Time-based Software Transactional Memory

Pascal Felber, *Member, IEEE,* Christof Fetzer, *Member, IEEE,* Patrick Marlier, and Torvald Riegel

Abstract—Software transactional memory (STM) is a concurrency control mechanism that is widely considered to be easier to use by programmers than other mechanisms such as locking. The first generations of STMs have either relied on visible read designs, which simplify conflict detection while pessimistically ensuring a consistent view of shared data to the application, or optimistic invisible read designs that are significantly more efficient but require incremental validation to preserve consistency, at a cost that increases quadratically with the number of objects read in a transaction. Most of the recent designs now use a "time-based" (or "timestamp-based") approach to still benefit from the performance advantage of invisible reads without incurring the quadratic overhead of incremental validation. In this paper, we give an overview of the time-based STM approach and discuss its benefits and limitations. We formally introduce the first time-based STM algorithm, the Lazy Snapshot Algorithm (LSA). We study its semantics and the impact of its design parameters, notably multi-versioning and dynamic snapshot extension. We compare it against other classical designs and we demonstrate that its performance is highly competitive, both for obstruction-free and lock-based STM designs.

Index Terms—Transactional memory, transactions, concurrency, atomicity.

1 Introduction

The recent move to multi-core processors has resulted in an increased research interest in transactional memory, especially *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. Recent STM implementations prefer invisible over visible reads for efficiency reasons (see Section 2), and several of them [2], [3], [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., *validation* happens during commit only). 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.

Validation, on the other hand, can be costly (see Section 2) if it is performed in the obvious way, i.e., checking every object previously read for changes. 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 investigate a time-based approach to efficiently construct "snapshots" of the objects accessed by a transaction that remain consistent during the whole execution of the transaction. We call this algorithm the *Lazy Snapshot Algorithm* (LSA). It keeps the read operations of a transaction invisible to other transactions, and consistency is verified by maintaining a validity interval for snapshots on the basis of object modification timestamps obtained from a global time base. In this way, the STM can efficiently verify during each object access that the snapshot of previous accesses remains consistent.

We have built object-based and word-based STMs using the time-based algorithm.[1] They ensure linearizability [5] for read-only and update transactions. All transactions, i.e., even those that are eventually aborted, have always a consistent view. Our measurements demonstrate that the performance of time-based algorithms such as LSA is very competitive with other STM designs even when ensuring linearizability and always provides transactions with a consistent view. Many STM implementations have been transformed to use time-based algorithms after we first introduced LSA in [6]. Furthermore, time-based algorithms enable using multiple versions of objects to increase performance of read-only transactions.

In what follows, we first give necessary background information and a brief overview of the related work (Section 2). We then present LSA in Section 3 and discuss a few design issues in detail in Section 4. Two STM implementations, one object-based and one word-based, are presented in Section 5. Finally, we evaluate the performance of both implementations and LSA in general (Section 6) and conclude the paper (Section 7).

2 Background and Related Work

In early STM implementations, read operations are either *visible* or *invisible* to other threads. In the case of visible reads, transactions reading an object or memory location acquire ownership of this object by, for example, adding themselves to a list of readers at every object they read from. With invisible reads, transactions do not announce read operations to other transactions but ensure a consistent view of the data by validating the entire read-set when reading an object (which is also called *incremental validation*).

Visible reads enable writing transactions to detect conflicts with reading transactions. However, if several transactions read the same object, their performance will suffer from contention on the memory location used to announce read operations. The performance of validation-based invisible reads decreases if transactions read many objects because then the costs of

• *P. Felber and P. Marlier are with University of Neuchâtel, Switzerland.*

• *C. Fetzer and T. Riegel are with Dresden University of Technology, Germany.*

Manuscript received XXX, 2008; revised XXX, 2009.

1. Object-based and word-based STM designs detect data-access conflicts between concurrent transactions on the granularity of objects or memory regions, respectively.

incremental validation rise quickly. Scherer and Scott [7], [8] investigated the trade-off between validation-based invisible and visible reads. They showed that visible reads perform better in several benchmarks but, ultimately, the decision remains application-specific. Time-based STM algorithms improve upon this situation by enabling the use of invisible read operations without having to use incremental validation, thus avoiding the problematic overheads of both visible and early invisible read implementations.

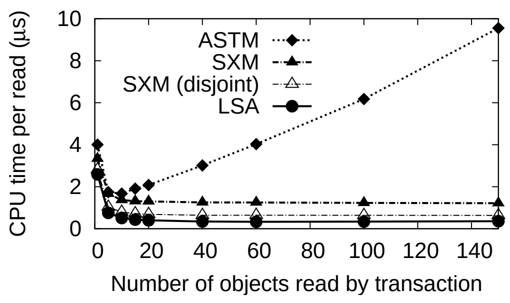


Fig. 1. Read overhead for differently-sized read-only transactions.

To highlight the differences between STM designs that use visible, validation-based invisible, and time-based invisible reads, Figure 1 shows the mean CPU time required for reading a single object in read-only transactions of different sizes on a Sun Fire T2000 (see Section 6 for details on the experimental setup). In this micro-benchmark, 8 threads read a given number of objects. All transactions read the same objects (except for the run labeled with "disjoint objects") and there are no concurrent updates to these objects. SXM and ASTM are STMs that we implemented according to the algorithm descriptions in [9] and [10]. SXM uses visible reads, whereas ASTM uses invisible reads with incremental validation. We compare them against LSA-STM, which uses time-based invisible reads (see Section 5). All three STMs are object-based Java designs that essentially differ in how they implement read operations and consistency of snapshots.

The fixed overhead of a transaction becomes 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 is due to the costs of the compare-and-swap (CAS) operation, as well as the possible cache misses and CAS failures when transactions on different CPUs update 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 larger. 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.

All STM designs that we consider in this paper guarantee linearizability for committed transactions [5]. In earlier work [6], we showed how to use LSA to build STMs that provide snapshot isolation [11]. The key idea of snapshot isolation (SI) is to provide each transaction with a consistent snapshot of all objects at a given time. Writes of this transaction 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. However, linearizability is a stronger guarantee and is easier to use by programmers (even though SI does always provide transactions with a consistent snapshot).

2.1 Related Work

Software Transaction Memory is not a new concept [1] but it only recently attracted much attention because of the rise of multi-processor and multi-core systems.

Two other STM designs based on a notion of time were published at roughly the same time as LSA. First, Dice *et al.* show in [12] how to use a global version clock to improve the performance of a low-overhead STM. However, in contrast to LSA, the validity of snapshots is fixed to the start time of a transaction and is not extended on demand. Second, Spear *et al.* [13] use a heuristic based on global counting of commits of update transactions to decrease the number of validations. Specifically, their STM still uses validation-based invisible reads but validates only if some update transaction committed since the last validation. This heuristic is not precise and can lead to many unnecessary validations, especially in large systems with many threads. In contrast, with LSA, a transaction can precisely detect whether a snapshot would still be consistent when reading an object, and validity extensions (which are similar to validations) only need to be performed when really necessary.

More recently, several STMs have been presented that use algorithms very similar to LSA [14], [15]. Riegel *et. al.* [16] show how to extend LSA so that imprecisely synchronized clocks can be used as a scalable implementation of the global commit time base. Zhang *et. al.* [17] present and evaluate several variations of the commit phase of time-based STM algorithms to avoid unnecessary updates of global time and unnecessary validations.

The LSA-based STM designs presented in this paper use various implementation techniques derived from other STMs, most importantly from DSTM [18] and TL2 [12].

3 The Lazy Snapshot Algorithm

We first informally explain the general principle of our algorithm and the way snapshots are constructed incrementally. We then give a formal definition of the algorithm and prove its correctness.

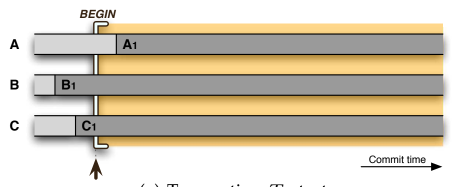
3.1 Principle of the Algorithm

Our LSA algorithm [19] handles transactional accesses to shared *objects*, which can designate either a complex data structure (as in object-oriented programming) or a single memory location. We will discuss and evaluate two concrete implementations of our algorithm in Java and C, which respectively support the two different kinds of shared objects.

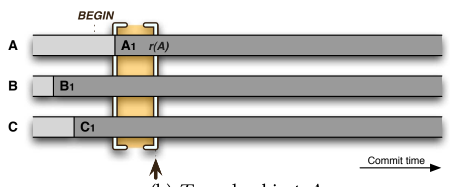
Every time an object is written by a committed transaction, a new version of this object is created and the previous one becomes obsolete. Our algorithm does not require to maintain old versions but can take advantage of them if they are available.

Our transactional memory uses a discrete logical global clock, designated by ***clock***.[2] When an update transaction commits, it acquires a unique timestamp from ***clock*** (informally, this represents progress by advancing the global time) and

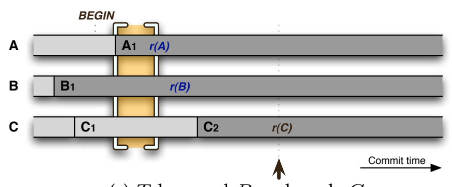
2. Note that the global clock can be replaced by more scalable alternatives, like approximately synchronized clocks as discussed in [16].



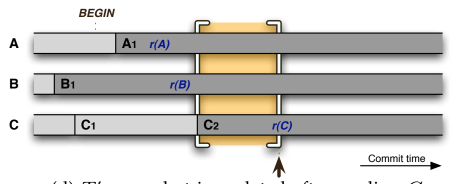
(a) Transaction T starts.



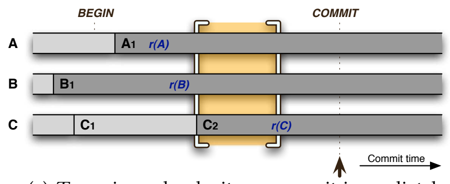
(b) T reads object A.



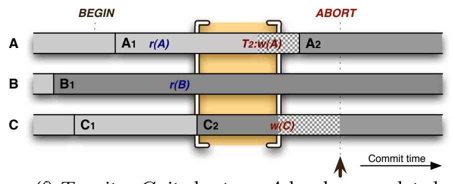
(c) T has read B and reads C.



(d) T's snapshot is updated after reading C.



(e) T can is read-only, it can commit immediately.



(f) T writes C, it aborts as A has been updated.

Fig. 2. Principle of the LSA-STM algorithm illustrated on a transaction T accessing three objects A, B, and C. Object versions are delimited by vertical lines and denoted respectively by A_i, B_i, and C_i $(i = 1, 2, \ldots)$. We represent the last committed version with a darker shade of grey. The thick arrow below the figures indicates the current time and the shaded region between large square brackets represents the transaction snapshot.

associates it with the objects it has written. That is, every shared object in the system has a timestamp that indicates the time from which its current version is valid, as well as an optional set of older versions with associated timestamps. The latest version of an object remains valid until it is overwritten by a committed transaction.

Every transaction maintains a *snapshot* that corresponds to a range of valid linearization points. The transaction can only commit if its snapshot is non-empty at completion time. Initially, the snapshot of a transaction is $[start, \infty]$, where $start$ is the value of ***clock*** at the time the transaction starts (see Figure 2(a)).

When a transaction reads an object, it must pick a version whose "validity range" (i.e., the period during which it is valid, see Section 3.2) intersects with the transaction's snapshot. The bounds of the snapshot are adjusted to the intersection. When reading the latest version of an object—the usual case—the upper bound is capped by the current value of the clock (see Figure 2(b) with version 1 of object A being read).

If the latest version of an object read by a transaction has a validity range that starts after the upper bound of the transaction's snapshot (see Figure 2(c) with object C being read), the transaction can either read an old version with a validity range that overlaps the snapshot, or attempt to *extend the snapshot*. An extension consists of trying to move the upper bound to some later point in time no higher than—but typically equal to—the current value of the clock. To that end, the transaction must verify that the versions of all the objects previously accessed by the transaction are still valid. If the extension succeeds, the transaction can read the latest version of the object and adjust the snapshot accordingly (see Figure 2(d) with version 2 of object C being read). Otherwise, if the transaction cannot read a valid version of the object while maintaining a non-empty snapshot (more precisely, a snapshot with a non-empty validity range), it aborts.

A transaction can only commit if it has a non-empty snapshot and a commit time that falls within the bounds of that snapshot. For a read-only transaction, as long as the snapshot is not empty, any point within the snapshot is a possible linearization point and, hence, a valid commit time. Therefore, such transactions can commit immediately (see Figure 2(e)).

Committing update transactions is slightly more complicated. In LSA, writes are visible, i.e., a transaction can determine whether an object is being written by another transaction. When an update transaction commits, it writes new versions of each updated object timestamped by the commit time of the transaction. Consider the example in Figure 2(f) where transaction T reads objects A and B before writing C. At commit time, transaction T must acquire a new, unique timestamp from the global clock that will be associated with the new version of C being written. Then, it must *validate* that all objects previously accessed are still valid at commit time, which corresponds to the linearization point of the transaction. In our example, another transaction has written a new version of object A, i.e., the version read by T is not valid anymore. Therefore, the transaction must abort.

We now describe the algorithm more precisely in the rest of this section.

3.2 Notations

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

We designate the discrete logical global time base of LSA by ***clock***. It can be implemented using a simple shared integer counter that is incremented *atomically* by update transactions to acquire a unique commit timestamp.[3]

A transaction T accesses a finite set of objects $\mathcal{O}_T \subseteq \mathcal{O}$. Each object o traverses a series of versions $o_1, o_2, \ldots, o_i$. The transactional memory may—but does not need to—keep multiple versions of an object at a given time; only the latest version is necessary.

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 time base ***clock***. We denote by $\lfloor o_i \rfloor$ the time when version i of object o has been written, and by $\lceil o_i \rceil$ the last time before the next version is written. We call the interval between these two bounds the "validity range" of the object version and we denote it simply by $[o_i]$. If o_i is

3. Atomic increment is achieved by hardware instructions like "increment-and-fetch" or "compare-and-swap" available on most modern processors.

the latest version of object o, then $\lceil o_i \rceil$ is undefined (because we do not know until when o_i will be valid), otherwise $\lceil o_i \rceil = \lfloor o_{i+1} \rfloor - 1$. For convenience, we denote by $o_\star$ the most recent version of object o.

The sequence $\mathcal{H}(o) = (\lfloor o_1 \rfloor, \ldots, \lfloor o_i \rfloor, \ldots)$ denotes all the times at which updates to object o are committed by some update transactions. $\lfloor o_1 \rfloor$ is the time when the object was created. Sequence $\mathcal{H}_i$ is strictly monotonically increasing, i.e., $\forall o_i \neq o_\star : \lfloor o_i \rfloor < \lfloor o_{i+1} \rfloor$.

Each transaction T maintains a read set $T.\mathcal{R}$ and a write set $T.\mathcal{W}$ that keep track of the object versions read and written by the transaction, respectively. To simplify the presentation, we assume in the pseudo-code that an object is accessed only once by a transaction (it is either read or written). We will explain in the description of the algorithm how multiple accesses by the same transaction are dealt with.

A transaction T incrementally constructs a snapshot of objects versions and keeps track of the validity ranges of these objects. To that end, T maintains the known bounds on the validity range $T.\mathcal{S}$ of the snapshot. These bounds, denoted by $\lfloor T.\mathcal{S} \rfloor$ and $\lceil T.\mathcal{S} \rceil$, are computed as the intersection of the validity ranges of the objects accessed by the transaction. We say that the snapshot is *consistent* if its bounds correspond to a non-empty range. Note that, by construction, the object versions contained in a consistent snapshot are always the most recent versions at any time $t \in T.\mathcal{S}$.

3.3 Snapshot Construction

The lazy snapshot algorithm is presented in Algorithm 1. A transaction completes successfully if it executes the algorithm until commit without encountering a call to ABORT (in which case it immediately terminates). Note that the pseudo-code does neither show how mutual exclusion is achieved nor how objects are atomically updated in memory. This will be discussed in Section 5 where we present two instantiations of LSA: obstruction-free and lock-based.

The main idea of the algorithm is to construct consistent snapshots on the fly during the execution of a transaction and to extend the validity range on demand (lazily). 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. We first describe the basic algorithm and then prove its correctness in Section 3.6.

The objects accessed by a transaction T are only discovered during its execution, i.e., the snapshot cannot be constructed beforehand. The final value of $T.\mathcal{S}$ might not even be known at the commit time of the transaction. We therefore maintain a *preliminary validity range* in $T.\mathcal{S}$ that represents the known bounds. When the transaction is started, we set $T.\mathcal{S}$ to $[\textbf{\textit{clock}}, \infty]$ (line 3). Note that $T.\mathcal{S}$ will never hold values smaller than the start time of T.

When accessing (i.e., reading or writing) the most recent version $o_\star$ of object o, it is not yet known when this version will be replaced by a new version. We therefore conservatively approximate the upper bound of its validity range by the current time t and we set the new snapshot range to $T.\mathcal{S} \cap [\lfloor o_\star \rfloor, t]$ (lines 10 and 21). During the execution of a transaction, time will advance and thus the preliminary validity ranges might get longer. We can try to "extend" $T.\mathcal{S}$ by recomputing its upper bound (lines 8, 19, 25–28). Note that this is not required for correctness—it only increases the chance that a suitable object version is available.

Algorithm 1: Lazy Snapshot Algorithm (LSA) for transaction T

```
1  clock ← 0
   // Start transaction
2  function START($T$)
3      $T.\mathcal{S} \leftarrow [\textbf{clock}, \infty]$                          // Snapshot bounds
4      $T.\mathcal{R} \leftarrow \varnothing$                                     // Read set
5      $T.\mathcal{W} \leftarrow \varnothing$                                     // Write set
   // Read a shared object
6  function READ($T$, $o$)
7      if $\lfloor o_\star \rfloor > \lceil T.\mathcal{S} \rceil$ then                // Is latest version too recent?
8          EXTEND($T$, clock)                                 // Try to extend
9      if $\lfloor o_\star \rfloor \leq \lceil T.\mathcal{S} \rceil$ then                // Can use latest version?
10         $T.\mathcal{S} \leftarrow [\max(\lfloor T.\mathcal{S} \rfloor, \lfloor o_\star \rfloor), \min(\lceil T.\mathcal{S} \rceil, \textbf{clock})]$   // Yes: use latest
11         $T.\mathcal{R} \leftarrow T.\mathcal{R} \cup \{o_\star\}$
12     else if $T.\mathcal{W} = \varnothing \wedge (\exists o_i : \lfloor o_i \rfloor \leq \lceil T.\mathcal{S} \rceil \wedge \lceil o_i \rceil \geq \lfloor T.\mathcal{S} \rfloor)$ then
13         $T.\mathcal{S} \leftarrow [\max(\lfloor T.\mathcal{S} \rfloor, \lfloor o_i \rfloor), \min(\lceil T.\mathcal{S} \rceil, \lceil o_i \rceil)]$   // No: use older
14         $T.\mathcal{R} \leftarrow T.\mathcal{R} \cup \{o_i\}$
15     else
16         ABORT($T$)                                     // Cannot find valid version: abort
   // Write a shared object
17 function WRITE($T$, $o$)
18     if $\lfloor o_\star \rfloor > \lceil T.\mathcal{S} \rceil$ then                // Is latest version too recent?
19         EXTEND($T$, clock)                                 // Try to extend
20     if $\lfloor o_\star \rfloor \leq \lceil T.\mathcal{S} \rceil$ then                // Can use latest version?
21         $T.\mathcal{S} \leftarrow [\max(\lfloor T.\mathcal{S} \rfloor, \lfloor o_\star \rfloor), \min(\lceil T.\mathcal{S} \rceil, \textbf{clock})]$   // Yes
22         $T.\mathcal{W} \leftarrow T.\mathcal{W} \cup \{o_\star\}$
23     else
24         ABORT($T$)                                     // Cannot find valid version: abort
   // Try to extend the snapshot
25 function EXTEND($T$, $t$)
26     $\lceil T.\mathcal{S} \rceil \leftarrow t$
27     foreach $o_i \in T.\mathcal{R} \cup T.\mathcal{W}$ do
28         $\lceil T.\mathcal{S} \rceil \leftarrow \min(\lceil T.\mathcal{S} \rceil, \lceil o_i \rceil)$
   // Try to commit the transaction
29 function COMMIT($T$)
30     if $T.\mathcal{W} \neq \varnothing$ then
31         $t_c \leftarrow (\textbf{clock} \leftarrow \textbf{clock} + 1)$    // Unique timestamp (atomic increment)
32         if $\lceil T.\mathcal{S} \rceil < t_c - 1$ then
33             EXTEND($T$, $t_c - 1$)                           // Try to extend
34             if $\lceil T.\mathcal{S} \rceil < t_c - 1$ then
35                 ABORT($T$)                             // Inconsistent snapshot: abort
36         foreach $o_i \in T.\mathcal{W}$ do                    // Atomically commit updates
37             $o_{i\star} \leftarrow o_i$                          // Write new version of shared object
38             $\lfloor o_{i\star} \rfloor \leftarrow t_c$                        // Validity starts at commit time
```

3.4 Read Accesses and Read-only Transactions

Read accesses in LSA are optimistic and invisible to other transactions. The algorithm assumes that the underlying STM always keeps the most recent version of an object. In addition, we might also have access to some older versions (e.g., objects that have not yet been garbage collected) that can be used to increase the probability of obtaining a consistent snapshot. When a transaction reads object o at time t, it first tries to select the most recent object version $o_\star$ (lines 9–11). If that version cannot be used because it was created after $T.\mathcal{S}$, we might still read some older version $o_i \in \mathcal{H}(o)$ whose validity range overlaps $T.\mathcal{S}$ and, hence, keeps the snapshot consistent (lines 12–14). In that case, we simply set the new range to $T.\mathcal{S} \cap [o_i]$. As a simple optimization (not shown in the code), we can mark the transaction as "closed" to indicate that it cannot be extended anymore. If there are multiple versions to chose from, we select the most recent one. If no such version exists, the transaction needs to be aborted (line 16).

If an object previously accessed by the current transaction is read, the same version must be returned to preserve consistence even if a new version has been committed in the meantime; otherwise the snapshot would contain multiple versions of the same object with non-overlapping validity ranges and the transaction would obviously have no linearization point.

By construction of $T.\mathcal{S}$, LSA guarantees that a transaction started at time t has a snapshot that is valid at or after the transaction started, i.e., $\lfloor T.\mathcal{S} \rfloor \geq t$. Hence, a read-only transaction can commit iff it has used a consistent snapshot for its whole lifetime (i.e., $T.\mathcal{S}$ remains non-empty). The global clock does not need to be increased when committing a read-only transaction because no object has been written. This optimization improves the memory cache hit rate if the clock is implemented as a counter in shared memory. Note that, as a consequence, multiple read-only transactions (even in the same thread) may share the same commit time.

3.5 Write Accesses and Update Transactions

Write accesses are very similar to reads except that one *must* always access the latest version $o_\star$ of an object o (lines 20–22) because a new version will be written at commit time. If the validity range of the latest version does not intersect with the snapshot even after extension, the transaction aborts (line 24).

When writing an object that has already been accessed by the current transaction, the version previously read or written must still be the most recent one. If a new version has been committed in the meantime, the transaction should abort because snapshot validation cannot succeed at commit time.

Informally, an update transaction T performs the following steps when committing: (1) it acquires a unique commit time t_c from the global time base ***clock***, which is atomically incremented (line 31), (2) it validates T (lines 32–35), and (3) it writes new versions of updated objects with timestamp t_c if validation was successful (lines 36–38), or aborts otherwise (line 35).

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, which guarantees that the transaction is atomic. This is checked during the validation step: $(t_c - 1) \in T.\mathcal{S}$. Therefore, accessed object versions must always be the most recent versions during the transaction.

The way conflicts are detected and new versions are atomically updated will be discussed in Section 5. One should note at this point that, if a new version of an object accessed by T has been written by another transaction with an earlier commit time $t < t_c$, validation will fail because $T.\mathcal{S}$ will have an upper bound strictly smaller than t and, hence, will not contain $t_c - 1$.

3.6 Proof of Linearizability

We now sketch proofs that transactions executed by an STM using LSA are linearizable. To that end, we need to show that T takes effect atomically between its start and its commit time. After introducing two lemmas, we demonstrate that this is the case for read-only and update transactions.

Lemma 1: For any transaction T that started at time t_s, we have at any time $\lfloor T.\mathcal{S} \rfloor \geq t_s$.

Proof: This property directly follows from the algorithm. $\lfloor T.\mathcal{S} \rfloor$ is initialized with the start time of the transaction and it never decreases (it is always set to the maximum of its current value and another value). □

Lemma 2: For any transaction T that has accessed at least one object, at any time t we have $\lceil T.\mathcal{S} \rceil \leq t$.

Proof: This property also follows from the algorithm. Each time an object is accessed, $\lceil T.\mathcal{S} \rceil$ is set to the minimum of the current time and another value. Upon extension, it never exceeds the current value of the clock. □

With the help of these lemmas, we can now prove that transactions executed with LSA are linearizable.

Theorem 1: LSA guarantees that every read-only transaction T that started at time t_s and that successfully commits between $t_c \geq t_s$ and $t_c + 1$ is linearizable.

Proof: T can only commit if its preliminary validity range $T.\mathcal{S}$ is non-empty when it commits. We know from lemmas 1 and 2 that $T.\mathcal{S}$ is contained in $[t_s, t_c]$. As $T.\mathcal{S}$ defines by construction a range during which all accessed objects are valid and not updated, T takes effect *atomically* at some time during $T.\mathcal{S}$, which happens *between the start and the end* of the transaction. □

Theorem 2: Each update transaction T that started at time t_s, that commits at time $t_c \geq t_s$, and that satisfies $\lceil T.\mathcal{S} \rceil \geq t_c - 1$, is linearizable.

Proof: On commit, LSA checks that $(t_c - 1) \in T.\mathcal{S}$ (lines 32–35) and, hence, that all object versions that T has accessed are still valid up to the time t_c when T commits its changes. Since each update transaction has a unique commit time, no other transaction can commit at t_c. This means that, logically, T reads all objects and commits all its updates *atomically* at time t_c, which happens *between the start and the end* of the transaction. □

4 Discussion

In this section, we discuss four aspects in the design space of the LSA related to the extension of snapshots, accesses to the global clock, the number of versions to keep, and the semantics of software transactional memory.

4.1 Snapshot Extensions

Validation is typically the performance bottleneck of STMs that use invisible reads. LSA only performs validation at commit time (for update transactions), or upon extension when accessing object versions that are more recent than the snapshot's upper bound. One might expect that LSA needs to perform extensions frequently when there are concurrent updates. It turns out, however, that 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 do not change and no extension is required for obtaining a consistent read snapshot. This is the case, in particular, if the value of ***clock*** has not changed since the start of T. If ***clock*** has been increased concurrently and T is an update transaction that commits at time t_c, one extension to $t_c - 1$ is needed. LSA requires at most one extension per accessed object. However, this worst case is extremely rare in practice because it requires very specific update patterns. In addition, once a concurrent update to an object previously accessed by T is detected, the validity range $T.\mathcal{S}$ becomes closed and no further extension is attempted. Experimental results (see Section 6) also suggest that extensions are seldom required.

4.2 Global Time

Accesses to the global commit time might become a bottleneck when many transactions execute concurrently. In practice, however, the number of accesses to ***clock*** remains small. All transactions must read the current time once when they are started, and update transactions must additionally acquire a unique commit time. Further accesses are not required for correctness. For example, if an update transaction needs to access a version more recent than its current validity range, it can extend the snapshot's upper bound up to any time at which the version was valid, not necessarily up to the current time (as shown in the algorithm). Time information gathered from the accessed objects can thus be used instead of reading the global commit time. Note again that the global clock can also be replaced by more scalable alternatives, such as approximately synchronized clocks [16], and various optimizations can be applied to improve performance of the commit phase [17].

4.3 Number of Versions

As previously mentioned, LSA can—but does not need to—maintain multiple object versions. The number of versions kept has an influence on the likelihood that a transactions can successfully commit.

LSA requires an update transaction to read the most current version of an object to be able to commit. A read-only transaction can commit even if its snapshot contains objects that have been overwritten at the time it commits. To guarantee the property that *any read-only transaction can commit without retry*, it would be sufficient to keep all object versions valid at the start of the transaction. The number of old versions to maintain per object could therefore be bounded by the number of concurrent read-only transactions.

When updating an object, we would need to check if the validity range of the replaced version contains the start time of an active read-only transaction. This check is prohibitively expensive to implement and, hence, we prefer maintaining a constant number of versions or use simple heuristics to determine at runtime the number of versions to keep. We evaluate several alternatives in Section 6.

4.4 Linearizability vs. Snapshot Isolation

Most STM implementations, including LSA, guarantee linearizability; i.e., each transaction appears to take effect atomically at a point between its start and its commit time. Some STMs guarantee serializability (e.g., [20], [21], [22]) in an attempt to increase the commit rate of the transactions, but they require more complex algorithms[4] and are not competitive in terms of performance.

LSA can be configured to provide *snapshot isolation* [11] semantics. The idea of snapshot isolation is to take a consistent snapshot of the data at the time when a transaction starts, and have all its read and write operations performed on that snapshot. When an update transaction tries to commit, it must acquire a unique timestamp that is larger than any existing start or commit timestamp. Snapshot isolation does not guarantee serializability but avoids common isolation anomalies like dirty reads, dirty writes, lost updates, and fuzzy reads. Snapshot isolation is an optimistic approach that is expected to perform well for workloads with short update transactions that conflict minimally and long read-only transactions. This matches many important application domains and slight variations of snapshot isolation are used in common databases.

When configured for snapshot isolation, only three minor modifications are necessary to the algorithm of Figure 1. First, no extensions are performed upon read or write (lines 8 and 19). Second, all read accesses are directed to the object versions that were valid at the start time of the transaction (lines 9–14). Third, validation is omitted upon commit (lines 32–35). It naturally follows that, when keeping sufficiently many versions, transactions can always commit except in case of write/write conflicts when executing under snapshot isolation.

Algorithms typically need to be adapted for snapshot isolation. Unlike linearizability, snapshot isolation permits read/write conflicts. In our experience, this makes algorithms more difficult to design because a programmer needs to identify which read/write conflicts need to be detected and convert them into write/write conflicts. For example, when removing an element from a linked list, one would need to add an extra write to the node that is removed. This prevents a concurrent transaction to insert a new element right after the removed one. Such a conversion is not always easy, e.g., trying to modify a red/black tree to support snapshot isolation proved to be more difficult than expected. Since the performance improvement of using snapshot isolation instead of linearizability showed to be minimal [6], we decided to only support linearizability.

5 LSA-BASED STM DESIGNS

We briefly outline in this section how one can use the LSA algorithm to implement concrete software transactional memories. We discuss two designs. The first one is object-based, written in Java, and provides obstruction freedom. The second one is word-based, written in C, and uses revocable locks (i.e., it is blocking). Both designs will be evaluated in Section 6.

5.1 An obstruction-free design

Our first LSA implementation is *object-based*, i.e., the granularity of conflict detection is an application-specific object (as defined by the programming language). As it closely follows the operation principle of DSTM [18], with the notable addition of multiple versions and LSA-specific data structures, we only give here a brief overview. It provides obstruction freedom, which informally means that any thread that runs for long enough by itself makes progress. We call this design LSA-STM.

LSA-STM is implemented as a Java software library. The main components exposed to the application developer are *transactions* and *transactional objects*.

Transactions are implemented as thread-local objects, i.e., the scope of a transaction is confined inside the current thread of control. The application developer can programmatically start a transaction, try to commit it, or force it to abort.

As in [18], transactions (see Figure 3) contain a status field, initially set to `ACTIVE`, that can be atomically changed to either `COMMITTED` or `ABORTED` using an atomic compare-and-swap (CAS) operation depending on whether the transaction successfully completes or not. A transaction can additionally keep track of the objects being read and updated (read-set and write-set) and maintains timestamps indicating the snapshot's lower and upper bounds. Timestamps are discrete values generated by a global lock-free counter (shared clock) that can be atomically read and incremented.

4. Unlike linearizability, serializability is not a local property. Serializable STM algorithms must typically maintain (partial) transaction dependency graphs at runtime.

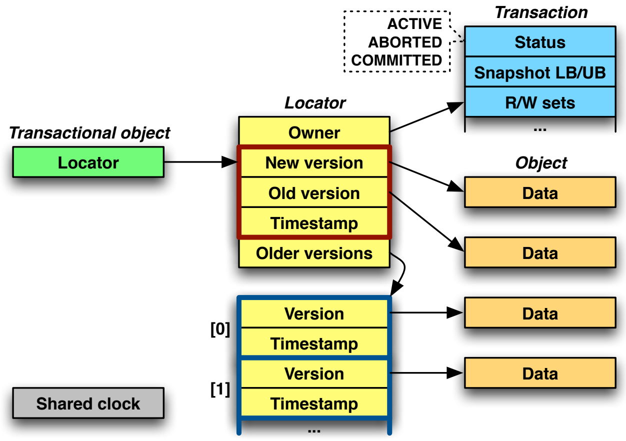


Fig. 3. Data structures for an obstruction-free design, similar to DSTM [18].

Transactional objects are STM-specific wrappers that control accesses to application objects. They manage multiple versions of the object's state on behalf of active transactions. Regular objects being wrapped must be able to duplicate their state, i.e., clone themselves, as transactional wrappers need to create new versions.

Transactional objects maintain a reference to a descriptor, called *locator* according to the terminology of [18], that keeps track of several versions of the object's state: a *tentative* (new) version being written to by an update transaction; a *committed* (old) version together with its commit timestamp; and several *previous* committed versions of the object together with their commit timestamp. A locator additionally stores a reference to its *owner*, i.e., the transaction that updates the tentative version, if any. Note that the locator does not keep track of transactions that read the object.

References to a locator can be read atomically and updated using a CAS operation. Once a locator is referenced by a transactional object, it becomes immutable and is never modified.

The *current* version of the object is defined as follows: if the owner field of the locator is null, or if the last writer has aborted, then the current version corresponds to the committed (old) version of the object with the commit timestamp stored in the locator; if the last writer has committed, then the current version corresponds to the tentative (new) version of the object with a commit timestamp equal to that of the writer; finally, if the writer is still active, the current version is undefined.

When a transaction writes an object for the first time, we check in the current locator whether there is already an *active* writer. If that is the case, there is a conflict and we ask a *contention manager* to arbitrate before retrying. A contention manager is a configurable module, invoked when a conflict occurs between two transactions, which must take actions to resolve the conflict, e.g., by aborting or delaying one of the conflicting transactions. If there is no conflict and the transaction's snapshot is valid, we create a new locator and register the current transaction as writer. We store references to the current and previous versions in the new locator and we create a new tentative version by duplicating the state of the current version. Finally, we try to update the reference to the locator in the transactional object using a CAS operation. If this fails, then a concurrent transaction has updated the reference in the meantime and we retry the whole procedure. Otherwise, the current transaction continues its execution by accessing its local tentative version. The use of the CAS operation is key to achieving obstruction-freedom in this design.

Upon reading an object for the first time, a transaction chooses the most recent version whose validity range intersects with the transaction's snapshot. If no such version exists, it aborts. One should stress that a transaction can read the last committed version of an object with an active writer. 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.

It is important to note that aborting or committing a transaction can be achieved by atomically changing the status of its descriptor from active to aborted or committed, respectively. Therefore, upon commit, after acquiring a new commit time and validating its snapshot, a transaction simply updates its status using a CAS operation. If this fails, it has been aborted in the meantime. Otherwise, all the objects updated by the transaction will be considered as committed by other transactions.

Our LSA-STM implementation uses a declarative approach based on Java byte-code transformations to ease integration of transactional memory in Java applications. The developer simply needs to add annotations to shared objects and atomic methods for the STM to automatically weave transactional constructs in the application. LSA-STM is freely available from http://tmware.org/lsastm.

5.2 A lock-based design

Our second LSA implementation is *word-based*, i.e., conflict detection is achieved at the level of memory addresses, and it uses revocable *locks* to protect shared data from concurrent accesses. It follows the same general principle used by TL2 [12] and other word-based STMs designs, notably Ennals' [23] and Saha *et al.*'s [2], [14]. It uses a *single-version* variant of LSA, i.e., transactions can only read the latest committed versions of an object. We call our implementation TINYSTM because of the simplicity of its design. Again, given the commonalities of TINYSTM with other classical STM implementations like TL2, we only describe its data structures and operation briefly.

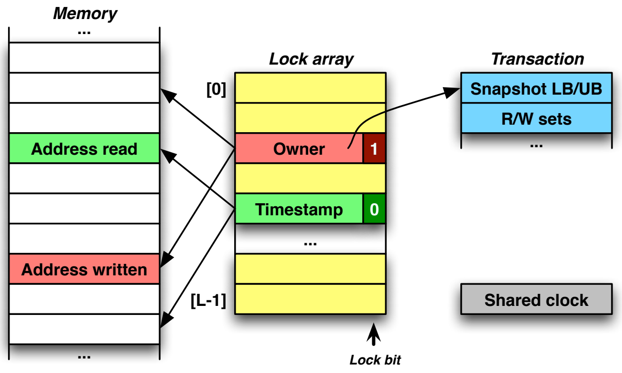


Fig. 4. Data structures for a lock-based design, similar to TL2 [12].

As several other word-based STM designs, TINYSTM relies upon a shared array of *locks* to protect memory from concurrent accesses (see Figure 4). Each lock covers a portion of the address space. In our implementation, we use a per-stripe mapping where addresses are mapped to locks based on a hash function.

Each lock is the size of an address on the target architecture. Its least significant bit is used to indicate whether the lock has been acquired by some transaction. If it is free, we store in the remaining bits a version number that corresponds to the commit timestamp of the transaction that last wrote to one of the memory locations covered by the lock.

If the lock is taken, we store in the remaining bits an address to the owner transaction.[5] Note that addresses point to structures that are word-aligned and their least significant bit is always zero; hence it can be safely used as lock bit.

When writing to a memory location, a transaction first identifies the lock entry that covers the memory address and atomically reads its value. If the lock bit is set, the transaction checks if it owns the lock using the address stored in the remaining bits of the entry. In that case, it simply writes the new value and returns. Otherwise, the transaction can try to wait for some time[6] or abort immediately depending on the contention management strategy. By default, we use the latter option in our implementation.

If the lock bit is not set, the transaction tries to acquire the lock by writing a new value—a pointer to itself and the lock bit—in the entry using a CAS operation. Failure indicates that another transaction has acquired the lock in the meantime and the whole procedure is restarted. If the CAS succeeds, the transaction becomes the owner of the lock. Our basic design does thus implement visible writes with objects being acquired when they are first encountered (this approach is usually called "encounter-time locking" or "eager acquire semantics"). Note that TINYSTM also provides a variant in which lock acquisition is delayed until the end of the transaction ("commit-time locking" or "lazy acquire semantics"), as will be discussed in Section 6.

When reading a memory location, a transaction must verify that the lock is not owned nor updated concurrently. To that end, the transaction reads the lock, then the memory location, and finally the lock again (obviously, appropriate memory barriers are used to ensure correct ordering of accesses). If the lock is not owned and its value (i.e., version number) did not change between both reads, then the value read is consistent.

Once a value has been read, LSA checks if it can be used to construct a consistent snapshot. If that is not the case and the snapshot cannot be extended, the transaction aborts.

Upon commit, an update transaction that has a valid snapshot writes its changes to memory and releases the locks (by storing its commit timestamp as version number and clearing the lock bit). Upon abort, it simply releases any lock it has previously acquired.

TINYSTM provides a simple C API for using STM in concurrent applications. It is freely available from http://tmware.org/tinystm.

6 PERFORMANCE EVALUATION

To evaluate the performance of LSA-STM, we compared it with two other classical implementations. The first one follows the object-based design with visible reads of SXM by Herlihy *et al.* [9]. The second follows the design of Eager ASTM by Marathe *et al.* as described in [10]. 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.

Similarly, TINYSTM has been compared with another widely used word-based implementation, TL2 [12], which also uses revocable locks. TL2 mainly differs from TINYSTM in that it does not support incremental snapshot construction, and it uses a commit-time locking strategy. To evaluate these implementations, we use a collection of micro-benchmarks, as well as the more realistic STAMP [24] benchmark suite.

6.1 Experimental Setup

All Java benchmarks were run on a Sun Fire T2000 server with an 8-core 1.2-GHz UltraSPARC T1 processor. Each core handles four hardware threads concurrently, i.e., the processor is capable of processing up to 32 concurrent threads. The machine has 16 GB of main memory and runs Solaris 10 with Java 1.6.0 (HotSpot 64-bit server VM, mixed-mode).

The C benchmarks were run on a machine with two quad-core 3 GHz Intel Xeon processors (X5365) and 5 GB of main memory running Linux 2.6.24-19 SMP (64-bit). Up to 8 threads can execute concurrently. Benchmarks were compiled using gcc version 4.2.3 and `-O3` optimization flags. We used TINYSTM version 0.9.5, TL2's x86 implementation version 0.9.6, and STAMP version 0.9.10.[7]

Results were obtained by executing ten runs of 10 seconds for every tested configuration and computing the 20%-trimmed mean, i.e., the mean of the six median values. Unless mentioned otherwise, Java STMs use the *Karma* [7] contention manager that provides good overall performance for most workloads (see Section 6.2.2).

6.2 Java Experiments

We evaluate several aspects of our Java-based LSA-STM implementation. We first consider transaction throughput. Next, we briefly observe the effect of different contention management strategies. We then study the impact of the number of versions kept, and finally the influence of extensions.

6.2.1 Throughput

Figure 5 shows throughput results for three micro-benchmarks that are commonly used to evaluate STM implementations, namely integer sets implemented via sorted linked lists, skip lists, and red/black trees. 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 are initially populated with a given number of elements and their size is maintained constant by alternating insertions and removals. We consider two sizes for each data structure and four different update rates ranging from 0% (i.e., all transactions are read-only) to 100% (i.e., only update transactions).

The linked list benchmark models transactions in which an update depends on a larger amount of data (nodes read during the list traversal) that might be concurrently modified by other transactions. We observe good scalability for low update rates. When the proportion of update transactions is high and

5. To be accurate, we store a pointer to an entry in the write set of the owner transaction for faster lookup.

6. Note that the transaction must not wait indefinitely as this might lead to deadlocks.

7. TL2 and STAMP are available from http://stamp.stanford.edu.

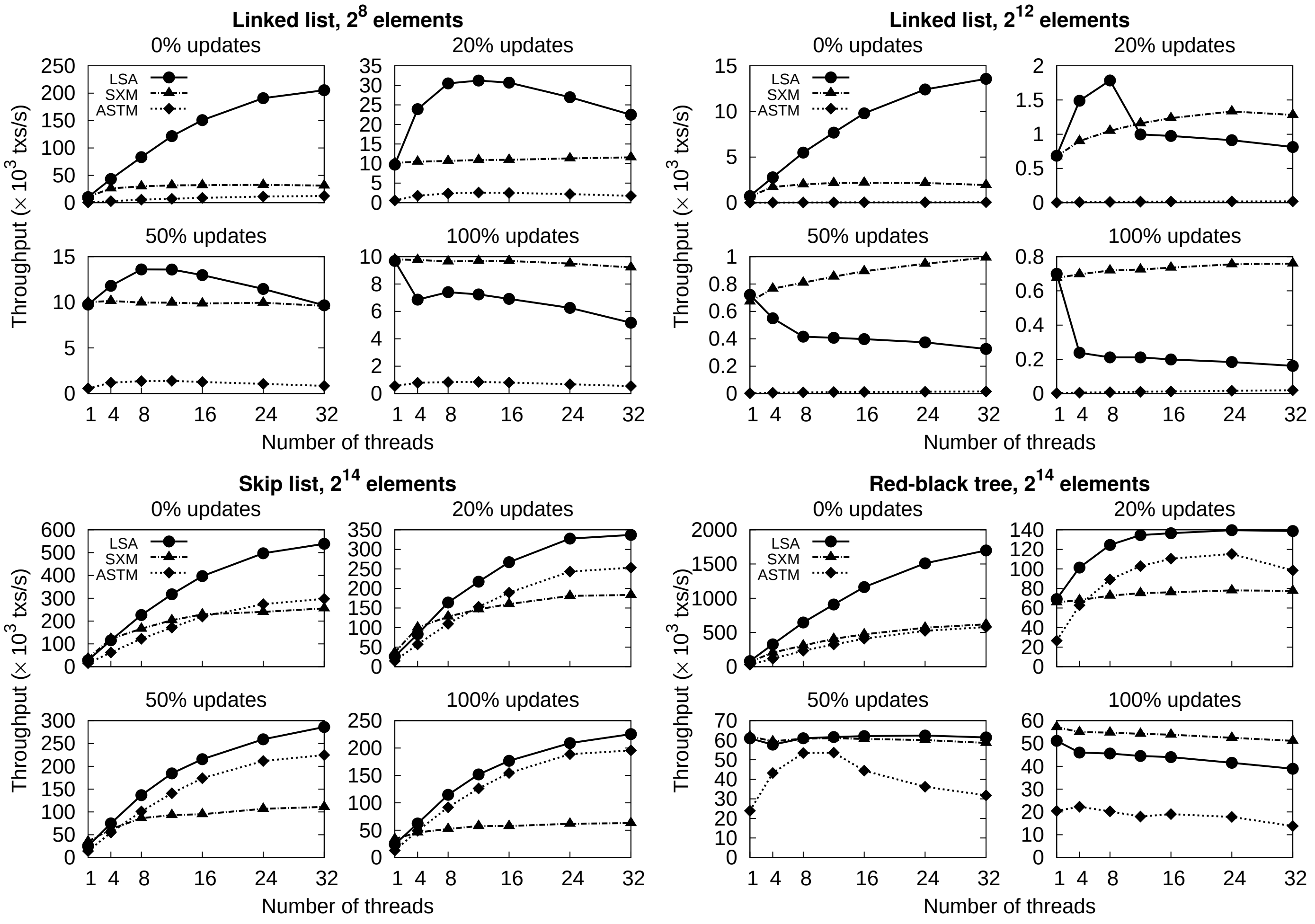


Fig. 5. Performance of LSA-STM compared to ASTM and SXM with three common micro-benchmarks.

when the list is long, adding more threads actually decreases performance. Only SXM handles write-dominated workloads satisfactorily because it uses visible reads and does not waste resources for doomed transactions that will eventually abort. The performance of ASTM is poor, especially with long lists, because of the incremental validation costs. In contrast, LSA-STM uses version information to compute the validity range much faster and scales up well to the number of available CPUs. Overall, LSA-STM performs significantly better than the other two designs on workloads with low contention.

For the skip list, STMs using invisible reads (ASTM and LSA-STM) outperform SXM, which suffers from the contention on the reader lists. As concurrent transactions typically follow different paths through the linked list, contention is limited even with high update rates and we observe good scalability with all workloads.

Finally, red-black trees exhibit the same trends as skip lists for low update rates, but do not perform as well when increasing the frequency of update. This can be explained by the larger number of nodes modified upon insertion and removal (due to repainting and rotations) and the higher contention, especially on nodes close to the root.

6.2.2 Contention Management

We have evaluated the performance of LSA-STM with different contention managers presented in the literature [7], [8], [25]. Our evaluation corroborates the conclusions of [8] that there is no single all-around winner. In fact, in our benchmarks, the choice of the contention manager had little influence on the transaction throughput. Figure 6 shows the performance of seven contention managers with the linked list and skip list benchmarks under a significant level of contention. The contention managers are: (1) *Aggressive* systematically kills the enemy transaction; (2) *Suicide* systematically aborts the current transaction; (3) *Timestamp* kills younger transactions and waits for the completion of older ones; (4) *Polite* waits for an exponentially increasing delay for the conflict to be

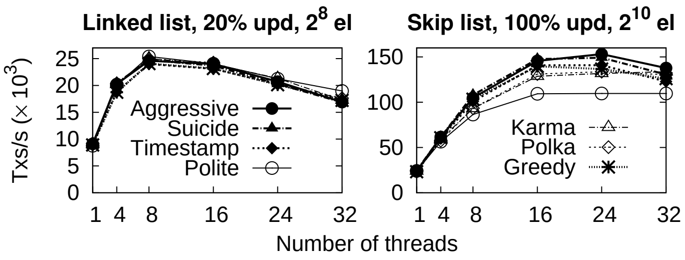


Fig. 6. Evaluation of several contention manager with the linked list and skip list benchmarks.

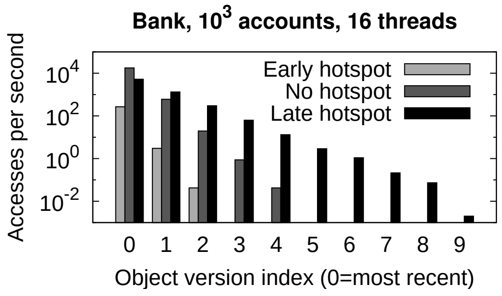


Fig. 7. Object versions accessed by long read-only transactions (bank micro-benchmark).

solved before killing the enemy; (5) *Karma* takes into account the amount of work already done by the transactions to decide which one to abort; (6) *Polka* extends *Karma* with a randomized exponential back-off mechanism; (7) *Greedy* is a variant of *Timestamp* with the additional rule that blocked transactions are killed upon conflict irrespective of their priority.

One can observe that the performance of all contention managers is remarkably similar. An intuitive explanation for this lack of significant differences is that LSA uses invisible reads: only few conflicts trigger the contention manager and transactions have reduced ability to determine which other transactions they are in contention with. When a transactions cannot extend or validate its snapshot, it aborts without intervention of the contention manager.

6.2.3 Number of Object Versions

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.

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 [18] 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 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).

Figure 7 shows access histograms of transactions computing the aggregate balance, with 16 threads performing a mix of 90% transfers and 10% balance computations on a set of 1,000 accounts. There are three benchmark modes: (1) no hotspots, that is, the 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.

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 hotspots are encountered late, the transaction has to use older versions if one of the objects accessed early has been updated, which prevents further extension of the validity range. 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.

We now study how the number of versions kept influencing performance. We investigate three strategies for determining the number of versions to keep. The first one consists in main-

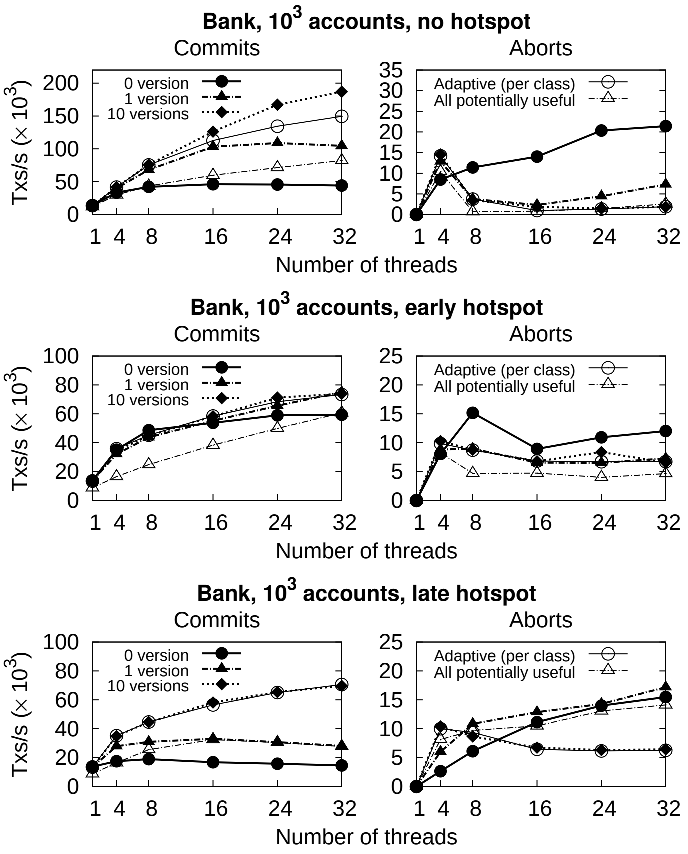


Fig. 8. Influence of the number of versions on performance without and with early and late hotspots (bank micro-benchmark).

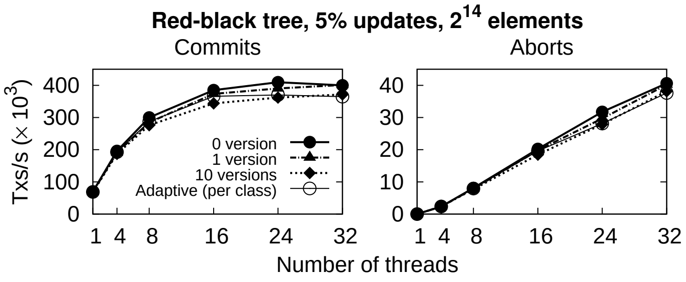


Fig. 9. Influence of the number of versions on performance (red-black tree with low contention).

taining a constant number of old versions (we experiment with 0, 1, and 10). With the second strategy ("per-class adaptive"), we maintain a per-class counter that is incremented each time a transaction aborts because it cannot find a version old enough to proceed; this counter indicates how many versions to keep for the objects of that class. The last strategy ("all potentially useful") dynamically determines the number of versions to keep by discarding versions whose validity range ends before the start time of the oldest active transaction.

Figure 8 shows the commit and abort rates of the bank application when using each of these strategies, with the same mix of 90% transfers and 10% balance computations. We can observe that keeping old versions can be beneficial as it can help long-running transactions to commit, especially with early hotspots. In fact, with multiple versions the number of aborts decreases when adding threads because old versions are more likely to be useful (i.e., to have been written by a non-conflicting thread). Dynamically determining which versions to discard based on the validity ranges ("all potentially useful") adds non-negligible runtime overhead and will not be considered further: keeping a constant or per-class adaptive number of versions provides the best performance vs. overhead trade-off.

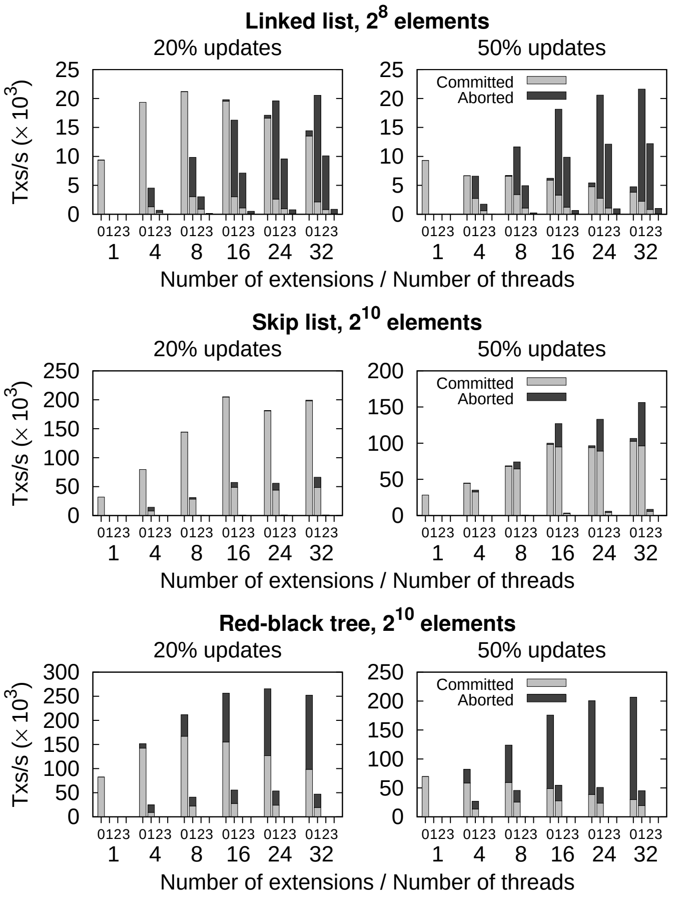


Fig. 10. Commit and aborts per extension for the linked list, skip list, and red-black tree micro-benchmarks.

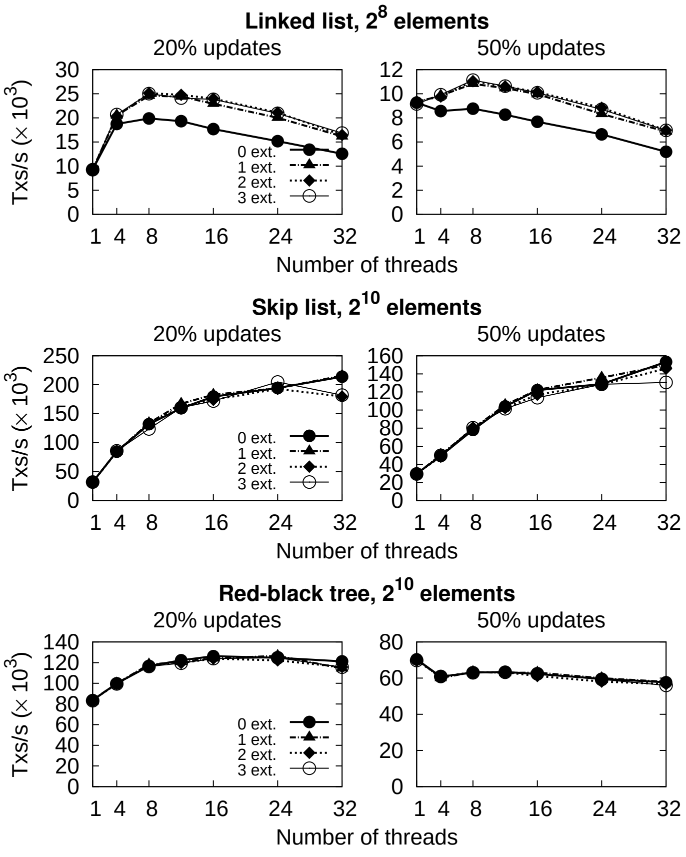


Fig. 11. Throughput when limiting the number of extensions for the linked list, skip list, and red-black tree micro-benchmarks.

On benchmarks with less contention, as for red-black trees (Figure 9), we can observe that the throughput actually increases when keeping less versions. Indeed, older versions are rarely necessary and the overhead of maintaining multiple versions dominates their benefits, as confirmed by the very limited reduction in abort rates (right-hand graph). In addition, reading old versions will close the validity range, which is disadvantageous for transactions that become update transactions after reading many objects, e.g., insertions in linked lists. Nevertheless, we have configured LSA-STM to use by default eight extra versions because this solution adapts well to various workloads (note that, in contrast, the variant of the LSA algorithm used by TINYSTM does not keep old versions).

6.2.4 *Snapshot Extensions*

We now study the number of validity range extensions performed by LSA-STM in the benchmarks and whether they are useful for ultimately committing transactions. Figure 10 shows the commit and abort rates for each successive validity

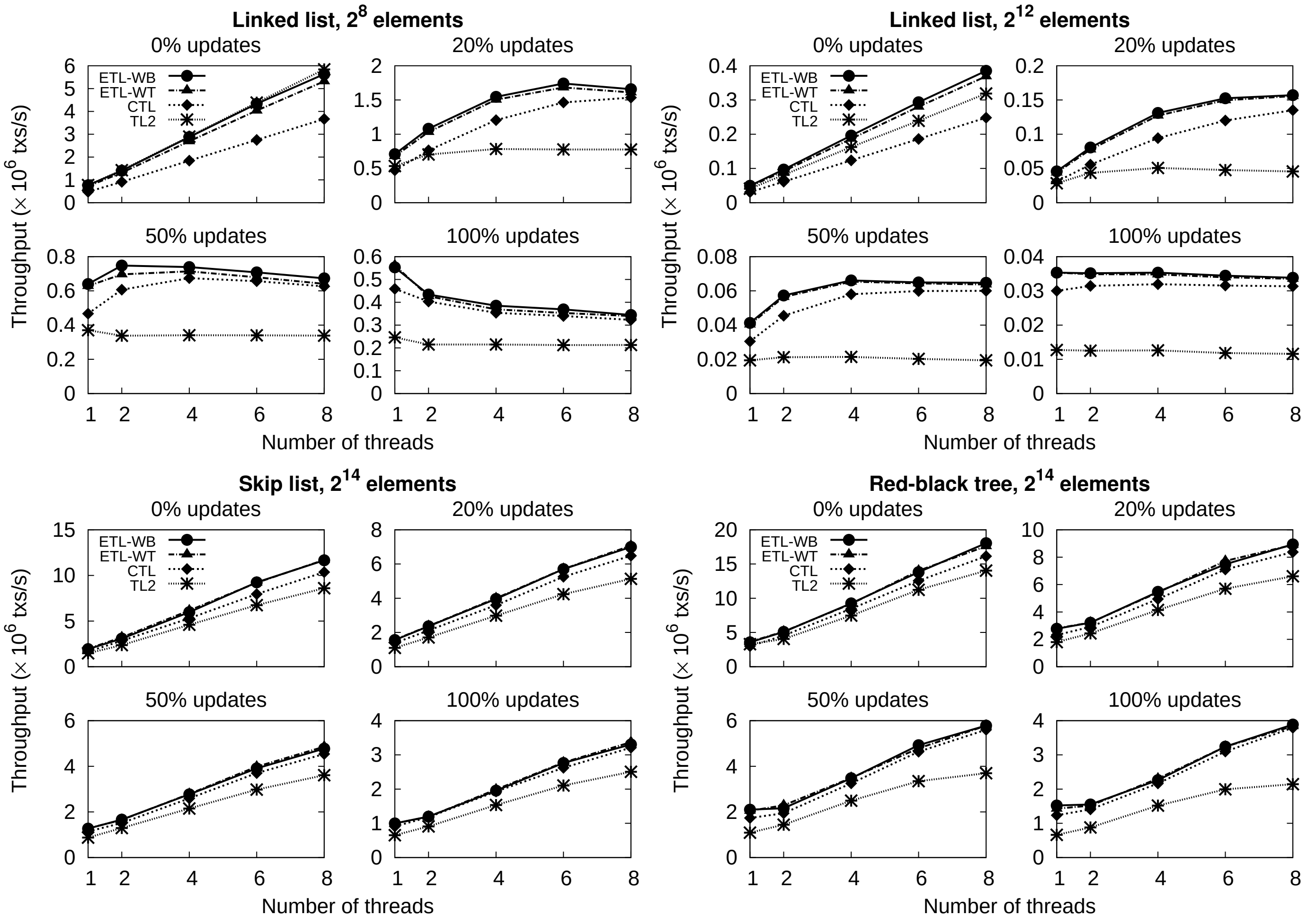


Fig. 12. Performance of TinySTM (ETL-WB, WTL-WT, CTL) and TL2 with three common micro-benchmarks.

range extensions with the linked list, skip list, and red-black tree micro-benchmarks. We first observe that the number of validity range extensions is very small. The vast majority of transactions uses less than two to four extensions.

6.2.5 Throughput

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. 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.

The figure shows that extensions do indeed help increase the number of committed transactions. In particular, with the skip list, more than half of the extended transactions successfully commit even with an update rate as high as 50%. Note however that this increase in number of committed transactions does not necessarily translate into better system performance because of the cost of performing the extensions, as discussed next.

Figure 11 shows the transaction throughput when limiting the number of extensions (i.e., the transaction aborts when the snapshot should be extended but the maximum number of extensions has been reached). One can observe that extensions provide the highest throughput benefits with linked list, as the cost of aborting is important (the complete list must be traversed again). In contrast, skip lists and red-black trees access less objects and the cost of an abort is proportionally smaller, hence extensions do not provide as much performance gain.

6.3 C Experiments

We now evaluate the performance of TinySTM, our LSA implementation in C. We first reproduce a subset of the Java experiments and we then evaluate TinySTM using the realistic STAMP [24] benchmark suite. We test three variants of TinySTM: ETL-WB uses *encounter-time locking* (i.e., locks are acquired at the time data is written) and a *write-back* update strategy (i.e., writes are buffered until commit time); ETL-WT also uses encounter-time locking, but with a *write-through* update strategy (i.e., writes are directly performed into main memory and an undo log is maintained in case of abort); CTL uses *commit-time locking* (i.e., locks are acquired at commit time and writes are buffered). We also compare the performance of TinySTM with the x86 port of TL2 [12].

Figure 12 evaluates the throughput of TINYSTM with the integer set micro-benchmarks, using the same workloads as for Java experiments (see Figure 5). We first observe that TINYSTM systematically outperforms TL2 by a small margin. Part of this difference can be explained by the extension mechanism of LSA, which helps improve throughput over TL2 especially with high update rates. Commit-time locking is slightly less efficient than encounter-time locking on these benchmarks, likely because the latter detects write conflicts early and avoids wasting time executing transactions that are doomed to abort. Scalability is good for all workloads, except write-dominated linked list where the cost of aborts is high due to the large number of transactional accesses. Remarkably, all STMs scale well with the skip list and red-black tree benchmarks even with 100% updates.

6.3.1 Snapshot Extensions

Figure 13 shows the commit and abort rates for each successive validity range extensions with the three integer set micro-benchmarks. Comparing with the Java experiments (see Figure 10), we observe that less extensions are performed and they are, in general, less successful. The obvious reason is that TINYSTM does not keep multiple versions. Nevertheless, with high update rates, a non-negligible of extensions lead to a commit especially in the linked list benchmark.

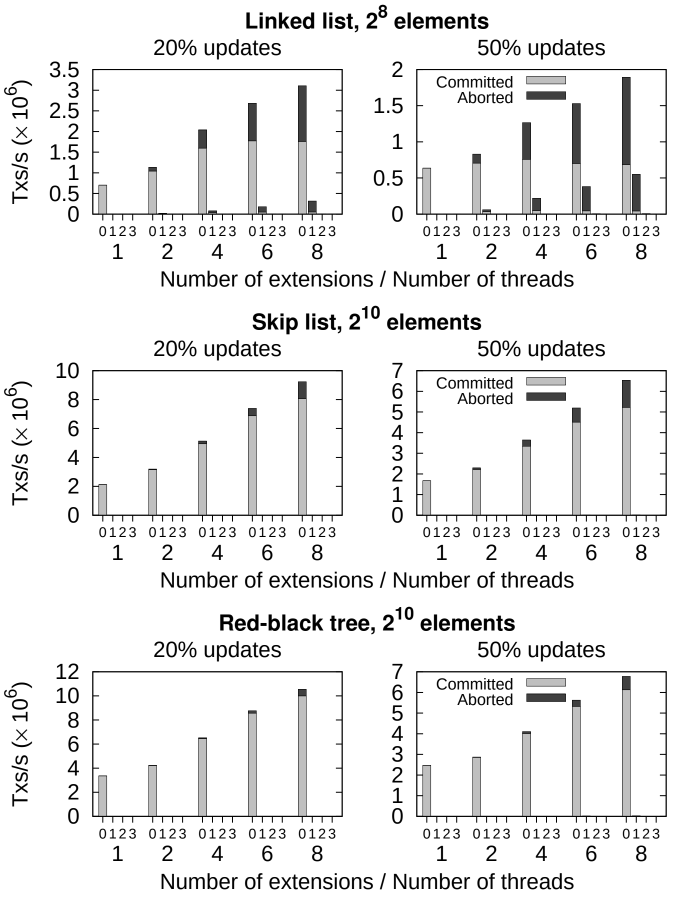


Fig. 13. Commit and aborts per extension for the linked list, skip list, and red-black tree micro-benchmarks.

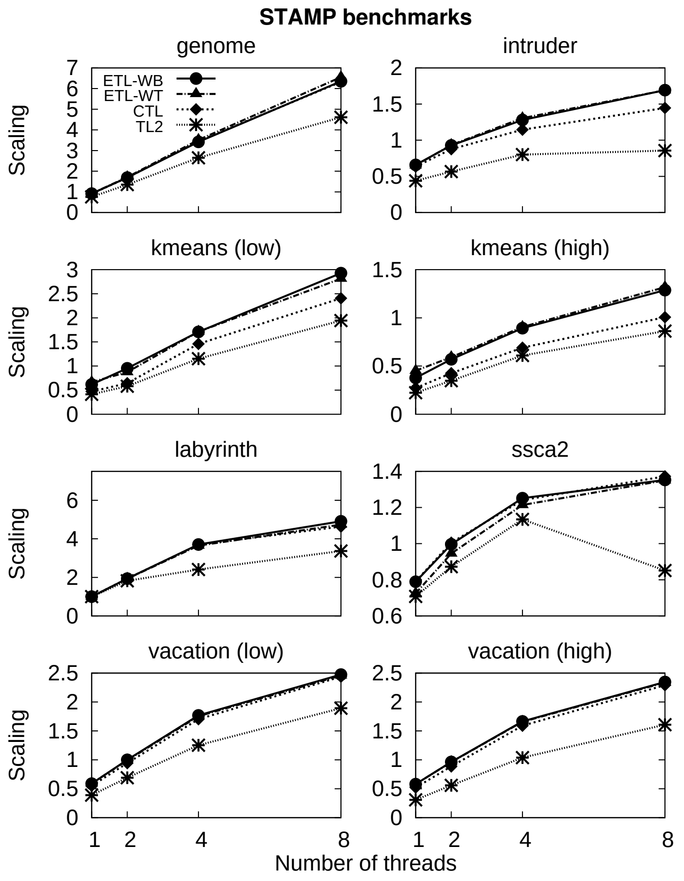


Fig. 14. Performance of TINYSTM (ETL-WB, WTL-WT, CTL) and TL2 with the STAMP benchmarks.

6.3.2 Real Applications

In our last experiment, we evaluate TINYSTM on STAMP [24], a set of realistic benchmarks. We ran tests using six different applications: `genome` takes a large number of DNA segments and matches them to reconstruct the original source genome; `intruder` emulates a signature-based network intrusion detection system; `kmeans` is an application that partitions objects in a multi-dimensional space into a given number of clusters; `labyrinth` executes a parallel routing algorithm in a three-dimensional grid; `ssca2` constructs a graph data structure using adjacency arrays and auxiliary arrays; and `vacation` implements an online travel reservation system. Additionally, two sets of parameters are recommended by the developers of STAMP for `vacation` and `kmeans`, for producing executions with low and high contention. The single-threaded execution time of STAMP applications takes from a few seconds to several minutes depending on the benchmark and parameters.

Performance results, shown in Figure 14, represent the scaling factor compared with a sequential execution without STM. While not all applications benefit as much from using STM, one can observe that both TINYSTM and TL2 exhibit good scalability up to 8 cores. The performance of TL2 is slightly lower on most experiments, which can be again explained by the differences in the underlying algorithms.

7 Conclusion

Time-based transactional memory inherits the performance of optimistic invisible reads without incurring the overhead of incremental validation. We have presented the original time-based STM algorithm, a lazy snapshot algorithm (LSA) that creates consistent snapshots on the fly. It is efficient both theoretically and practically. The idea is to maintain, for each transaction, a validity range based on global time that is sufficient to decide if a snapshot is consistent and if transactions are linearizable. The snapshots are created in such a way that their freshness is maximized: they can be dynamically extended and they might actually become valid at a time after the corresponding transactions have started. The algorithm takes advantage of old object versions, if any, to increase the probability of successfully constructing a consistent snapshot.

We have presented two instantiations of the LSA algorithm: an obstruction-free implementation in Java based on an object-based design, and a lock-based implementation in C that uses a word-based design. Performance evaluation demonstrates the benefits of the time-based approach for STM compared to earlier algorithms. As a matter of fact, several recent STM implementations have also adopted time-based designs.

Acknowledgements

This work was supported in part by European Union grant FP7-ICT-2007-1 (project VELOX).

References

[1] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, Aug. 1995.

[2] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP ' 06)*, 2006.

[3] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions," in *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI '06)*, Jun. 2006.

[4] D. Dice and N. Shavit, "What really makes transactions fast?" in *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '06)*, Jun 2006.

[5] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[6] T. Riegel, C. Fetzer, and P. Felber, "Snapshot isolation for software transactional memory," in *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '06)*, Jun 2006.

[7] W. N. Scherer III and M. L. Scott, "Contention management in dynamic software transactional memory," in *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.

[8] W. Scherer III and M. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC '05)*, Jul 2005.

[9] R. Guerraoui, M. Herlihy, and B. Pochon, "Polymorphic contention management," in *Proceedings of the 19th International Symposium on Distributed Computing (DISC '05)*, Sep 2005.

[10] V. Marathe, W. Scherer III, and M. Scott, "Adaptive software transactional memory," in *Proceedings of the 19th International Symposium on Distributed Computing (DISC '05)*, 2005.

[11] 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 (SIGMOD '95)*, 1995.

[12] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Proceedings of the 20th International Symposium on Distributed Computing (DISC '06)*, September 2006.

[13] M. Spear, V. Marathe, W. Scherer III, and M. Scott, "Conflict detection and validation strategies for software transactional memory," in *Proceedings of the 20th International Symposium on Distributed Computing (DISC '06)*, 2006.

[14] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai, "Code generation and optimization for transactional memory constructs in an unmanaged language," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*, 2007.

[15] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian, "Design and implementation of transactional constructs for C/C++," in *Proceedings of the 23rd annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications (OOPSLA '08)*, 2008.

[16] T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in *Proceedings of the 19th Symposium on Parallelism in Algorithms and Architectures (SPAA ' 07)*, June 2007.

[17] R. Zhang, Z. Budimlić, and W. Scherer III, "Commit phase in timestamp-based STM," in *Proceedings of the 20th Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*, Jun. 2008.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC '03)*, Jul 2003.

[19] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proceedings of the 20th International Symposium on Distributed Computing (DISC '06)*, Sep. 2006.

[20] J. Napper and L. Alvisi, "Lock-free serializable transactions," Department of Computer Sciences, University of Texas at Austin, Tech. Rep. TR-05-04, 2005.

[21] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber, "From causal to z-linearizable transactional memory (brief announcement)," in *Proceedings of the 26th annual ACM symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[22] U. Aydonat and T. S. Abdelrahman, "Serializability of transactions in software transactional memory," in *3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '08)*, Feb. 2008.

[23] R. Ennals, "Software transactional memory should not be obstruction-free," unpublished.

[24] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An effective hybrid transactional memory system with strong isolation guarantees," in *Proceeding of the 34th Intl. Symposium on Computer Architecture (ISCA '07)*, 2007.

[25] R. Guerraoui, M. Herlihy, and S. Pochon, "Toward a theory of transactional contention managers," in *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC '05)*, Jul 2005.

Prof. Pascal Felber received his M.Sc. and Ph.D. degrees in Computer Science from the Swiss Federal Institute of Technology. He has then worked at Oracle Corporation and Bell-Labs in the USA, and at Institut EURECOM in France. Since October 2004, he is a Professor of Computer Science at the University of Neuchâtel, Switzerland, working in the field of dependable, distributed, and concurrent systems. He has published over 80 research papers in various journals and conferences.

Prof. Christof Fetzer received his diploma in computer science from the University of Kaiserlautern, Germany in 1992 and his Ph.D. from UC San Diego in 1997. Dr. Fetzer joined AT&T Labs-Research in 1999 as a principal member of technical staff. Since April 2004 he has an endowed chair (Heinz-Nixdorf endowment) in Systems Engineering in the Computer Science Department at TU Dresden. He is the chair of the Computational Engineering International Masters Program. Prof. Dr. Fetzer has published over 80 research papers in the field of dependable distributed systems and has been member of more than 40 program committees.

Patrick Marlier received his diploma of Computer Engineer and his M.Sc. from the University of Technology of Compiègne, France in 2007. After an internship at Orange Labs, he joined the University of Neuchâtel in 2008 as a PhD student.

Torvald Riegel received his diploma in Computer Science from Technische Universität Dresden in 2005. Currently, he is a PhD student in the Systems Engineering Group of the same university. He won Sun Microsystems' CoolThreads Prize for Innovation in 2006.