Time-based Transactional Memory with Scalable Time Bases

Torvald Riegel
Dresden University of Technology, Germany
torvald.riegel@inf.tu-dresden.de

Christof Fetzer
Dresden University of Technology, Germany
christof.fetzer@tu-dresden.de

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

ABSTRACT

Time-based transactional memories use time to reason about the consistency of data accessed by transactions and about the order in which transactions commit. They avoid the large read overhead of transactional memories that always check consistency when a new object is accessed, while still guaranteeing consistency at all times—in contrast to transactional memories that only check consistency on transaction commit.

Current implementations of time-based transactional memories use a single global clock that is incremented by the commit operation for each update transaction that commits. In large systems with frequent commits, the contention on this global counter can thus become a major bottleneck.

We present a scalable replacement for this global counter and describe how the Lazy Snapshot Algorithm (LSA), which forms the basis for our LSA-STM time-based software transactional memory, has to be changed to support these new time bases. In particular, we show how the global counter can be replaced (1) by an external or physical clock that can be accessed efficiently, and (2) by multiple synchronized physical clocks.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Performance

Keywords

Transactional Memory

1. INTRODUCTION

Over the last few years, major chip manufacturer have shifted their focus from ever faster clock speeds to increased parallel processing capabilities. With the advent of multi-core CPUs, application programmers have to develop multi-threaded programs in order to harness the parallelism of the underlying CPUs. Yet, writing *correct* and *efficient* concurrent programs is a challenging task. Conventional synchronization techniques based on locks are unlikely to be effective in such an environment: coarse-grained locks do not scale well whereas fine-grained locks introduce significant complexity, are not composable, and can lead to such problems as deadlocks or priority inversions.

Transactional memory (TM) was proposed as a lightweight mechanism to synchronize threads by optimistic, lock-free transactions. 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.

Software transactional memory (STM) has therefore recently gained a lot of interest, not only in the research community (e.g., [5, 7]) but also in industry. It offers a familiar transactional API at the programming language level to help isolate concurrent accesses to shared data by multiple threads. Besides isolation, STMs guarantee atomicity of the sequential code executed within a transaction: in case the transaction cannot commit, any modification performed to shared data is automatically undone.

In STMs there is currently a trade-off between consistency and performance. Several high-performance STM implementations [11, 6, 3] use optimistic reads in the sense that the set of objects read by a transaction might not be consistent. Consistency is only checked at commit time, i.e., commit *validates* the transaction. However, having an inconsistent view of the state of the objects during the transactions might prevent the application to run properly (e.g., it might enter an infinite loop or throw unexpected exceptions).

On the other hand, validating after every access can be costly if it is performed in the obvious way, i.e., checking every object previously read. Typically, the validation overhead grows linearly with the number of objects a transaction has read so far.

A *time-based transactional memory* uses the notion of time to reason about the consistency of data accessed by transactions and about the order in which transactions commit. This allows a TM to always access consistent objects (as if validating after every read) without incurring the cost of validation. In [9, 10], we have proposed a time-based STM that uses invisible reads but guarantees consistency

SPAA'07, June 9–11, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-667-7/07/0006 ...$5.00.

by maintaining an always consistent snapshot for transactions. Other groups have proposed time-based STMs since then [2, 12]. However, the time bases that are currently used in these STMs are very simple because they all rely upon shared counters (with some optimizations). As the number of concurrent threads grows, the shared counter can quickly become a bottleneck: update transactions typically update the counter, which results in cache misses for all concurrent transactions. There is therefore a dire need for more scalable time bases.

1.1 Time-based Transactional Memory

A transactional memory consists of a set of shared objects that can be accessed concurrently by multiple processors or threads. Using time as the basis for the transactional memory does not impose a certain implementation in general: both object-based and word-based STMs, as well as hardware transactional memories, can be used. However, timing information has to be stored at each object. In what follows, we do not assume a specific implementation.

Time-based transactional memory uses a time base to impose a total order on transactions and object versions. In its simplest form, the time base can be a simple integer counter shared by all processors, but the focus of this paper is on more scalable time bases. Furthermore, the clocks of time bases can either tick when required by the transactional memory (i.e., they work like counters) or independently of the transactional memory (e.g., as real-time clocks do).

Transactions are either *read-only*, i.e., they do not modify any object, or *update* transactions, i.e., they write one or more objects. Each object traverses a sequence of versions as it gets modified by update transactions. A new version becomes valid at the time a transaction that updated the object commits. We call the *validity range* $v.R$ of an object version v the interval between the time a version becomes valid and the time it gets superseded by another version. The lower and upper bounds of the validity range $v.R$ are referred to by $\lfloor v.R \rfloor$ and $\lceil v.R \rceil$. A version that is still valid has an upper bound set to ∞. Even though different processors may have distinct time references, the value of the commit times of transactions are agreed upon and so are the validity ranges of objects. Note however that while the values are agreed upon, a thread might only be able to approximate a remote timestamp ts with its own local clock, e.g., it might be able to compute that ts was read between local timestamps ts_1 and ts_2.

A transaction T accesses a finite set of object versions, denoted by $T.O$. We assume that objects are only accessed and modified within transactions. We define the *validity range* $T.R$ of transaction T to be the time range during which all object versions in $T.O$ are valid, that is, the intersection of the validity ranges of the individual versions. $\lfloor T.R \rfloor$ and $\lceil T.R \rceil$ denote the lower and upper bounds of $T.R$. We say that the object versions accessed by transaction T are a *consistent snapshot* if the validity range $T.R$ is non-empty.

The transactional memory has to read the current time at the beginning of each transaction to make sure that transactions are linearizable. Another access to the time base is usually performed when update transactions are committed. Whether further accesses are performed depends on the specific implementation of the transactional memory.

A transaction T can try to *extend* $T.R$ to avoid that $T.R$ becomes empty. When T accesses the most recent versions of the objects in $T.O$, the upper bound of its validity range is set to the current time (and not ∞) because T cannot predict if or when objects will be changed. When extending $T.R$, it is checked whether some objects have been changed. If not, the validity range can be extended to the current time (i.e., the time before the check was performed). Otherwise, $\lceil T.R \rceil$ will be set to the earliest commit time of any of the object versions that replace a version in $T.O$.

If time has progressed since the start of a transaction (which indicates that the state of the transactional memory has progressed), one extension has to be performed when committing an update transaction (see Section 2). Existing implementations of time-based transactional memory differ mostly in the implementation of the time base, the maintenance of object versions, and the computation of validity ranges, as described next.

1.2 Related Work

In what follows, we give an overview of transactional memory implementations that benefit from using some notion of time or progress in the system.

The Lazy Snapshot Algorithm (LSA) [9] uses time in the way described in Section 1.1, with an integer counter shared between all threads as time base. The shared counter is read on every start of a transaction and incremented when committing update transactions. It is also read when LSA-STM tries to extend a transaction's validity range, but that is optional. The STM does not need to access the counter during ordinary accesses because information in the validity ranges of object versions and the transaction is sufficient. The overhead of this counter is negligible on current multicore CPUs in which sharing data is inexpensive. On larger systems, however, the counter becomes a bottleneck.

Transactional Locking II (TL2) [2] uses time in a similar way 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 range extensions are performed (except when committing update transactions). Thus, an object can only be read if the most recent update to the object is before the start time of the current transaction. A shared integer counter is used as time base together with an optimization that lets transactions share commit timestamps when the timestamp-acquiring $C\&S$ operation[1] fails. It is also suggested to use hardware clocks instead of the shared counter to avoid its overhead.

In [13], Wang *et al.* describe compiler support for a time-based STM that uses an algorithm similar to the one presented in [9] but also maintains only a single object version. A shared integer counter is used as time base.

RSTM [12] is an object-based STM that uses single-version objects and validation (i.e., it checks on each access that every object previously read has not been updated concurrently). To reduce the validation overhead, a heuristic is used: a global "commit counter" counts the number of attempted commits of update transactions in the system (i.e., the counter shows whether there was progress) and the set of accessed objects is validated only if there was progress. This heuristic is less effective than the methods used by [9, 2, 13]. Using a time base that can be efficiently read is very important when using this heuristic because the counter has

[1] The *compare-and-swap* operation $C\&S(v, e, n)$ atomically stores a new value n into variable v if v contains the expected value e.

to be read on every access to an object. Thus, even disjoint updates will lead to cache misses, slowing down transactions that are never affected by these updates.

1.3 Contributions and Organization

In this paper, we study the concept of time-based transactional memory and explore two options for providing scalable time bases. We introduce a novel STM algorithm that uses perfectly synchronized real-time clocks to optimistically synchronize concurrent transactions. We then extend our algorithm so that it can also support imprecise clocks (e.g., externally synchronized clocks). Experimental results tend to confirm that using real-time does indeed improve scalability on large systems.

The rest of the paper is organized as follows: Section 2 describes the real-time extension of our consistent snapshot construction algorithm for transactional memories. In Section 3, we explain how our algorithm can use perfectly synchronized and imprecise real-time clocks. Section 4 presents experimental results that show the scalability of our approach as compared to accessing a shared counter. Finally, Section 5 concludes the paper.

2. REAL-TIME LAZY SNAPSHOT ALGORITHM

In what follows, we introduce the LSA-RT algorithm, an extension of LSA [9], that does not assume a global shared counter as time base.

2.1 Time Bases

In this paper, we use two time bases: (1) perfectly synchronized clocks and (2) externally synchronized clocks. Perfectly synchronized clocks give (conceptually) all threads access to one global clock without any reading error. The reading error is the difference between the value read and the correct value. Typically, such a perfectly synchronized clock would need to be implemented in hardware. An externally synchronized clock also provides access to a global clock but with some reading error that might vary and might not be bounded.

In both cases, the global clock does not actually need to be a real-time clock, i.e., neither its speed nor its value needs to be approximately synchronized with real-time. However, having a global real-time clock typically simplifies the implementation of an externally synchronized clock (because local clocks with a bounded drift rate can be used to approximate real-time). In particular, this reduces the overhead and error if the synchronization is done in software.

We use several utility functions whose implementation depends on the time base that is used (see Algorithm 1): getTime, getNewTS, $t_1 \succcurlyeq t_2$ (guaranteed later than or equal), $t_1 \succsim t_2$ (possibly later than), `max` and `min`. We first focus on the semantics of these utility functions. Their implementation will be described later together with their respective time base.

Function getTime returns the current time. We assume that the timestamps that a thread is reading are monotonic, i.e., if a thread reads first t_1 and then t_2, then we know that t_2 is guaranteed to be later or equal to t_1. In our terminology, we will denote this by $t_2 \succcurlyeq t_1$. We do not require t_2 to be strictly later than t_1 because we want to support clocks that tick rarely, e.g., only when a transaction commits. If a thread needs to read such a fresh timestamp, it needs to call function getNewTS. This function ensures that the value returned to a thread is strictly greater than any timestamp that has so far been returned to this thread by getNewTS or getTime. Note that the timestamps returned by getTime and getNewTS are not necessarily unique: other threads might read the same timestamps.

Algorithm 1 Generic utility functions

```
use function getTime()                  ▷ Get current timestamp
                                        ▷ (module-specific)
use function getNewTS()                 ▷ Get strictly greater timestamp
                                        ▷ (module-specific)
use function $\succcurlyeq(t_1, t_2)$   ▷ Guaranteed later than or equal
                                        ▷ (module-specific)

function $\succsim(t_1, t_2)$           ▷ Possibly later than
    return $t_2 \not\succcurlyeq t_1$
end function
use function max$(t_1, t_2)$            ▷ Maximum (module-specific)
use function min$(t_1, t_2)$            ▷ Minimum (module-specific)
```

We assume that getTime and getNewTS return a clock value that was read instantaneously at some point in real time between the time getTime/getNewTS was called and the time it returned. If two timestamps are read by different threads, it might not be possible to say which timestamp was read later or earlier (e.g., because there is a non-zero clock reading error). To cope with this uncertainty, we say that $t_2 \succcurlyeq t_1$ iff it is guaranteed that t_1 was read no later than t_2. Sometimes we might only be able to say that t_2 was *possibly* read at a later point than t_1. We denote this by $t_2 \succsim t_1$. Note that $t_2 \succcurlyeq t_1$ always implies $t_1 \not\succsim t_2$ and $t_2 \succsim t_1$ implies $t_1 \not\succcurlyeq t_2$.

The utility functions `max`(t_1, t_2) and `min`(t_1, t_2) have the following semantics. For any timestamp t_3, if t_3 is guaranteed to be later than `max`(t_1, t_2) then t_3 is guaranteed to be later than both $t1$ and $t2$. Similarly, for any timestamp t_3 that is guaranteed to be earlier than `min`(t_1, t_2), then t_3 is guaranteed to be earlier than both t_1 and t_2.

2.2 Snapshot Construction

The main idea of LSA-RT (see Algorithm 2) is to construct consistent snapshots on the fly during the execution of a transaction and to *lazily* extend the validity range on demand. By this, we can reach two goals. First, transactions working on a consistent snapshot always read consistent data. Second, verifying that there is an overlap between the snapshot's validity range and the commit time of a transaction can ensure linearizability, if so desired.

The set of objects being accessed by a transaction and their specific versions are determined during the execution of a transaction. The validity range $T.R$ is therefore constructed incrementally. When a transaction T is started, the lower bound of its validity range is set to the current time (line 3), i.e., the transaction cannot execute in the past. The getTime function returns the current time—as observed by the current thread—according to the time base being used. The timestamps returned by the function to any single thread are guaranteed to be monotonically increasing, but not strictly (a thread may read more than once the same timestamp).

When accessing the most recent version of an object o, it is not yet known when this version will be replaced by a new version. We therefore obtain an approximate validity range

Algorithm 2 Real-Time Lazy Snapshot Alg. (LSA-RT)

```
 1: procedure START($T$)                     ▷ Initialize transaction attributes
 2:   $T.CT \leftarrow 0$                                  ▷ $T$'s commit time
 3:   $T.R \leftarrow [\text{GETTIME}(), \infty]$                ▷ $T$'s validity range
 4:   $T.O \leftarrow \varnothing$                ▷ Set of objects versions accessed by $T$
 5:   $T.update \leftarrow$ false           ▷ $T$ starts as a read-only transaction
 6:   $T.status \leftarrow$ active                             ▷ $T$ is active
 7: end procedure

 8: procedure OPEN($T, o, m$)     ▷ Opens $o$ in mode $m$ (read/write)
    ▷ To simplify, we assume an object is opened at most once per $T$
 9:   if $m =$ write then
10:     $T.update \leftarrow$ true
11:     repeat
12:       $v_c \leftarrow \text{GETVERSION}(T, o, [\lfloor T.R \rfloor, \infty])$
                                        ▷ Get latest committed version
13:       $v \leftarrow clone(v_c)$                ▷ Create new copy for writing
14:       $v.T \leftarrow T$                     ▷ Current transaction is writer
15:       $T_w \leftarrow o.writer$
16:       if $T_w \neq null \wedge T_w.status \notin \{$aborted, committed$\}$ then
17:         $solveConflict(o, T, T_w)$         ▷ Contention manager ...
18:       else                         ▷ ...arbitrates and aborts the loser
19:         $C\&S(o.writer, T.w, T)$          ▷ Try registering as writer
20:       end if
21:     until $o.writer = T$
22:     if $\lfloor v.R \rfloor \succsim \lceil T.R \rceil$ then            ▷ Is the version too recent?
23:       EXTEND($T$)                        ▷ Extend as much as possible
24:     end if
25:   else
26:     $v \leftarrow \text{GETVERSION}(T, o, T.R)$
                                 ▷ Get latest committed version in interval
27:   end if
28:   $\lfloor T.R \rfloor \leftarrow \max(\lfloor T.R \rfloor, \lfloor v.R \rfloor)$
29:   $\lceil T.R \rceil \leftarrow \min(\lceil T.R \rceil, \text{GETPRELIMUB}(T, o, v, \lceil T.R \rceil))$
30:   if $\lfloor T.R \rfloor \succsim \lceil T.R \rceil$ then                ▷ Possibly inconsistent?
31:     ABORT($T$)               ▷ Yes: abort (and terminate execution)
32:   end if
33:   $T.O \leftarrow T.O \cup \{(o, v)\}$                  ▷ Access object versions
34: end procedure

35: procedure COMMIT($T$)                      ▷ Try to commit transaction
36:   if $\neg T.update$ then
37:     $C\&S(T.status,$ active, committed$)$
                                          ▷ Validation not necessary
38:   else
39:     $C\&S(T.status,$ active, committing$)$     ▷ Start committing
40:     if $T.status =$ committing then
41:       $t \leftarrow \text{GETNEWTS}()$
                          ▷ Tentative commit time (may not be unique)
42:       $C\&S(T.CT, 0, t)$                ▷ Try imposing our timestamp
43:       for all $(o, v) \in T.O$ do      ▷ Are versions still valid at $t$?
44:         $ub \leftarrow \text{GETPRELIMUB}(T, o, v, T.CT)$
45:         if $T.CT \succsim ub$ then
46:           ABORT($T$)         ▷ No: abort (and terminate execution)
47:         end if
48:       end for
49:       $C\&S(T.status,$ committing, committed$)$▷ Yes: commit
50:     end if
51:   end if
52: end procedure

53: procedure ABORT($T$)    ▷ Abort transaction (unless committed)
54:   if $\neg C\&S(T.status,$ active, aborted$)$ then           ▷ Still active?
55:     $C\&S(T.status,$ committing, aborted$)$           ▷ Committing?
56:   end if
57:   if $T.status =$ aborted then                              ▷ Aborted?
58:     throw $AbortedException$ in $T$           ▷ Terminate execution
59:   end if
60: end procedure
```

r by obtaining the latest version of o (line 12) and computing a lower bound on its maximum validity range. We call this the *preliminary upper bound* on the validity range (see line 29). Note that we use the GETPRELIMUB function to recompute the preliminary upper bound of an object version according to the current thread's time reference. During the execution of a transaction, time might advance and thus the preliminary validity ranges might get longer. We can try to *extend* $T.R$ by recomputing its lower bound (line 23 of Algorithm 2 and lines 1–6 in Algorithm 3). Extensions are not required for correctness, but they increase the chance that a suitable object version is available. To avoid unnecessary extensions, we mark a transaction as closed as soon as EXTEND detects that T has read an object version that has in meantime be replaced by a new version, i.e., no further extension of the validity interval $T.R$ is possible. For simplicity, we have not included this optimization in the pseudo-code.

Algorithm 3 Helper functions

```
 1: procedure EXTEND($T$)
                    ▷ Try to extend $T$'s validity range to at least $t$
 2:   $\lceil T.R \rceil \leftarrow \text{GETTIME}()$
 3:   for all $(o, v) \in T.O$ do
                        ▷ Recompute the upper bound on validity range
 4:     $\lceil T.R \rceil \leftarrow \min(\lceil T.R \rceil, \text{GEPRELIMTUB}(o, v, \lceil T.R \rceil))$
 5:   end for
 6: end procedure

 7: function GETVERSION($T, o, R$)
                            ▷ Get latest version of $o$ overlapping $R$
 8:   loop
 9:     $v \leftarrow$ latest version of $o$ s.t. $\lceil v.R \rceil \succcurlyeq \lfloor R \rfloor \wedge \lceil R \rceil \succcurlyeq \lfloor v.R \rfloor \wedge$
          $(v.T =$ null $\vee\ v.T.status \in \{$committing, committed$\})$
10:     if $v =$ null then                          ▷ Any valid version?
11:       ABORT($T$)           ▷ No: abort (and terminate execution)
12:     else if $v.T \neq$ null $\wedge\ v.T.status =$ committing then
13:       COMMIT($v.T$) ▷ Help committing transaction to complete
14:     else
15:       return $v$              ▷ Always return a commited version
16:     end if
17:   end loop
18: end function

19: function GETPRELIMUB($T, o, v, t$)
                           ▷ Get conservative estimate on $\lceil v.R \rceil$
20:   $T_w \leftarrow o.writer$
21:   if $\lceil v.R \rceil \neq \infty$ then                             ▷ Still open?
22:     return $\lceil v.R \rceil$              ▷ No: return version upper bound
23:   else if $T_w \neq$ null then   ▷ Yes: only $T_w$ may set UB before $t$
24:     if $T_w.status \in \{$committing, committed$\}$ then
25:       if $T_w.CT > 0$ then
26:         if $T_w = T$ then
27:           return $T_w.CT$          ▷ Off by 1 but simplifies COMMIT
28:         else
29:           return $T_w.CT - 1$▷ Version valid at least until then
30:         end if
31:       end if
32:     end if
33:   end if
34:   return $t$          ▷ Return caller's timestamp (GETTIME() $\succcurlyeq t$)
35: end function
```

If the validity range r of the latest version of o does not intersect with $T.R$ and the transaction is read-only, we can look for an older version whose range overlaps with $T.R$ (the algorithm requests the most recent among the valid overlapping versions, but any of them would do). The new value of $T.R$ is computed as the intersection of the previous value and the validity range of the version being accessed (lines 28–29). The transaction must abort if no suitable version can be found (line 31 of Algorithm 2 and line 11 of Algorithm 3).

By construction of $T.R$, LSA-RT guarantees that a transaction started at time t has a snapshot that is valid at or after the transaction started, i.e., $\lfloor T.R \rfloor \succcurlyeq t$. Hence, a read-only transaction can commit iff it has used a consistent snapshot, i.e., $T.R$ is non-empty.

2.3 Update Transactions

An update transaction T can only commit if it can extend its validity range up to and including its commit time. This ensures that at the time T commits no *other* transaction has modified any of these objects including at the commit time. Note that in this way, we permit multiple transactions to commit at the same time as long as they are not in conflict with each other. The preliminary upper bound of an object version written to by T is overestimated by 1 (line 27 in Algorithm 3) to simplify the test in Commit(): we know that T will try to commit a new version o at $T.CT$ but, more importantly, we also know that no other transaction can commit a new version of o until $T.CT+1$ if T can indeed commit.

The commit of an update transaction (lines 35–52) is a two-phase process. The transaction first enters the *committing* state before determining whether it can commit or must abort. The reason for keeping track of the transaction's status and updating it using a $C\&S$ operation is that another thread can help the transaction to commit or force it to abort, as will be discussed shortly.

A committing thread will try to set the timestamp obtained from its local time reference as the commit time of the transaction. If it fails, i.e., another thread has set the commit time beforehand, then the current thread uses that previously set commit time $T.CT$. The thread will then check whether the upper bound of the validity range of the transaction can be extended to include $T.CT$. The transaction can only commit if this succeeds because otherwise some objects accessed by T might have been modified by another transaction that committed before $T.CT$.

If it is possible to update a most recent version (i.e., $T.R$ remains non-empty), LSA-RT atomically marks the object o that it is writing (visible write) by registering itself in $o.writer$. When another transaction tries to write the same object, it will see the mark and detect a conflict (lines 16-17). In that case, one of the transactions might need to wait or be aborted. This task is typically delegated to a *contention manager* [7], a configurable module whose role is to determine which transaction is allowed to progress upon conflict; the other transaction will be aborted.

Setting the transaction's state atomically commits—or discards in case of an abort—all object versions written by the transaction and removes the write markers on all written objects (as in DSTM [7]).

2.4 Use of Real-Time Clocks

The function getNewTS is actually required to return a timestamp that is larger than the time at which the function got invoked. For time bases that can tick on demand (e.g., counters), this condition is easily satisfiable. However, if a clock ticks independently and rarely (e.g., a slow real-time clock), the committing transaction T would have to wait for a new timestamp. If reading the time takes always longer than the time between two ticks of the time base (which is the case in our system), then this requirement is trivially satisfied.

The reason for this requirement is that threads need to agree on the validity ranges of object versions. Informally, we have to avoid a situation where one transaction draws conclusions about the state at time t and later another transaction modifies state at t. We ensure this by first putting an update transaction T into the *committing* state, which will get visible to other transactions at some time t_c when the $C\&S$ returns. getNewTS then sets $T.CT$ to a value larger than t_c (see above). Because transactions always read the time before they start to access objects (see Algorithms 2 and 3), it is guaranteed that if a transaction T_a accesses a version at time t, all transactions T that could commit a change to the object at $T.CT = t$ are already in the *committing* state. T_a sees this state indirectly in all possibly updated objects, and will either not access the version at time t or wait for all T to commit or abort.

3. TIME BASES

We will now show that we can use two kinds of real-time clocks for LSA-RT: perfectly synchronized clocks and externally synchronized clocks. Synchronizing real-time clocks in distributed systems is a well studied topic [1, 4]. With the appropriate hardware support, one could achieve perfectly synchronized clocks in the sense that there is no observable semantic difference between accessing some global real-time clock or processor-local replicas of the global real-time clock. There is of course a performance difference because there is no contention when processors access their local clock but there might be quite some contention when instead accessing a single global real-time clock. We show in Section 3.1 how one can implement the utility functions of LSA-RT with a perfectly synchronized clock.

In systems that do not have hardware-based clock synchronization, we can synchronize clocks in software. When doing so, we need to expect that there is an observable deviation between the individual real-time clocks. We address the issues of externally synchronized clocks in Section 3.2.

3.1 Perfectly Synchronized Real-Time Clocks

We assume a notion of real-time. Each thread p has access to a local clock C_p. Clocks are perfectly synchronized if C_p at real-time t ($C_p(t)$) is always equal to t. Reading a local clock that is perfectly synchronized always satisfies linearizability. Furthermore, we require that synchronization instructions (e.g., $C\&S$) are linearizable. Algorithm 4 shows the functions for a time base that uses perfectly synchronized clocks. getTime simply reads C_p and returns its value. getNewTS has to make sure that the returned time t is larger than the time at which the getNewTS was invoked. If time always advances when reading the local clock (e.g., because reading the clock takes some time) the busy-waiting loop is not necessary. Because perfectly synchronized clocks are linearizable, $\succcurlyeq$, `min`, and `max` have straightforward definitions.

3.2 Externally Synchronized Real-Time Clocks

Previously, we assumed that the global time base is a linearizable counter or a perfectly synchronized clock. However, scalable counters that provide low-latency accesses are hard to implement. In turn, perfectly synchronized clocks need to have a high resolution to avoid waiting when acquiring a new timestamp (lines 7–9 in Algorithm 4), which

Algorithm 4 Utility functions for perfectly synchronized clocks

```
function getTime()                              ▷ Get current timestamp
    $t \leftarrow readLocalClock()$
    return $t$
end function

function getNewTS()                             ▷ Get strictly greater timestamp
    $t_s \leftarrow readLocalClock()$
    repeat                                      ▷ Loop only required for slow clocks
        $t \leftarrow readLocalClock()$
    until $t > t_s$
    return $t$
end function

function $\succcurlyeq(t_1, t_2)$               ▷ Guaranteed later than or equal
    return $t_1 \geq t_2$
end function

function $\max(t_1, t_2)$                       ▷ Maximum
    if $t_1 \succ t_2$ then
        return $t_1$
    else
        return $t_2$
    end if
end function

function $\min(t_1, t_2)$                       ▷ Minimum
    if $t_1 \succ t_2$ then
        return $t_2$
    else
        return $t_1$
    end if
end function
```

makes perfect synchronization more expensive. Finally, the cost of accurate hardware clocks can be prohibitive.

Therefore, we want to be able to use clocks that return imprecise values but for which the deviation *dev* between real-time t and the value of the local clock C_p at time t is bounded. For a time-based transactional memory, the imprecision essentially means that it cannot be certain whether an object version was valid at a certain time or not.

We handle that uncertainty in a straightforward way. If a transaction is not sure that a version is valid at a certain time, it assumes that the version is not valid at this time. Thus, it masks uncertainty errors. Because the deviation is bounded, only the lower and upper bounds of a version's validity range are affected. Informally, the bounds of the validity range are virtually brought closer by *dev* each. This creates gaps of size $2 \cdot dev$ between versions, which can reduce the probability that LSA-RT finds an intersection between the validity ranges of object versions.

A transactional memory can always fall back to using validation. In current time-based transactional memories, updates to objects are always visible independently of timing information. Thus, time-based transactional memories are at least as efficient as transactional memories that only rely on validation.

For externally synchronized clocks, we require that the local clock EC_p for each thread p has a known maximum deviation *dev* from real time t (i.e., $|EC_p(t) - t| \leq dev$). Accordingly, a timestamp obtained at real-time t from EC_p consist of a local time $ts = EC_p(t)$, an ID *cid* for the local clock, and the maximum deviation *dev*.

The utility functions for externally synchronized clocks are shown in Algorithm 5. In $\succcurlyeq$, timestamps from the same clock are handled specially because no deviation has to be considered in this case. Otherwise, the deviation represents the uncertainty and is taken into account (line 14). Function `max` checks if one of the timestamps is guaranteed to be later than or equal to the other, in which case the former is returned. Otherwise timestamps do not originate from the same clock. Therefore, we select the timestamp with the largest upper bound (value plus deviation) and we set its clock ID to *undefined* to indicate that future comparisons will always need to take into account the deviation (line 14). Function `min` is defined similarly.

getTime and getNewTS are similar to those of perfectly synchronized clocks. However, the loop in getNewTS is not required because we assume that $dev > 0$. The way in which $\succcurlyeq$ masks uncertainty makes sure that versions are never valid at exactly the time at which they were committed, which prevents the misbehavior that would otherwise require waiting for a new time value.

Algorithm 5 Utility functions for externally synchronized clocks

```
 1: function getTime()                          ▷ Get current timestamp
 2:     $(ts, cid, dev) \leftarrow readExtSyncClock()$
                                                ▷ Time, clock ID, deviation
 3:     $t \leftarrow (ts, cid, dev)$
 4:     return $t$
 5: end function

 6: function getNewTS()                         ▷ Strictly greater timestamp
 7:     $t \leftarrow getTime()$                ▷ Loop is not necessary when $dev > 0$
 8:     return $t$
 9: end function

10: function $\succcurlyeq(t_1, t_2)$           ▷ Guaranteed later than or equal
11:     if $t_1.cid = t_2.cid \wedge t_1.cid \neq undefined$ then
12:         return $t_1.ts \geq t_2.ts$
13:     else
14:         return $t_1.ts - t_1.dev \geq t_2.ts + t_2.dev$
15:     end if
16: end function

17: function $\max(t_1, t_2)$                   ▷ Maximum
18:     if $t_1 \succcurlyeq t_2$ then
19:         return $t1$
20:     else if $t_2 \succcurlyeq t_1$ then
21:         return $t2$
22:     else if $t_1.ts + t_1.dev > t_2.ts + t_2.dev$ then
23:         return $(t_1, undefined, t_1.dev)$
24:     else
25:         return $(t_2, undefined, t_2.dev)$
26:     end if
27: end function

28: function $\min(t_1, t_2)$                   ▷ Minimum
29:     if $t_1 \succcurlyeq t_2$ then
30:         return $t2$
31:     else if $t_2 \succcurlyeq t_1$ then
32:         return $t1$
33:     else if $t_1.ts - t_1.dev < t_2.ts - t_2.dev$ then
34:         return $(t_1, undefined, t_1.dev)$
35:     else
36:         return $(t_2, undefined, t_2.dev)$
37:     end if
38: end function
```

4. CASE STUDY: MMTIMER VS. SHARED INTEGER COUNTER

To show how a time base can affect transactional memory performance, we investigate performance on a machine in which shared counters have a noticeable overhead. We executed our benchmarks on a 16-processor partition of an SGI Altix 3700, a ccNUMA machine with Itanium II processors. Pairs of two processors share 4GB of memory and are

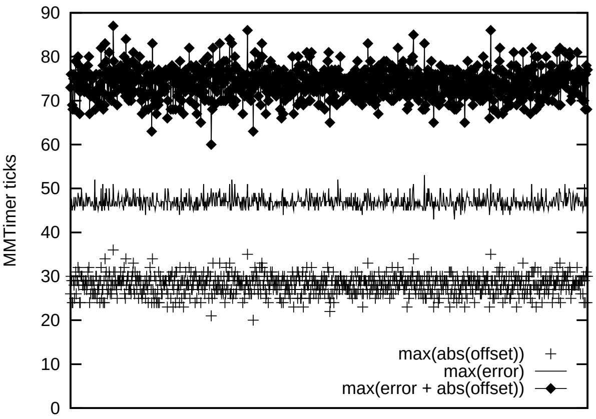

Figure 1: MMTimer synchronization errors and offsets.

connected to all other processor pairs. The STM we use is a C++ implementation of LSA-RT. We use two time bases: (1) an ordinary shared integer counter and (2) MMTimer, a hardware clock built into the Altix machines.

4.1 MMTimer

MMTimer is a real-time clock with an interface similar to the High Precision Event Timer widely available in x86 machines. It ticks at 20 MHz but reading from it takes always 7 to 8 ticks of the MMTimer, so the effective granularity is much coarser than one would expect from 20 MHz. In particular, the MMTimer is therefore strictly monotonic, i.e., both GetTime and GetNewTS just return the value of MMTimer.

At first, we had no information about whether MMTimer is a synchronized clock or not. We therefore used a simple test to measure the synchronization error by having threads on different CPUs read from the MMTimer and comparing the clock value obtained at each CPU with a reference value published by a thread on another CPU. Figure 1 shows the results of a four-hour run with synchronization rounds every tenth second.

In the figure, offsets represent the estimated difference of local clock values to the reference clock value and errors denote the largest possible deviation between the estimated offset and the offset that could be achieved by a perfect comparison. Only the maximum values of all CPUs are shown for each round. The results show that there is no drift, so the MMTimer behaves as a global clock or a set of synchronized clocks. Second, errors are always larger than offsets, so MMTimer could well be a perfectly synchronized clock. Third, the error seems to be bounded and is not too large: 90 ticks seems to be a reasonable estimate for its bound (see Figure 1). However, our clock comparison algorithm suffers from its communication over shared memory, so the MMTimer's actual synchronization error bounds could be much smaller.

We later came to know that MMTimer is indeed a synchronized clock [8]. Every node in the Altix system has one register for the clock that is accessible via the MMTimer interface. Before system boot, a single node is selected as source for the clock signal, and all other nodes' clocks are synchronized to this node. During runtime, the source clock then advances all other nodes' clocks. Dedicated cables are used for the clock signal. However, we do not know how the synchronization mechanism works in detail (e.g., synchronization errors could arise from a varying latency of the clock signal). We have reasons to assume that such potential errors are already masked by the time that it takes to read the MMTimer (7 or 8 ticks of MMTimer), which would mean that MMTimer behaves like a linearizable perfectly synchronized clock.

The important observation is that, unsurprisingly, hardware support can ensure a much better clock synchronization than mechanisms that require communication via shared memory (in our case, 8 ticks vs. 90 ticks). We would like to see more multiprocessor systems providing synchronized clocks. Furthermore, synchronization errors should be guaranteed to be bounded and the bounds should be published to enable concurrent applications to use synchronized clocks to their full potential.

4.2 Time Base Overheads

To investigate the overheads of using shared counters and MMTimer as time bases, we used a simple workload in which transactions update distinct objects (but this fact is not known a priori). This type of workload exists in many larger systems: the programmer relies on the transactional memory to actually enforce atomicity and isolation of concurrent computations. Furthermore, performance in this workload is not affected by other properties of the transactional memory (e.g., contention management), which makes the overhead of the time base more apparent. Figure 2 shows throughput results for this workload for update transactions of different sizes. For very short transactions, MMTimer's overhead decreases throughput in the single-threaded case. However, the overhead gets negligible when transactions are larger. More importantly, using a shared counter as time base prevents the STM from scaling well, whereas with MMTimer, performance increases linearly with the number of threads. The influence of the shared counter decreases when transactions get larger because the contention on the counter decreases. However, the influence of the shared counter's overhead would increase again if the STM would perform its operations faster or more CPUs would be involved. An optimization for the counter similar to the one used by TL2 (see Section 1.2) showed no advantages on our hardware.

4.3 Synchronization Errors

Synchronization errors shrink the object versions' validity ranges. If the STM is a multi-version STM and accesses old versions, validity ranges will be decreased at the beginning and at the end. Thus if the length of a transaction's validity range can be smaller than twice the error (see Algorithm 5), then the transaction will abort more often.

For single-version STMs, the synchronization error only affects the beginning of object versions (i.e., the commit timestamps are virtually increased by the size of the error). Whether that matters depends again on the workloads and STM-specific costs.

Based on these observations, it is difficult to draw generic conclusions about the influence of synchronization errors. Benchmarks are typically more influenced by the performance of the transactional memory's implementation (e.g., object access costs) and very much by the properties of the workload (e.g., locality, update frequencies, or the duration

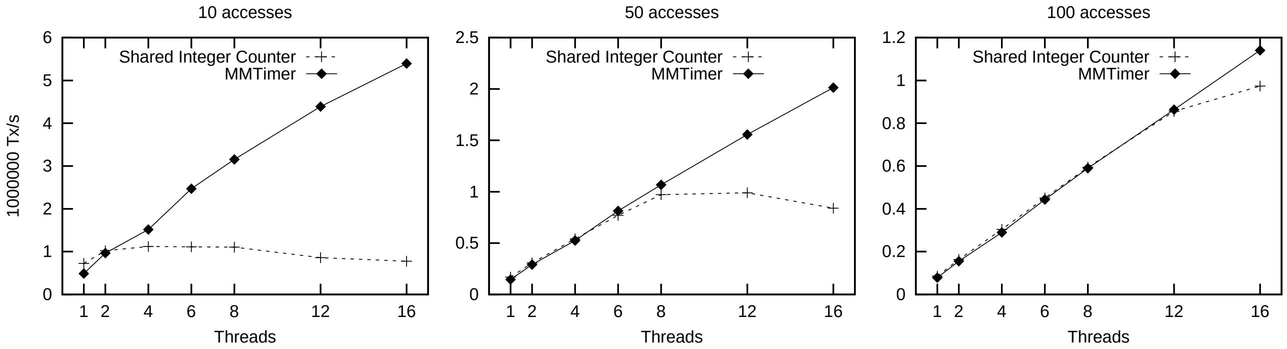

Figure 2: Overhead of time bases for update transactions of different size.

of transaction). For example, if the synchronization error is smaller than the complete overhead imposed by a cache miss plus the commit of an update transaction, then the error has no influence at all in single-version transactional memories. This is because the vulnerability period associated with the possible synchronization error has elapsed at the time when a reader could actually work with an object version. Note as well that the synchronization error does not need to be considered if a transaction reads local updates (see Algorithm 5).

5. CONCLUSION

Time-based transactional memories use the notion of time to reason about the consistency of data accessed by transactions without requiring frequent and expensive validity checks. It is our strong belief that time is an essential concept for implementing scalable transactional memories.

In this paper, we have focused on transactional memories that use real-time as a time base. Real-time clocks have significant benefits over simple logical counters. In particular, one can more easily parallelize real-time clocks, e.g., using internal or external clock synchronization algorithms. They provide increased scalability because they avoid the contention on a single shared counter, which we have shown to be a major bottleneck for short transactions or when executing many threads.

Perfectly synchronized clocks with a high frequency would be the ideal basis for an time-based transactional memory. Such clocks could be implemented with relative ease in hardware. If not available, one can also implement clock synchronization in software with lower accuracy. Tight external (or internal) clock synchronization is achievable and the tighter the clocks are synchronized, the better will the transactional memory perform.

We have introduced a new lazy snapshot construction algorithm that can use different time bases. We have specifically shown how it can be used with perfectly synchronized clocks and externally synchronized clocks. However, it can also be used in a straightforward way with internally synchronized clocks or logical commit time counters. Using our algorithm, one can balance the scalability of the time base and the tightness of its synchronization. The trade-off will be different for different systems: for small systems a simple shared commit time counter will be sufficient whereas, for very large systems, a hardware based external clock synchronization might be the best choice.

Acknowledgments

We thank Andreas Knüpfer for pointing us to the MMTimer. Robin Holt and Reiner Vogelsang from SGI provided valuable information about MMTimer's synchronization.

6. REFERENCES

[1] F. Cristian. A probabilistic approach to distributed clock synchronization. *Distributed Computing*, 3:146–158, 1989.

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

[3] D. Dice and N. Shavit. What really makes transactions fast? In *TRANSACT*, Jun 2006.

[4] C. Fetzer and F. Cristian. Integrating external and internal clock synchronization. *Journal of Real-Time Systems*, 12(2):123–171, March 1997.

[5] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA*, pages 388–402, Oct 2003.

[6] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *PLDI '06: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC*, Jul 2003.

[8] Robin Holt, SGI. Personal Communication.

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

[10] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *TRANSACT06*, Jun 2006.

[11] 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 *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[12] 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 International*

Symposium on Distributed Computing (DISC), September 2006.

[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 *International Symposium on Code Generation and Optimization (CGO)*, 2007.