Université de Neuchâtel
Institut d'Informatique

Rapport de Recherche — RR-I-07-02.1

From Causal to z-Linearizable Transactional Memory

Torvald Riegel, Heiko Sturzrehm, Pascal Felber, Christof Fetzer

February 20, 2007

From Causal to z-Linearizable Transactional Memory

Torvald Riegel[1], Heiko Sturzrehm[2], Pascal Felber[2], Christof Fetzer[1]

[1] Dresden University of Technology, Dresden, Germany

[2] University of Neuchâtel, Neuchâtel, Switzerland

Abstract

The current generation of time-based transactional memories (TMs) has the advantage of being simple and efficient, and providing strong linearizability semantics. Linearizability matches well the goal of TM to simplify the design and implementation of concurrent applications. However, long transactions can have a much lower likelihood of committing than smaller transactions because of the strict ordering constraints imposed by linearizability. In this paper, we investigate the use of weaker semantics for TM and introduce a new consistency criterion that we call z-linearizability. By combining properties of linearizability and serializability, z-linearizability provides a good trade-off between strong semantics and good practical performance even for long transactions.

1 Introduction

Transactional memory (TM) has been proposed as a lightweight mechanism to synchronize threads. It alleviates many of the problems associated with locking, offering the benefits of transactions without incurring the overhead of a database. It makes memory, which is shared by threads, act in a transactional way like a database. The main goal is to simplify the development of concurrent applications, which are becoming more widespread because of the increasing shift to multicore processors and multiprocessor systems.

We refer to a TM that is based on a notion of time or progress as a *time-based transactional memory* (TBTM). Some sort of global time base is used to reason about the consistency of data accessed by transactions and about the order in which transactions commit. Typically, TBTMs employ optimistic read operations (i.e., read operations are not visible to other transactions) because invisible reads are less expensive than visible reads. TBTMs then guarantee on the basis of their time base that the snapshot that a transaction takes of the transactional memory at runtime is always consistent. TBTMs thus have an advantage over TMs that only ensure the consistency of the transaction at commit time: transactions that work with inconsistent data could for instance enter infinite loops or throw unexpected exceptions. Nevertheless, the per-access costs of TBTMs are small. Several recent STM designs [8, 2, 10, 13] have evolved to being time-based. Current TBTMs will be more closely described in Sections 2 and 3.

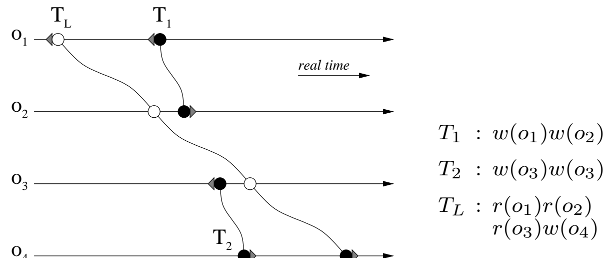

Figure 1: Linearizability schedules T_1 before T_2, which forces long transaction T_L to abort (each transaction executes in a separate thread). Serializability would allow all three transactions to commit.

Problem. TBTMs provide strong semantics (linearizability) and are efficient and scalable. However, linearizability restricts the concurrency one can achieve, in particular with long transactions. Indeed, as they rely on the "first committer wins" rule, a commit by any concurrent short transaction may force a long transaction to abort. Further, the fact that linearizability preserves the real time ordering of transactions forbids some schedules valid for the application (e.g., serializable schedules) that would let the long transaction commit. Consider

for instance the three transactions in Figure 1 executing on three different threads. Filled circles correspond to write operations (i.e., new versions that will be visible to other transactions at commit time) while empty circles represent read operations. Transaction demarcation is represented by the gray triangles. Linearizability imposes an ordering of T_1 before T_2 , which prevents long transaction T_L from committing, even though there is a valid serialization. Linearizability may thus limit concurrency because it imposes a total order on transactions even when they access disjoint object sets and would not otherwise need to be ordered.

Contributions. In this paper, we investigate whether the use of semantics weaker than linearizability can increase the throughput of long transactions. In particular, we consider two consistency criteria, causal serializability and serializability, and we show how one can implement them in a TBTM using vector clocks or plausible clocks.

We then introduce a novel criterion, which we call z-linearizability, that combines features of linearizability and serializability and offers a good tradeoff between strong semantics and good performance even for long transactions. With z-linearizability, long transactions partition short transactions into different "time zones". The set of long transactions and each set of short transactions within a zone are linearizable; the set of all transactions is serializable.

Besides being a consistency criterion with semantics sufficiently strong for most applications, z-linearizability also provides good practical performance. Our evaluation shows that the throughput of long transaction significantly increases without penalizing the overall throughput, unlike approaches based on vector clocks that suffer from a non-negligible runtime overhead.

Roadmap. The rest of the paper is organized as follows. Section 2 gives an overview of time-based transactional memory. Section 3 discusses related work. In Section 4, we show how one can use vector clocks to support (causal) serializability. Section 5 introduces the z-linearizability consistency criterion and algorithms. Finally, Section 6 concludes.

2 Overview of TBTMs

Current TBTMs execute transactions by performing two phases. First, a consistent snapshot of accessed objects is built during an execution of the transaction. If it is not possible to construct such a snapshot, then the transaction is aborted. The snapshot is virtually taken at a certain time of the global time base, which we call the *snapshot time*. This can also be a time interval in some implementations.

Read-only transactions can commit directly after the snapshot phase. Update transactions, in contrast, have to perform additional steps to commit. Updates to shared objects are either held in a transaction-local storage or are immediately written to the objects. Current TBTMs use write locks or a nonblocking equivalent. Write locks of a transaction are released when the transaction aborts or commits.

When an update transaction tries to commit, it first sets write locks or activates previously set write locks. Then, the transaction obtains a commit time from the global time base. The transaction has to make sure that all other transactions performing snapshots at the commit time also have seen the activated locks. Thus, the update transaction can either acquire a new commit time, e.g., by incrementing a commit time counter, or wait one clock tick. The commit time is then the time at which the transaction's updates become visible.

Before the update transaction can commit, it must be validated. This checks that the snapshot is still valid and consistent at the commit time (i.e., there have been no concurrent updates to objects read by the transaction between the snapshot time and the commit time). This ensures serializability because the time interval at which write locks are visible and the validity interval of the snapshot intersect (i.e., it works like two-phase locking). It also ensures linearizability if the time base is linearizable and transactions do not take snapshots in the past.

The simplest implementation for a global time base is a global shared linearizable integer counter. The current time is obtained by reading the counter. The counter is atomically incremented whenever a commit time is acquired (i.e., an update transaction commits), which models progress in the TBTM.

However, such a counter does not scale well in larger systems because of contention and cache misses. On the other hand, it has a small space overhead and integer values are inexpensive to compare.

Using real-time clocks is another option. Perfectly synchronized clocks are almost as easy to use as a shared counter and they scale very well with the number of cores or processors. However, it is difficult to perfectly synchronize clocks in software and it usually requires dedicated hardware (which we expect systems will have). If hardware support is not available, internally synchronized real-time clocks are sufficient to implement TBTM but the probability of spurious aborts increases with the deviation of clocks. Further details about how to implement TBTMs with real-time clocks can be found in [9].

3 Related Work

The Lazy Snapshot Algorithm (LSA) [8] is an algorithm for TBTMs that support multiple object versions and uses a shared integer counter as time base. The algorithm works in the way described previously and tries to extend the validity interval of the snapshot if necessary. Transactional Locking II (TL2) [2] is a TBTM that uses a similar algorithm but is optimized towards providing a lean STM and decreasing overheads as much as possible; only one version is maintained per object and no validity extensions are performed. One further TBTM is presented in [13]. It uses a single-version variant of the algorithm described in [8] and performs compiler optimizations.

All these three TBTMs use a shared integer counter as time base and thus potentially suffer from contention on the counter. We describe in [9] how shared counters can be replaced with more scalable time bases, namely real-time clocks of arbitrary speed and sets of synchronized clocks with a bounded deviation between the clocks. At least parts of the overhead of the shared integer counter are avoided in TL2 [2] by letting transactions share commit times.

RSTM [10] is an STM that uses a shared integer counter that counts commits to be able to use the validation fast path. It reads the counter when opening a transactional object and skips object-level validation if there has been no progress in the system.

Our Z-STM (see Section 5) guarantees z-linearizability and is based on a combination of an optimistic time stamp ordering [11] for long transactions together with LSA for short transactions. Long transactions that violate the timestamp ordering rule (i.e., an operation by transaction T_1 on object o is executed before an operation by transaction T_2 iff T_1 has a smaller timestamp than T_2) are aborted. Since we expect long transactions to interfere very rarely (because most of the executed transactions are expected to be short), this simple scheme works very well in practice.

4 A Vector Time Base

A reasonable way to improve the chances for long transactions to commit is to use weaker semantics to improve the concurrency of transactions. Some weaker types of semantics can be implemented with the help of vector clocks, which have the potential to increase the concurrency but also produce higher runtime overheads. One might be able to cope with these runtime overheads using various optimizations like plausible clocks (see below). We show in this section how one can implement STMs that guarantees causal serializability and serializability before discussing their effectiveness for dealing with long transactions.

Vector clocks were proposed independently by Fidge [3] and Mattern [6] as a technique to characterize causality. Each processor maintains a local monotonically increasing counter that represents its *local time*. In a system with n processors, we represent the *global time* as a vector of n components, where the ith element is the local time of processor i. A processor perceives the local time of another processor when communicating via accesses to shared objects. The *perceived global time* of a processor is a vector clock containing its up-to-date local clock and a stale version of the local clocks of other processes. Each time a processor receives a vector clock from another processor, it updates its perceived local time by computing the element-wise maximum of both vectors.

Vector timestamps are compared using the following rules:

(1) $t_i = t_j \Leftrightarrow \forall k, t_i[k] = t_j[k]$;
(2) $t_i \preceq t_j \Leftrightarrow \forall k, t_i[k] \leq t_j[k]$;
(3) $t_i \prec t_j \Leftrightarrow t_i \preceq t_j \wedge t_i \neq t_j$.

Unlike Lamport's logical clocks [5], vector clocks can accurately determine whether two events e_i and e_j with distinct vector timestamps t_i and t_j are causally related or not:
(1) $e_i \rightarrow e_j \Leftrightarrow t_i \prec t_j$;
(2) $e_i \parallel e_j \Leftrightarrow t_i \nprec t_j \wedge t_j \nprec t_i$.

Vector clocks can be used as a time base for implementing a TBTM. They offer two attractive features. First, unlike time bases implemented with a single shared counter (see Section 2), vector clocks do not suffer from contention on the time base because each thread can have its own component in a vector clock and timestamps are exchanged between threads only when they access shared objects. This loose synchronization can be beneficial when interprocessor communication delays are not negligible or contention on the shared counter is high.

Second, vector timestamps allow us to identify transactions that are not causally related and do not conflict, for which it is not necessary to determine a total ordering at commit time.

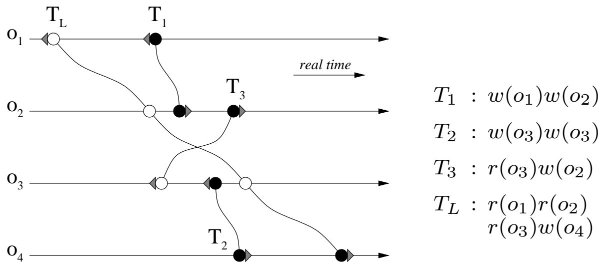

Figure 2: A causally serializable, but not serializable, execution (each transaction executes in a separate thread).

Consider again Figure 1. In a TBTM that uses a single clock (e.g., a shared counter), transaction T_1 commits before, and is thus ordered before, T_2. Hence, T_L must abort because it reads versions of o_1 and o_2 that are not valid anymore at the time T_L commits. This ordering of T_1 and T_2 is not necessary because the two transactions access disjoint sets of objects. There is a valid serialization $T_2 \rightarrow T_L \rightarrow T_1$ but one cannot take advantage of it. The reason is that, without keeping track of causal relationships, we are not able to determine that T_1 and T_2 can execute in any order even though the former commits before the latter in real time.

4.1 Causal Serializability

Moving from single clocks to vector timestamps allow us to easily implement causal serializability [7], a consistency criterion weaker than serializability but stronger than causal consistency. Informally, causal consistency allows each processor to have its own sequential view of the execution as long as the individual views preserve the causality relation. Causal serializability adds the additional constraint that all transactions that update the same object must be perceived in the same order by all processors. For instance, the execution shown in Figure 2 is causally serializable: the thread running T_3 observes the sequential execution $T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_L$ while the thread running T_L observes $T_2 \rightarrow T_L \rightarrow T_1 \rightarrow T_3$. It has been argued that causal serializability is a consistency criterion strong enough to satisfy a wide range of applications [7].

Note that causal serializability provides semantics comparable to snapshot isolation [1], but whereas the latter requires a snapshot to be observed, the former only requires transactions to observe causally consistent views.

Algorithm. The pseudo code of the causally serializable STM (CS-STM) is shown in Algorithm 1. For simplicity, we assume that methods execute atomically in the pseudo-code. Our implementation of CS-STM ensures atomicity using an approach similar to DSTM [4] where shared objects are accessed indirectly via "locators" and atomic updates are performed using compare-and-swap operations.

The CS-STM algorithm works as follows: Shared objects traverse a sequence of versions. Each time a transaction that has written to a shared object commits, it installs a new version. A vector timestamp is associated with each object version, corresponding to the commit time of the transaction that last updated the object. We denote the timestamp of version v_i of an object by $v_i.ct$. The validity of a version lasts until the start of the next version, i.e., the validity of version v_i is $[v_i.ct, v_{i+1}.ct)$.

A transaction T computes incrementally its commit timestamp $T.ct$ during its execution. Initially,

it is set to the timestamp of the transaction that was last committed by the current thread (line 3). Thereafter, each time an object is read or written, $T.ct$ is updated by computing the element-wise maximum with the timestamp of the accessed object version (line 8). Reads are invisible, i.e., a transaction does not know which objects have been read by other transactions. Therefore, read/write conflicts are not detected before commit time.

Algorithm 1 CS-STM (algorithm of thread p)

```
 1: procedure START(T)                      ▷ Initialize transaction attributes
 2:     T.state ← active
 3:     T.ct ← VC_p                          ▷ Tentative commit timestamp
 4:     T.rs ← ∅                             ▷ Read set (objects read by T)
 5: end procedure

 6: procedure OPEN(T, o_i, m)               ▷ T opens o_i in mode m
 7:     v_i ← CURRENT(o_i)                   ▷ Last committed version
 8:     T.ct ← max^(T.ct, v_i.ct)            ▷ Element-wise maximum
 9:     if m = write then                    ▷ T updates o_i
10:         if o_i.writer ≠ null then
11:             arbitrate(T, o_i.writer)     ▷ CM resolves conflict
12:         end if
13:         o_i.writer ← T
14:         v_{i+1} ←DUPLICATE(v_i)          ▷ Tentative version
15:         return v_i+1
16:     else                                 ▷ T reads o_i
17:         return v_i
18:     end if
19: end procedure

20: procedure VALIDATE(T)                   ▷ Validate the read set of T
21:     for all v_i ∈ T.rs do
22:         if ∃v_{i+1} s.t. v_{i+1}.ct ≺ T.ct then
23:             ABORT(T)                     ▷ Not causally serializable
24:         end if
25:     end for
26: end procedure

27: procedure COMMIT(T)                     ▷ Try to commit transaction
28:     VALIDATE(T)                          ▷ Verify causal serializability
29:     T.ct[p] ← T.ct[p] + 1                ▷ Increment p's component of VC
30:     T.state ← committed                  ▷ Commit tentative versions
31:     VC_p ← T.ct                          ▷ Remember last committed timestamp
32: end procedure
```

Transactions always read the last committed version of an object (hence, old versions do not need to be kept)[1] and a single writer is allowed per shared object. When two transactions try to update the same object, only one of them is allowed to proceed; the other one has to abort or wait. Therefore, write/write conflicts are prevented by allowing just a single writer (lines 10–12). Conflict arbitration is performed by a configurable module called *contention manager*, which is responsible for the liveness of the system.

At commit time, transaction T validates that the objects it has read are still valid (lines 20–26), i.e., it looks for read/write conflicts. To that end, the transaction verifies that the upper bound of the validity range of each object version v_i in its read set is not strictly smaller than $T.ct$, i.e., $v_{i+1}.ct \not\prec T.ct$. If validation is successful, the current thread increment its local component in $T.ct$ (this step is not necessary if T is a read-only transaction) and commits. Otherwise, T must abort.

Consider again Figure 1. Assuming that transactions T_1, T_2, and T_L execute respectively on three threads p_1, p_2, and p_3, with an initial vector clock of $[0,0,0]$, we have: $T_1.ct = [1,0,0]$ and $T_2.ct = [0,1,1]$. As $T_1.ct \not\prec T_2.ct$ and $T_2.ct \not\prec T_1.ct$, transactions T_1 and T_2 are not causally ordered. $T_L.ct$ is computed as $[0,1,1]$ and upon validation we have $T_1.ct \not\prec T_L.ct$ (test on line 22); therefore, T_L can commit.

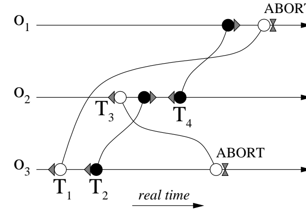

Figure 3: Transactions T_1 and T_3 must abort because they have read object versions that are not valid at commit time.

Consider now Figure 3. Transactions T_3 cannot commit because it has read object versions that both causally precede and follow T_2. Similarly, T_1 must abort because the version of o_3 that it has read has been overwritten by causally preceding transaction T_2. In both cases, validation fails.

Correctness. To show that the CS-STM algorithm ensures causal serializability, we need to prove that (1) the individual views of committed transactions at each processor preserve the causality relation, and (2) all transactions that update the same object must be perceived in the same order by all processors.

Condition (1) follows from the construction of

[1] Keeping multiple versions would allow a transaction to choose the version (line 7) that maximizes the chances of successful validation and could thus increase concurrency at the price of a higher space overhead.

timestamps: a committed transaction has a timestamp strictly greater than all the object versions that is has accessed (line 8). Therefore, timestamps reflect the causality dependencies between the transaction that have previously updated the objects, and the transactions that access them later. The validation phase (lines 20–26) guarantees that no transaction can both causally precede and follow another transaction. Indeed, a transaction T that does not pass the validation phase would see the transactions that have updated objects from its read set but have a smaller commit timestamp as occurring both after and before T and would not be able to construct a valid serialization. If the validation phase succeeds, then these transactions causally follow T in the view of the processor that executes T (but they may precede it in the view of other processors as timestamps are not comparable).

Condition (2) trivially follows from the fact that we only allow a single writer per object. Once a version has been committed, any transaction that updated the same object will have a commit timestamp strictly greater and will be causally ordered after the first transaction.

4.2 Serializability

Although causal serializability is sufficient for a wide range of applications, one also often needs stronger consistency criteria. Serializability ensures that the concurrent execution of a set of transactions is equivalent to some sequential execution. As compared to causal serializability, the additional constraint is that all update transactions (irrespective of the objects that update the same object) must be perceived in the same order by all processors. Although conceptually simple, this constraint is surprisingly difficult to deal with if one does not want to artificially reduce concurrency.

Consider Figure 2. The execution of all four transactions is clearly not serializable: only one of T_L or T_3 can commit without breaking serializability as both transaction try to impose a different, incompatible ordering between events that are not causally related. Yet, we want to allow either of these transactions to commit, i.e., we do not want to order T_1 and T_2 a priori. The first transaction of T_L or T_3 that commits will order T_1 and T_2; the other one will abort.

Algorithm. The objective of our algorithm is to maintain transactions unordered as long as possible and, once a transaction imposes an order, prevent any other transaction to change this order. Considering again Figure 2, assume that T_3 commits and hence orders T_1 before T_2. Note that $T_2.ct \parallel T_3.ct$. A solution to ensure that this ordering cannot be changed is to force any transaction accessing objects updated by T_2 after T_2 has committed, i.e., versions that causally follows T_2, to have a commit timestamp greater than that of T_3 (by construction, the timestamp will also be greater than that of T_2).

The general principle of our serializable STM (S-STM) algorithm is to have object versions keep track of active transactions that have read past versions. If such a transaction commits, one must make sure that the new version of the associated object has a timestamp strictly greater than that of the committed reading transaction. Information about past readers is carried along causal chains.

S-STM works along the same lines as CS-STM, with the major following differences: First, reads are visible, i.e., a reading transaction lets other transactions know that is is reading an object. To that end, a reading transaction atomically inserts itself in a "reader list" associated with the read version.

Second, timestamps associated with transactions carry besides their vector clock a list of active transactions that were reading the *previous* version(s) of the objects accessed by the transaction at the time it has committed. This list is constructed at commit time and its size does not exceed the number of threads as it only needs to hold active transactions. New versions of updated objects hold a reference to the timestamp of the transaction that wrote this version, and hence to the list of causally preceding transactions. This allows us to construct a partial precedence graph of active transactions at runtime. At commit time, we make sure that the timestamp of the transaction is larger than that of any committed transaction that causally precedes the timestamp of the objects accessed by the committing transaction. A conflict occurs if we detect a cycle, i.e., an active transaction causally precedes

another active transaction and conversely. This can easily be checked at commit time using the lists of causally preceding transactions.

The algorithm augments CS-STM by preventing cycles in the precedence graph and, hence, guarantees that the schedule is serializable.

Implementation notes. The details of our implementation are omitted as they are quite intricate, especially regarding the handling of race conditions when maintaining the partial precedence graph. While we do not rely on locks (we use compare-and-swap operations), we use an additional state to indicate when transactions are *committing*. A transaction that cannot progress because it waits for the outcome of a committing transaction helps that transaction commit. This may result in unnecessary work but improves liveness and fault tolerance in case the committing transaction is delayed or crashed.

The runtime overhead of managing both vector timestamps and the partial precedence graphs can be deemed prohibitive, especially for short transactions. Therefore, this algorithm is mostly useful when (1) causal serializability is not sufficient for the considered application, and (2) the increase in the level of concurrency offered by the algorithm compensates for the overhead.

4.3 Practical Vector Time using Plausible Clocks

The main drawbacks of vector clocks are their runtime and space overheads. Indeed, storing, updating, and comparing vector timestamps is significantly costlier than managing a single counter. Furthermore, their size varies according to the number of threads, which might not be known a priori (although one would typically like to map only one thread on each processor or core).

A solution to that problem is to use *plausible clocks* [12] instead of vector clocks. They combine ideas from logical and vector clocks: they have the same theoretical strength as scalar clocks, but better practical accuracy. Plausible clocks can always determine the order of causally related events correctly but may order events that are actually concurrent.

Given any two events e_i and e_j of a global history, a plausible time sampling system $\mathcal{C}$ guarantees that:

(1) $e_i \stackrel{\mathcal{C}}{=} e_j \Leftrightarrow e_i = e_j$;

(2) $e_i \stackrel{\mathcal{C}}{\rightarrow} e_j \Rightarrow (e_i \rightarrow e_j) \vee (e_i \parallel e_j)$;

(3) $e_i \stackrel{\mathcal{C}}{\leftarrow} e_j \Rightarrow (e_i \leftarrow e_j) \vee (e_i \parallel e_j)$;

(4) $e_i \underset{\mathcal{C}}{\parallel} e_j \Rightarrow e_i \parallel e_j$.

Plausible clocks offer a compromise between accuracy and size. It turns out that our CS-STM and S-STM algorithms can use plausible clocks instead of vector clocks with almost no modifications. Indeed, vector clock comparisons are used to verify the validity of objects read by a transaction. For validation to abort, there must be some causal dependencies between the versions read and the commit time. As plausible clocks always report causal relationships correctly, correctness is not violated. However, some concurrent events may be reported as causally related, leading to unnecessary aborts.

We have implemented plausible clocks based on r-entries vectors (REV), where each timestamp is a vector of $r \leq n$ elements for n processors. As there are fewer entries in the vector than processors, entries are shared. There are many possible mappings between processors and entries but, in our study, we only consider the *modulo* r mapping, where processor p_i uses entry $i \mod r$. Shared entries are incremented atomically using a get-and-increment operation to avoid that two threads generate the same timestamp. In the extreme case where $r = 1$, we have a single-clock TBTM (see Section 2) whereas, when $r = n$, we have classical vector clocks. Note that there exist other types of plausible clocks [12].

4.4 Discussion

Although weaker semantics like causal serializability and serializability can offer a higher degree of concurrency for long transactions, it is difficult to use them in practical implementations. Even when using vector clocks, a TBTM still has to build a consistent snapshot at a certain time. Vector time provides some flexibility in the way a snapshot is constructed as it can take effect at different times for different objects, but long transactions still have to strive to find a consistent snapshot that is not invalidated by concurrent transactions.

Second, using vector clocks requires all objects to

participate in tracking causality. It would be preferable to only impose this overhead on long transactions and on the transactions that are in direct conflict with them.

Third, a TBTM typically needs old object versions to construct a consistent snapshot for a long transactions when objects are being updated concurrently. Keeping multiple copies does not only increase the memory overhead but also the runtime overhead because these versions need to be tracked.

In our experiments with CS-STM and S-STM we have observed that single-version objects can decrease performance and that the overheads of vector clocks and the respective STM algorithms are quite high. We also know that single-clock TBTMs provide good performance for short transactions and in workloads with little contention. We would thus like to combine the efficient and simple approach of single-clock TBTMs with a more involved mechanism for long transactions, without the need to keep multiple versions.

5 A Pragmatic Approach: z-linearizability

In this section we introduce a novel way to address the problem of how to increase the chances that long transactions can commit. The problem is that in TBTMs, long transactions that access a large number of objects have typically a higher likelihood of interfering with other transactions (e.g., T_{L1} interferes with T_1 and T_2 in Figure 4). Moreover, when a short transaction (e.g., T_5 or T_6) updates objects that are read by a long transaction and the short transaction commits first, this will result in the abort of the long transaction.

To illustrate our approach, consider the scenario depicted in Figure 4. Long transaction T_{L1} accesses all objects (e.g., all accounts of a bank) while the remaining transactions only access a small number of objects (e.g., two accounts in case of a transfer between two accounts). To permit T_{L1} to commit and to guarantee serializability, the system would need to abort at least transactions T_1 and T_2 because neither T_1 and T_{L1} nor T_2 and T_{L1} can be ordered. To permit T_{L1} to commit and to guarantee linearizability, one would additionally need to

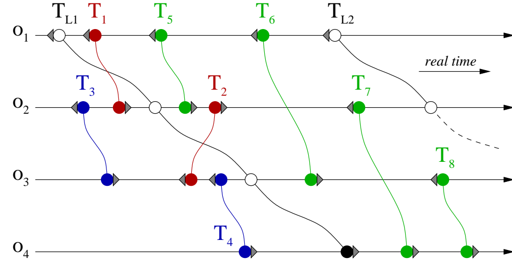

Figure 4: Long transactions have typically a higher chance of interfering with other transactions. Since the first committer wins, a commit by any of the short transactions T_5 and T_6 will result in the abort of T_{L1}.

abort T_4 or T_5 because linearizability requires T_5 to be ordered before T_4 but T_{L1} must be ordered after T_4 and before T_5.

Figure 4 indicates that during long transactions we need either to delay or abort some short transactions to enforce linearizability. A practical alternative is to slightly weaken linearizability during a long transaction. We propose to permit an ordering of short transactions that can violate the real time ordering of transactions but only while a long transaction is executing. To explain this, consider for a moment that two long transactions T_{L1} and T_{L2} access all objects (e.g., they calculate the sum of all accounts). Our basic assumption is that long transactions are quite infrequent in comparison to short transactions. Hence, long transactions partition short transactions into different "time zones" (see Figure 5).

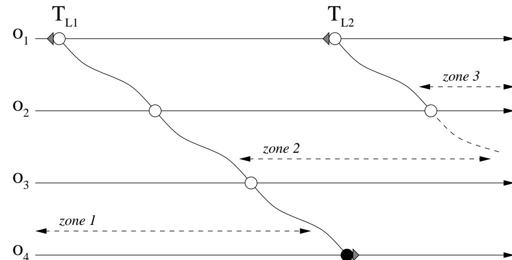

Figure 5: Long transactions partition short transactions into zones. Transactions in each zone are linearizable while the set of all transactions is only serializable.

All transactions in the individual zones are linearizable and all transactions are serializable. More precisely, the semantics (we call this *z-linearizability*) that we want to enforce is (1) that the set of long transactions is linearizable, (2) the

set of short transactions between two long transactions is linearizable, (3) the set of all transactions is serializable, and (4) the serialization order observes the sequential execution ordering of the individual threads. This is a weakening of linearizability in the sense that we permit some short transactions to violate linearizability while a long transaction is in progress. Property (4) states that a thread cannot cross an active long transaction "backwards", e.g., in Figure 4 a thread t could execute T_3 and then T_5 but not T_5 and then T_4 because the latter would cross the path of T_{L1} backwards.

5.1 Algorithm

To implement a z-linearizable STM, we use a slightly extended LSA for short transactions and long transactions are ordered with the help of a logical clock. The idea is that each long transaction T reserves a unique logical clock value $T.zc$ with the help of a global counter (which we call the zone counter) ZC. Long transactions need to commit in the order of their unique timestamps (i.e., $T.zc$). Since we assume that long transactions are executed infrequently, we can enforce that with the help of a simple commit counter CT: a long transaction T can only commit if its logical time is greater than the current commit time, i.e., $T.zc > CT$, and it then sets CT atomically to $T.zc$. This implies that all active transactions have a logical time that is bounded by the interval $AI = (CT, ZC]$.

Conflicts between long transactions are resolved with the help of logical time $o.zc$ attached to each object o. When a long transaction T opens an object o, it sets $o.zc$ atomically to $T.zc$ if $o.zc < T.zc$. If T was "passed" by a long transaction T_2 with a higher zone number, i.e., $T_2.zc > T.zc$ and $o.zc = T_2.zc$, then we abort T.

We assume that a transaction opens each object that it accesses exactly once. When opening an object in read mode, a reference to the current version of the object is returned. This object will not change because when an object is opened in write mode, a private copy of the object is created and returned to the transaction. Write accesses are visible, i.e., write/write conflicts are detected on open and result in one of the involved transactions being delayed or aborted by the contention manager.

Committing of a long transaction is straightforward. A long transaction just atomically flips its status from "active" to "committed" if its zone number $T.zc$ is greater than the current commit time CT. Otherwise, the transaction has to abort. Such a simple check is sufficient because any read or write conflict with another long transaction is detected via the logical timestamps $o.zc$ of the objects and leads to an abort or delay of one of the involved transactions.

5.2 Short Transactions

The percentage of short transactions is expected to be substantially higher than that of long transactions. To improve scalability, we can use a LSA that uses real-time stamps. This permits us to parallelize the time base through the use of synchronized real-time clocks [9] (and avoids in this way the contention on shared commit time counter).

To enforce z-linearizability, we need to detect and abort short transactions that would "cross" the path of a long transaction (e.g., like transactions T_1 and T_2 in Figure 4). The detection is implemented as follows. Each short transaction T has a logical timestamp $T.zc$. Unlike for a long transaction, $T.zc$ is set when T opens the first object. This means that the first object that T opens determines its zone. We have to detect when a transaction crosses from one zone into another zone (unless the transactions that defined these zones have already committed). More precisely this can be achieved as follows.

When a short transaction T opens the ith object o_i ($i > 1$), it compares its timestamp $T.zc$ with that of the opened object $o_i.zc$. If they are in the same zone, T can proceed. If they are different but both belong to transactions that are not active anymore (i.e., $o_i.zc \leq CT$ and $T.zc \leq CT$), T can also proceed because T cannot interfere with any of these long transactions. If however the transaction might still be active (i.e., $o_i.zc \in AI$ or $T.zc \in AI$) we call the contention manager, which would typically abort T.

The decision of whether a transaction can commit is performed by the underlying LSA algorithm, i.e., the detection of a short transaction crossing a long transaction is entirely performed during the open of

objects.

5.3 Implementation Details

The pseudo code for implementing z-linearizability is depicted in Algorithms 2 and 3. The basic prerequisite is that we classify transactions as either *long* or *short*: a long transaction calls the functions depicted in Algorithm 2 while a short transaction the ones depicted in Algorithm 3. The class must be known at the start of a transaction. In the simplest case, the programmer might need to mark explicitly transactions that are long. However, an automatic marking based on past behaviors of transactions would be a viable alternative.

Algorithm 2 Z-STM for long transactions (algorithm of thread p)

```
 1: procedure START_long(T)                      ▷ Start long transaction
 2:     T.state ← active
 3:     T.zc ← ZC++
 4: end procedure

 5: procedure OPEN_long(T, o_i, m)               ▷ T opens o_i in mode m
 6:     if o_i.zc < T.zc then
 7:         o_i.zc ← T.zc
 8:         if o_i.writer ≠ null then
 9:             arbitrate(T, o_i.writer)         ▷ CM resolves conflict
10:             o_i.writer ← null                ▷ T won
11:         end if
12:         if m = write then                    ▷ T updates o_i
13:             o_i.writer ← T
14:             o_i.next ← DUPLICATE(CURRENT(o_i))
15:             return o_i.next
16:         else                                 ▷ T reads o_i
17:             return CURRENT(o_i)
18:         end if
19:     else                     ▷ Transaction with higher zc beats us
20:         ABORT(T)
21:     end if
22: end procedure

23: procedure COMMIT_long(T)                    ▷ Try to commit transaction
24:     if T.state = active ∧ T.zc > CT then
25:         T.state ← committed
26:         CT ← T.zc
27:         LZC_p ← T.zc          ▷ Remember last zone committed in
28:     else
29:         T.state ← aborted
30:     end if
31: end procedure
```

The pseudo code assumes access to an implementation of LSA (e.g., see [9]), which is called when a short transaction T starts, opens an object, commits, or aborts. For simplicity, we assume that methods execute atomically in the pseudocode. Atomicity is implemented with the help of compare-and-swap operations and indirect accesses to shared objects via locators, as in [4].

Algorithm 3 Z-STM for short transactions (algorithm of thread p)

```
 1: procedure START_short(T)                     ▷ Start short transaction
 2:     T.zc ← 0
 3:     START_LSA(T)
 4: end procedure

 5: procedure OPEN_short(T, o_i, m)
 6:     if T.zc = 0 then                         ▷ Opening first object?
 7:         if o_i.zc < LZC_p then               ▷ From old zone?
 8:             if LZC_p > CT then               ▷ Zone still active?
 9:                 ABORT(T)                     ▷ Cannot move to past zone
10:             else
11:                 T.zc ← CT
12:             end if
13:         else
14:             T.zc ← o_i.zc
15:         end if
16:     else if T.zc ≠ o_i.zc then              ▷ Different zones?
17:         if T.zc > CT ∨ o_i.zc > CT then      ▷ Zones still active?
18:             conflict(T, o_i.zc)              ▷ CM delays/aborts T
19:         else
20:             T.zc ← CT
21:         end if
22:     end if
23:     OPEN_LSA(T, o_i, mode)
24: end procedure

25: procedure COMMIT_short(T)                   ▷ Try to commit transaction
26:     COMMIT_LSA(T)
27:     if T.state = committed then
28:         LZC_p ← T.zc          ▷ Remember last zone committed in
29:     end if
30: end procedure
```

5.4 Discussion

We discuss briefly why Algorithms 2 and 3 ensure z-linearizability. First, we need to explain why the set of committed long transactions is linearizable. Informally, we need to sketch why any long transaction T takes effect atomically somewhere between the start and the end of the transaction. The way this is achieved is that a long transaction uses a consistent snapshot of all objects it accesses. The snapshot of the objects is created by opening the objects and by making sure that the zone counter ($o.zc$) of all objects is strictly monotonic, i.e., all long transactions access objects in the order of their zone counter (i.e., $T.zc$) or get aborted. Short transactions cannot "cross" a transaction that is still active (see Algorithm 3 lines 16-22). We use visible writes to detect write/write conflicts and updates become visible to other transactions when the up-

date transaction's status changes from "active" to "committed" (Algorithm 2, line 25). In this way, a long transaction takes atomically effect when the status changes to committed.

Second, we need to sketch why short transactions sandwiched between two long transactions are linearizable. Such short transactions use LSA [8] to make sure that that their read and write snapshot is consistent at the time a short transaction commits. This ensures linearizability.

Third, the set of all transactions is serializable. This follows from the property that all long transactions are linearizable and all short transactions belonging to the same zone are linearizable. Note that a short transaction T can access objects belonging to different zones as long as T and all the objects T accesses belong (at the time of the open) to a zone that is already in the past (Algorithm 3: test on line 17). LSA ensures that at the time T accessed the last object (and until T commits), T's snapshot of all accessed objects is consistent. This means that T can be ordered between the last long transaction that committed before T accessed its last object and the next long transaction that commits.

Fourth, we explain why the serialization order observes local execution order. This is enforced with the help of a thread local variable LZC. This variable keeps track of the last zone in which the thread committed either a long or a short transaction. Long transactions will always commit with a new maximum zone number (see Algorithm 2: test on line 24) and this will be used to update LZC (Algorithm 2 line 27). In short transactions, we make sure that a thread only accesses objects with a zone counter of at least LZC (Algorithm 3 line 7) unless all long transactions with a zone counter $\leq LZC$ have already committed (Algorithm 3 line 8). In the latter case, we can safely assume that the short transaction executes at the current CT (Algorithm 3 lines 11 and 20) because the underlying LSA enforces a consistent snapshot.

5.5 Performance Evaluation

In what follows, we highlight the usefulness and advantages of z-linearizability by showing performance results for a simple bank micro-benchmark. This benchmark 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 (Compute-Total). Whereas the former transaction is a small update transactions, the latter is a long transaction and treated as such. Compute-Total has two variants: a read-only transaction, or an update transactions that write to private but transactional state. There are 1,000 accounts. One of the threads performs both transfers (with 80% probability) and balance computations (with 20% probability), while the other threads only execute transfers. We executed the benchmark on an 8-core UltraSparc T1 system, which runs four threads concurrently per core. The STMs are Java prototypes, LSA-STM is from [8], and Z-STM is LSA-STM adapted as described in Section 5. We used Sun's HotSpot Java VM 1.5 running on Solaris.

Figure 6 shows that with read-only Compute-Total transactions, Z-STM and LSA-STM perform very similar, and the overhead of updating and checking the per-object zone counters is negligible on our system. Z-STM performs Compute-Total faster than LSA-STM because the latter always maintains read sets. An optimized version of LSA-STM that detects when read sets are not required is as fast as Z-STM. The slight decline of throughput when the number of threads increases is due to several threads sharing one CPU core.

However, once Compute-Total transaction become update transactions, LSA-STM is not able to execute them anymore because the probability that an account is updated during the runtime of the long transaction is very high. Because of invisible reads, Compute-Total transactions cannot easily prevent or delay transfers. In contrast, Z-STM is able to sustain the throughput, as can be seen in Figure 7. Note also that the transfer throughput does not decrease as compared to LSA-STM. Traditional visible reads would require that transfers wait for the long transaction to complete; in Z-STM, transfers can update an object right after the long transaction has completed its read access.

6 Conclusion

The linearizability semantics supported by time-based transactional memories (TBTMs) are simple to reason about for the developers and allow for efficient and scalable implementations. Yet, enforcing linearizability also has the negative effect of restricting concurrency under some common workloads, notably when long transactions compete with many short transactions.

We have investigated the use of semantics weaker than linearizability as a mean to increase the throughput of long transactions. *Causal serializability* is sufficient for a wide range of applications and can be easily implemented using vector clocks (or plausible clocks). However, we have also observed in our prototype that the runtime overhead for managing vector time can be quite significant. *Serializability* is strong enough for many applications but is hard and costly to fully support as it requires maintaining a partial precedence graph at runtime.

We have therefore proposed a novel consistency criterion, which we call *z-linearizability*, that offers a good trade-off between strong semantics and good performance. Long transactions partition short transactions into different zones. Linearizability is guaranteed among long transactions and within a zone, while the whole set of transactions is serializable. We have observed a significant increase in the throughput of long transactions using z-linearizability in our experiments. Some of the throughput gains of Z-STM can be attributed to the following factors: (1) Z-STM does neither have to maintain a read set nor a write set for long transactions, (2) long transactions can commit with a very simple and efficient validation test, (3) Z-STM does not need visible reads for short transactions, and (4) Z-STM does not, in particular, need to keep multiple versions (like snapshot isolation) to get good performance for long transactions.

References

[1] 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 SIGMOD*, pages 1–10, 1995.

[2] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[3] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–666, 1988.

[4] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2003.

[5] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[6] F. Mattern. Virtual Time and Global States of Distributed Systems. In *International Workshop on Parallel and Distributed Algorithms*, 1988.

[7] M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *Proceedings of the 23rd EUROMICRO Conference*, 1997.

[8] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[9] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with Scalable Time Bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

[10] M. F. Spear, V. J. Marathe, W. N. S. III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *20th Intl. Symp. on Distributed Computing (DISC)*, 2006.

[11] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2), June 1979.

[12] F. J. Torres-Rojas and M. Ahamad. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. *Distributed Computing*, 12(4):179–195, 1999.

[13] 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 *CGO*, 2007.

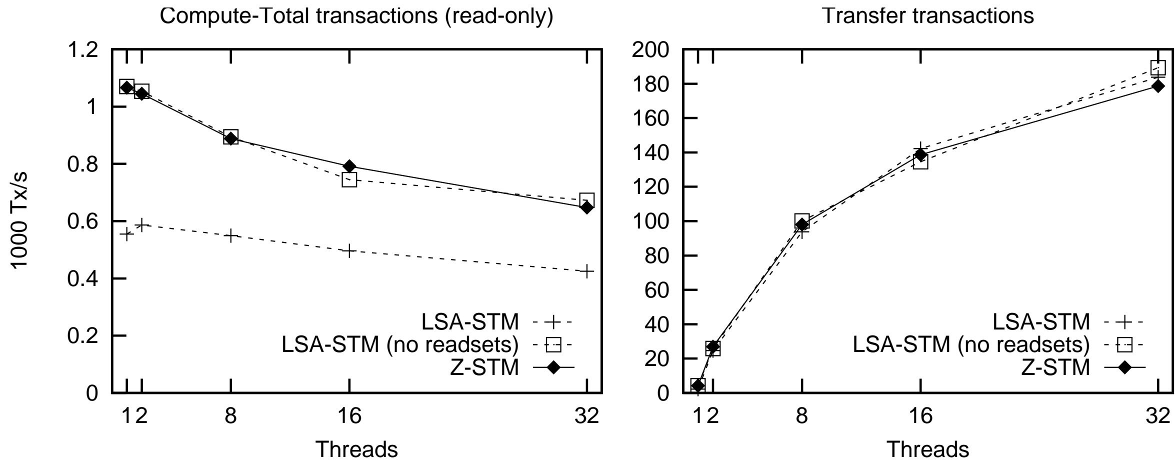

Figure 6: Throughput results for the Bank benchmark with read-only Compute-Total transactions.

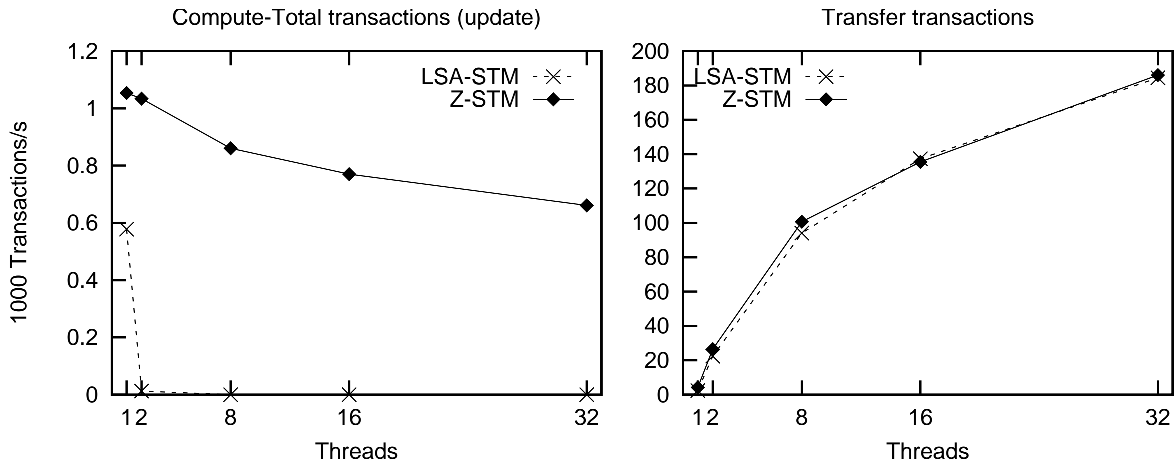

Figure 7: Throughput results for the Bank benchmark with Compute-Total transactions that update state.