i when accessing the object the first time at t. Instead an older version might be used. Hence, for each object o_i in transaction O_T we denote the version of o_i by o_i^*, its version number by $v_{i,*}$, and its validity range by $R_{i,*}$ (o_i^*, $v_{i,*}$, and $R_{i,*}$ are specific to transaction T but, for simplicity, we do not make this explicit in our terminology).

By $\lfloor o_i^t \rfloor$ we denote the time of the most recent update of object o_i performed no later than time t, i.e., this update is still valid at global time t. We define $\lfloor o_i^t \rfloor$ as follows:

$$\lfloor o_i^t \rfloor := v_{i,j} \mid v_{i,j} \le t \wedge t < v_{i,j+1}.$$

By $\lceil o_i^t \rceil$ we denote the time until which the version of object o_i that is valid at time t remains valid:

$$\lceil o_i^t \rceil := v_{i,j+1} - 1 \mid v_{i,j} \le t \wedge v_{i,j+1} > t.$$

We define the *validity range* R_T of a transaction T to be the time range during which each of the objects accessed by T is valid. This is the intersection of the validity ranges of the individual versions accessed by a transaction:

$$R_T := \bigcap_{o_i \in O_T} R_{i,*}.$$

We say that the object versions accessed by transaction T, i.e., $\{o_i^* | \forall o_i \in O_T\}$ are a *consistent snapshot* if the validity range R_T of T is non-empty. Note that because of the intersection, the object versions contained in a consistent snapshot are always the most recent versions at any time $t \in R_T$.

An *update* transaction T writes to a subset of the objects in O_T. In our implementation, writing to an object always includes reading the object. We denote by $U_T \subseteq O_T$ the set of objects written to by T.

3.2 Snapshot Construction

The main idea of LSA (see Algorithm 1) is to construct consistent snapshots on the fly during the execution of a transaction and to—lazily—extend the validity range on demand. By this, we can reach two goals. First, transactions working on a consistent snapshot always read consistent data. Second, verifying that there is an overlap between the snapshot's validity range and the commit time of a transaction can ensure linearizability (if that is desired). Note that LSA-STM uses a lock-free implementation of LSA, whereas we assume sequential execution for the pseudocode in Algorithm 1 for simplicity. We will first describe the basic algorithm and then show that it is correct in Section 3.4.

Which objects are accessed by a transaction is determined during the execution of a transaction. The final R_T might not even be known at the commit time of the transaction. We therefore maintain a *preliminary validity range* R'_T. When a transaction T is started, we set R'_T to $[CT, \infty]$ (lines 2–3, $min(R'_T)$ and $max(R'_T)$ denote the lower and upper bound of R'_T). Note that R'_T will never assume a value smaller than the start time of T.

When accessing the most recent version of o_i, it is not yet known when this version will be replaced by a new version. We therefore approximate $R_{i,*}$ at time t by a *preliminary* range $R'_{i,*} = R_{i,*} \cap [0, t]$ and we set the new range to $R'_T \cap R'_{i,*}$ (lines 18–19). During the execution of a transaction, time will advance and thus the preliminary validity ranges might get longer. We can try to *extend* R'_T by recomputing $max(R'_T)$ (lines 14 and 29–37). Note that this is not required for correctness—it only increases the chance that a suitable object version is available.

Read accesses are optimistic and invisible to other transactions. LSA assumes that a system always keeps the most recent version of an object. In addition, LSA might also have access to some old versions (e.g., which have not yet been garbage collected) that can be used to increase the probability to create a consistent snapshot. When a transaction reads object o_i at time t, LSA tries to select the newest object version from H_i that still exists and that keeps the snapshot consistent, i.e., R'_T non-empty.

If the most recent version of o_i cannot be used because it was created after R'_T, we might still read some older version whose validity range overlaps R'_T. In that case, we simply set the new range to $R'_T \cap R'_{i,*}$ and we mark the transaction as "closed" to indicate that it cannot be extended anymore (lines 21–23). If no such version exists anymore, the transaction needs to be aborted. We omitted this in the simplified pseudocode of LSA.

By construction of R_T, LSA guarantees that a transaction started at time t has a snapshot that is valid at or after the transaction started, i.e., $\min(R'_T) \geq t$. Hence, a read-only transaction can commit iff it has used a consistent snapshot (i.e., R'_T is non-empty). The commit time CT is not increased when committing a read-only transaction because nothing has been modified.

3.3 Update transactions

Informally, an update transaction T performs the following steps when committing: (1) acquire a unique commit time CT_T from the CT time base (line 40), (2) validate T (lines 41–46), and (3) set T's state to committed if the validation was successful and to aborted otherwise (not shown). Update transactions can only commit if their validity range and their unique commit time (i.e., the global version that they are going to produce) overlap; in this case, the transaction is atomic. This is checked during the validation step (i.e., $(CT_T - 1) \in R'_T$). Therefore, accessed object versions must always be the most recent versions during the transaction and the validity range must be "open". If this is not possible, the transaction is marked as aborted (lines 25 and 35) because it would not be able to commit anyhow.

If it is possible to update a most recent version (i.e., R'_T remains non-empty), LSA atomically marks the object that it is writing (visible write, not shown in Algorithm 1). When another transaction T_1 tries to write o_i, T_1 will see the mark and detect a conflict. In that case, one of the transactions might need to wait or be aborted. This task is typically delegated to a *contention manager* [9], a configurable module whose role is to determine which transaction is allowed to progress upon conflict.

Setting the transaction's state atomically commits—or discards in case of an abort—all object versions written by the transaction and removes the write markers on all written objects (as in DSTM [9]).

If a transaction reads from the most recent version of an object that is write-marked by another transaction that has not yet committed, then the validity of the most recent version of this object ends at CT (the current time); the updating transaction cannot commit at a time earlier than $CT + 1$. This allows the STM to defer read-write conflicts to the commit time of the updating transaction, which minimizes the duration of such conflicts and lets reading transactions run unobstructed for a longer time.

For STMs that provide snapshot isolation [6], validation requires checking that only all object versions written by T are still valid at $CT_T - 1$. Since we use visible writes, this check is performed implicitly because write/write conflicts will result in one of the two conflicting transactions being aborted (or delayed) by the contention manager.

3.4 Linearizability

We sketch that transactions executed by an STM in the way outlined previously are linearizable.[3] To show this, we need to show that T takes effect atomically in between the start of T at time t and its commit time CT_T. We show that for

[3] We do not consider snapshot isolation (SI) here.

Algorithm 1 Lazy Snapshot Algorithm (LSA)

```
1: procedure Start($T$)                                  ▷ Initialize transaction attributes
2:     $T.min \leftarrow CT$                                          ▷ $= min(R'_T)$
3:     $T.max \leftarrow \infty$                                           ▷ $= max(R'_T)$
4:     $T.O \leftarrow \varnothing$                                ▷ Set of objects accessed by $T$
5:     $T.open \leftarrow$ true                                   ▷ Can $T$ still be extended?
6:     $T.update \leftarrow$ false                               ▷ Is $T$ an update transaction?
7: end procedure

8: procedure Open($T, o_i, m$)                   ▷ $T$ opens $o_i$ in mode $m$ (read or write)
9:     if m = write then
10:        $T.update \leftarrow$ true
11:    end if
12:    if $\lfloor o_i^{CT} \rfloor > T.max$ then                   ▷ Is most recent version too recent?
13:        if $T.update \wedge T.open$ then                                ▷ Try to extend?
14:            Extend($T$)
15:        end if
16:    end if
17:    if $\lfloor o_i^{CT} \rfloor \leq T.max$ then                  ▷ Can we use the latest version?
18:        $T.min \leftarrow \max(T.min, \lfloor o_i^{CT} \rfloor)$            ▷ Yes, $T$ remains open if it is still open
19:        $T.max \leftarrow \min(T.max, CT)$
20:    else if $\neg T.update \wedge VersionAvailable(o_i^{T.max})$ then
21:        $T.open \leftarrow$ false                          ▷ No, $T.max$ has reached its maximum
22:        $T.min \leftarrow \max(T.min, \lfloor o_i^{T.max} \rfloor)$
23:        $T.max \leftarrow \min(T.max, \lceil o_i^{T.max} \rceil)$
24:    else                                                     ▷ Cannot maintain snapshot
25:        Abort($T$)
26:    end if
27:    $T.O \leftarrow T.O \cup \{o_i\}$                                          ▷ Access object
28: end procedure

29: procedure Extend($T$)                          ▷ Try to extend the validity range of $T$
30:    $T.max \leftarrow CT$
31:    for all $o_i \in T.O$ do                          ▷ Recompute the whole validity range
32:        $T.max \leftarrow \min(T.max, \max(R'_{i,*}))$
33:    end for
34:    if $T.max < CT \wedge T.update$ then
35:        Abort($T$)                 ▷ Update transaction must access most recent versions
36:    end if
37: end procedure

38: procedure Commit($T$)                                       ▷ Try to commit transaction
39:    if $T.update$ then
40:        $CT_T \leftarrow (CT \leftarrow CT + 1)$                ▷ Acquire $T$'s unique commit time $CT_T$
41:        if $T.max < CT_T - 1$ then
42:            Extend($T$)          ▷ For update transactions, $CT_T$ and $R'_T$ must overlap
43:            if $T.max < CT_T - 1$ then
44:                Abort($T$)
45:            end if
46:        end if
47:    end if                                              ▷ $T$ can now be safely committed
48: end procedure
```

read-only transactions and then for update transactions. However, we introduce some lemmas first.

By the construction of LSA, it is guaranteed that the lower bound of R'_T is always greater than or equal to the start time of transaction T.

Lemma 1. *For any transaction T with a non-empty R'_T, it is guaranteed that $\min(R'_T)$ is greater or equal to the start time of T.*

Since the preliminary validity range of an object is always bounded by the current commit time, we know the following:

Lemma 2. *The preliminary time interval of transaction T at time t (after T has opened the first object) is at most t, i.e., $\max(R'_T) \leq t$.*

Theorem 1. *LSA guarantees that each read-only transaction T that started at time t_s and that commits between t_c and before $t_c + 1$ is linearizable.*

Proof. T can only commit if its preliminary validity range R'_T is non-empty when T commits. We know from lemma 1 and 2 that R'_T is a subset of $[t_s, t_c]$. This means that T is executed atomically between its start and before T terminates.

Theorem 2. *Each update transaction T that started at time t, that commits at time CT_T, and that satisfy $\max(R'_T) \geq CT_T - 1$, is linearizable.*

Proof. On commit, LSA checks that $(CT_T - 1) \in R'_T$ (lines 41–46 in Algorithm 1) and hence that all object versions that T accessed are still valid up to the time CT_T at which T commits its changes. Since each transaction has a unique commit time, no other transaction can commit at CT_T. This means that, logically, T reads all objects and commits all its updates atomically.

3.5 Extensions and global time

Validation is the bottleneck of STMs that use invisible reads. Whereas LSA can verify validity for any commit time by trying to extend the validity range to this time, other STMs usually verify the state at the time of validation. One might expect that LSA needs to perform extension frequently when there are concurrent updates that increase commit time fast. However, LSA is quite independent of the speed in which concurrent transactions increase time: if there are no concurrent updates to the objects that a transaction T accesses, the most recent object versions are accessible and do not change. Thus, no extension is required for obtaining a consistent read snapshot. If CT has been increased concurrently and T is an update transaction, one extension from R_T to $CT_T - 1$ is needed. If concurrent updates are not disjoint, LSA will require at most one extension per accessed object. However, this worst case should be very rare in practice because it requires certain update patterns; for example, once an already accessed object gets updated, R_T will be closed and there will be no further extensions.

Furthermore, the number of required accesses to the commit time is small. All transactions must read the current time when they are started. Update transactions must additionally acquire a unique commit time. Further accesses are not required for correctness. For example, if an update transactions needs to access a most recent version, then it can extend to a time at which this version was valid, but this time does not need to be the current time. Time information gathered from the accessed objects can be used instead of the current time.

Although we evaluate LSA only on smaller systems (see Section 4), we believe that the properties previously described as well as other mechanisms, such as using multiple commit times to partition data, make it suitable even for larger systems with higher communication costs.

4 Performance Evaluation

To evaluate the performance of our LSA-STM, we compared it with two other implementations. The first one follows the design of SXM by Herlihy *et al.* [18], an object-based STM with visible reads, with a few minor extensions. The second follows the design of Eager ASTM by Marathe *et al.* as described in [13]. Henceforth, we shall call these STM implementations *SXM* and *ASTM*. All three STMs are implemented using Java. Read operations in SXM are visible to other threads, whereas they are invisible in ASTM and LSA-STM. All STM implementations guarantee that all objects read in a transaction always represent a consistent view.

We executed all benchmarks on a system with four Xeon CPUs, hyperthreading enabled (resulting in eight logical CPUs).[4] Results were obtained by executing eight runs of 10 seconds for every tested configuration and computing the 12.5%-trimmed mean, i.e., the mean of the six median values. All STMs use the Karma [11] contention manager.

Overheads of validation and of a global commit time To highlight the differences between STM designs that use visible and invisible reads, Figure 1 shows the mean CPU time required for reading a single object in read-only transactions of different sizes. In this micro-benchmark, 8 threads read a given number of objects. All transactions read the same objects (with the exception of the SXM benchmark run with disjoint accesses) and there are no concurrent updates to these objects. The fixed overhead of a transaction gets negligible when the number of objects read during the transaction is high. SXM's visible reads have a higher overhead than LSA-STM's invisible reads. This overhead consists of the costs of the compare-and-swap (CAS) operation and possible

[4] 8GB of RAM, Sun's Java Virtual Machine version 1.5.0 with default configuration (server-mode, Parallel garbage collector), start and maximum heap size of 1GB. The machine has a light background load that we cannot control. Executing with more than 8 threads can give us a larger percentage of the CPUs and potentially a slight speedup when increasing beyond 8 threads.

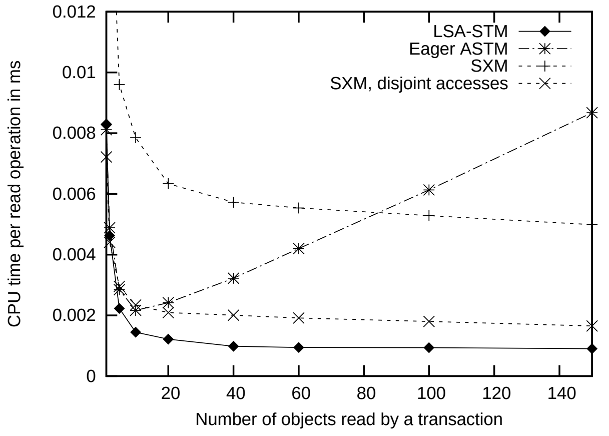

Fig. 1. LSA-STM read overhead in comparison to (Eager) ASTM and SXM.

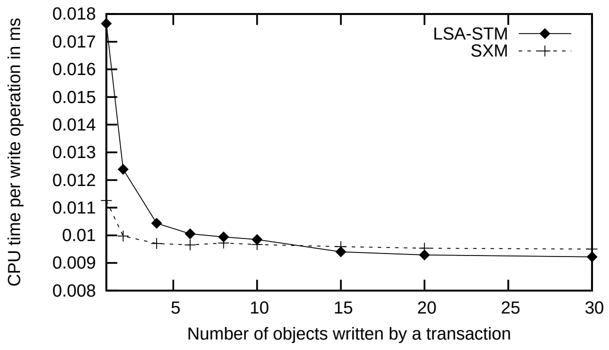

Fig. 2. LSA-STM write overhead.

cache misses and CAS failures if transactions on different CPUs add themselves to the reader list of the same object. ASTM has to guarantee the consistency of reads by validating all objects previously read in the transaction, which increases the overhead of read operations when transactions get large. Note that, although not shown here, ASTM transactions with only a single validate at the end of each transaction perform very similar to LSA-STM. However, for these transactions, consistency is not guaranteed during the execution of the transaction.

LSA-STM currently implements the global commit time *CT* as a shared integer counter. SXM and ASTM do not need such a counter that could become a source of contention if the rate of commits of update transactions is high. Figure 2 shows the overhead of write operations in LSA-STM by means of a micro-benchmark similar to the one used for Figure 1. However, now the 8 threads write to disjoint, thread-local objects. Acquiring timestamps induces a small overhead, which, however, gets negligible when at least 10 objects are written by a transaction. Furthermore, the overhead is smaller than the costs

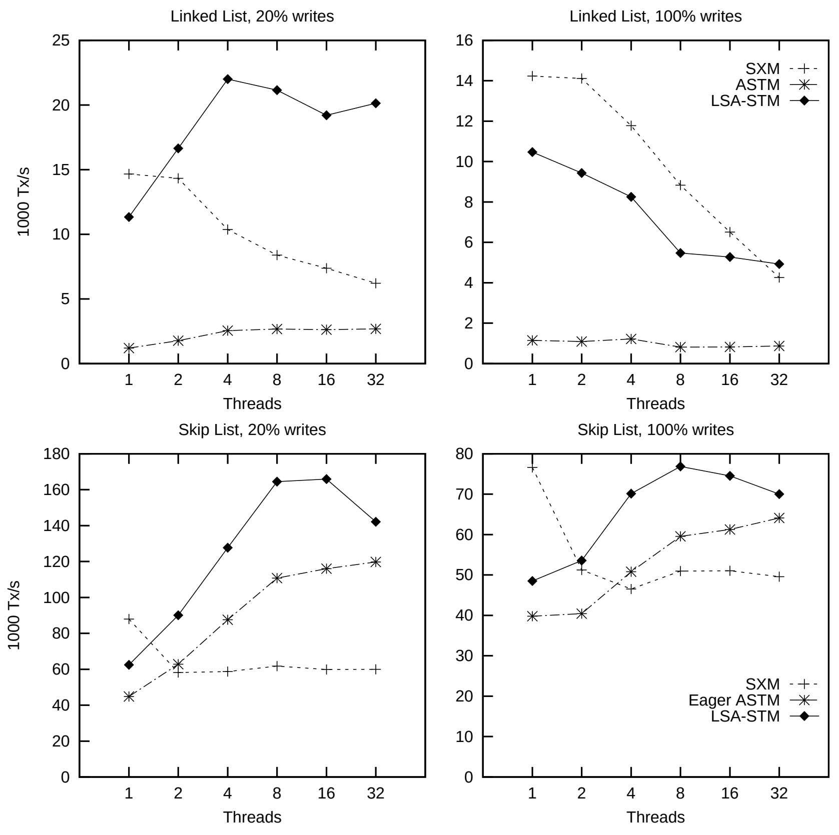

Fig. 3. Throughput results for the linked list (top) and skip list (bottom) benchmarks.

of a single write operation. Note, of course, that the results in Figure 1 and Figure 2 are hardware-specific.

Throughput Figure 3 shows throughput results for two micro-benchmarks that are often used to evaluate STM implementations, namely integer sets implemented via sorted linked lists and skip lists. Each benchmark consists of read transactions, which determine whether an element is in the set, and update transactions, which either add or remove an element. The sets consist of approximately 250 elements during the benchmark runs. We do not release objects early: although this would decreases the possibility of conflicts, it can mainly be used in cases in which the access path to an object is known.

We use the linked list to conveniently model transactions in which a modification depends on a larger amount of data that might be modified by other transactions.

For the skip list, STMs using invisible reads (ASTM and LSA-STM) show good scalability and outperform SXM, which suffers from the contention on the reader lists. However, the transactions in the linked list benchmark are quite large (the integer sets contain 250 elements) and ASTMs validation is expensive. LSA-STM, on the contrary, uses version information to compute the validity range much faster and scales up well to the number of available CPUs.

In all previous benchmarks, we always configured LSA-STM to keep eight old versions per object besides the most recent committed version. Keeping several versions can typically increase the commit rate but also adds memory overhead. In the following, we examine this problem further.

Object versions accessed In LSA-STM, references to object versions are stored in both a "locator" structure associated with transactional objects and an extra version array referenced by the locator. Like SXM and ASTM, LSA-STM is an object-based STM based on the design of DSTM [9] and thus uses locators to manage two object versions. However, whereas the other STMs use one of these versions as the working copy modified by updating transactions and the other version as a backup copy, LSA-STM can—because of LSA and validity range information—let reading transactions efficiently access the backup copy when an update is happening (it is the the most recent version) and when the working copy is committed (then the backup copy is the most recent old version). Thus, LSA-STM can provide one or two consistent versions of the object with the same space overhead. In the following, we denote accesses to the two versions (primary and backup) managed by the locator as accesses to version 0 or version 1, respectively. The extra version array stores references to older versions (the most recent version in the array has number 2).

Which object versions are accessed by a read-only transaction depends on how objects are concurrently updated by other transactions. To investigate this, we use a simple bank micro-benchmark, which consists of two transaction types: (1) transfers, i.e., a withdrawal from one account followed by a deposit on another account, and (2) computation of the aggregate balance of all accounts. Whereas the former transaction is small and contains 2 read/write accesses, the latter is a long transaction consisting only of read accesses (one per account and always in account order). There are 1,000 accounts and 8 threads perform either transfers or, with a 10% probability, balance computations.

Figure 4 shows access histograms of transactions computing the aggregate balance. There are three benchmark modes: (1) no hotspots, that is, update probability is equal for all accounts, (2) hotspots are encountered early during aggregate-balance computation, and (3) late hotspots. Hotspots are modeled by making the probability of updates to the first or last 50 accounts (accessed early or late, respectively) as probable as updates to other accounts.

In Figure 4, we can see how different update frequencies affect LSA's version selection (note the logarithmic scale). First, we observe that most accesses are performed to recent versions. When there are no hotspots, eight old versions are sufficient. When hotspots are encountered early during the runtime of a transaction, subsequent accesses will use even more recent object versions, because the relative update frequency of objects accessed late is smaller. In contrast, if

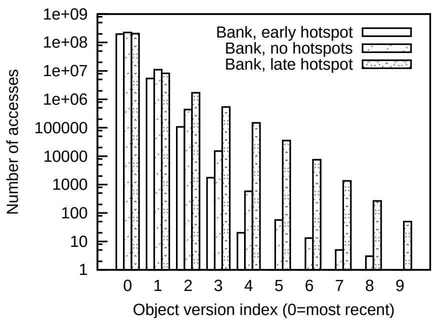

Fig. 4. Object versions accessed by balance computation transactions.

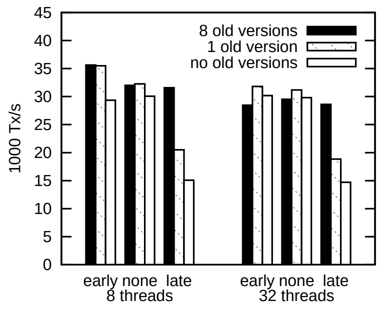

Fig. 5. Throughput results: versions kept per object, Bank with early, no, or late hotspots.

hotspots are encountered late, the transaction has to use older versions if one of the objects accessed early has been updated, which lets the validity range become closed. Thus, the probability that an old version will be useful increases with the size of the transactions and when hotspots happen late in their execution.

Throughput when keeping fewer old versions Figure 5 shows the throughput of the bank application when LSA-STM is configured to keep an extra version array with eight or one version, or no extra versions at all besides the versions 0 and 1 in the locator.

We can observe that keeping old versions can be beneficial, especially if hotspots are encountered late. However, the figure shows as well that throughput can increase if there are less versions. We will address this problem in future work.

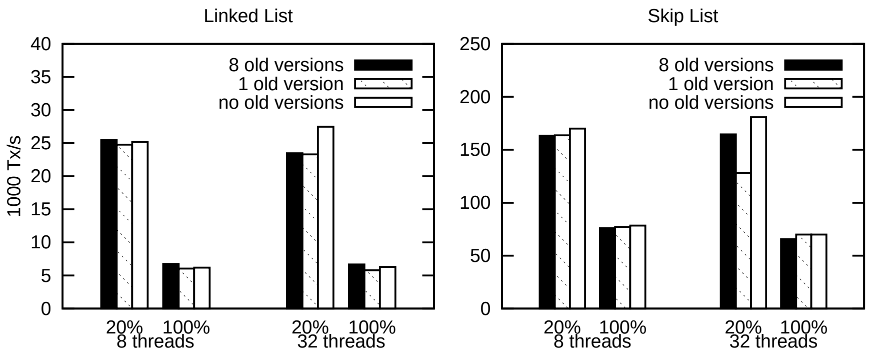

Fig. 6. Throughput results: versions kept per object, Integer Sets with 20% and 100% updates.

Figure 6 shows throughput results for the linked list and skip list benchmarks, with 20% or 100% update transactions and 8 or 32 threads. The skip list is mostly unaffected by the number of old versions available, or benefits only slightly from old versions. On the contrary, fewer versions increase the throughput of the linked list significantly. Using old versions will close the validity range, which is disadvantageous for transactions that become update transactions after reading a lot of objects (we assume that the type of a transaction is not known *a priori*). Nevertheless, we have focused our study on eight extra versions because this solution adapts well to various workloads. Adaptive version selection strategies might be able to increase the throughput further.

Even without any extra versions, and thus with the same space overhead as SXM or ASTM, LSA-STM is able to provide high throughput thanks to LSA and up to two consistent versions being available for reading transactions.

Validity range extensions The number of validity range extensions is very small in all of our benchmarks (not shown because of space limitations). The vast majority of transactions uses less than two to four extensions, depending on the transactions. In general, it can be observed that committed read-only transaction mostly use no or a single extension, whereas aborted read-only transactions often use at least one extension but seldom more. This is not surprising because high numbers of extensions can essentially be caused by scenarios in which (1) the update frequencies of objects accessed late during the transactions runtime are higher than those of objects accessed earlier, or (2) updates always happen immediately prior to accesses. For example, a single transition from non-hotspot to hotspot access patterns, as takes place in the benchmark, does not cause a lot of extensions. In the bank benchmark, read-only transactions almost never use extensions. In the linked list benchmark, however, almost all aborted read-only transactions use a single extension.

Update transactions behave as expected: the number of extensions for obtaining a snapshot is similar to that of read-only transaction, plus at most one extension per object update and at most one per commit. For example, less than 1% of the transfer transactions require an extension at all.

5 Conclusion

We introduced a lazy snapshot algorithm that creates consistent snapshots on the fly and can be used by STMs for read-only and update transactions. It is efficient both theoretically and practically. The idea is to maintain, for each transaction, a validity range based on global time. This range is sufficient to decide if a snapshot is consistent and transactions using this snapshot are linearizable. The snapshots are created in such a way that their freshness is maximized, e.g., a snapshot might actually become valid at a time after the snapshot is started. One issue is that the validity range of some of the objects is not known at the time the snapshot is created. This might require, in some cases, an *extension* of the preliminary validity range.

Acknowledgements. We thank the anonymous reviewers for their comments.

References

1. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of PODC. (1995)
2. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. (2006)
3. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing Memory Transactions. In: PLDI '06: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation. (2006)
4. Dice, D., Shavit, N.: What really makes transactions fast? In: TRANSACT. (2006)
5. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492
6. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: TRANSACT06. (2006)
7. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of SIGMOD. (1995) 1–10
8. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proceedings of OOPSLA. (2003) 388–402
9. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.: Software transactional memory for dynamic-sized data structures. In: Proceedings of PODC. (2003)
10. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems. (2003)
11. Scherer III, W.N., Scott, M.L.: Contention management in dynamic software transactional memory. In: Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs. (2004)
12. Scherer III, W., Scott, M.: Advanced contention management for dynamic software transactional memory. In: Proceedings of PODC. (2005)
13. Marathe, V.J., III, W.N.S., Scott, M.L.: Adaptive software transactional memory. In: Proceedings of DISC. (2005) 354–368
14. Cole, C., Herlihy, M.: Snapshots and software transactional memory. Science of Computer Programming (2005)
15. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: 20th International Symposium on Distributed Computing (DISC). (2006)
16. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. In: Proceedings of SCOOL. (2005)
17. Lu, S., Bernstein, A., Lewis, P.: Correct execution of transactions at different isolation levels. IEEE Transactions on Knowledge and Data Engineering **16**(9) (2004) 1070–1081
18. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Proceedings of DISC. (2005)