Commit Phase in Timestamp-based STM

Rui Zhang
Dept. of Computer Science
Rice University
Houston, TX 77005, USA
ruizhang@rice.edu

Zoran Budimlić
Dept. of Computer Science
Rice University
Houston, TX 77005, USA
zoran@rice.edu

William N. Scherer III
Dept. of Computer Science
Rice University
Houston, TX 77005, USA
scherer@rice.edu

ABSTRACT

Timestamp-based Software Transactional Memory (STM) validation techniques use a global shared counter and timestamping of objects being written to reason about sequencing of transactions and their linearization points, while reducing the number of unnecessary validations that have to be performed, thus improving overall system performance.

During the commit phase of a timestamp-based validation scheme, several actions have to be performed: locking of the objects being written to the memory, atomically incrementing a shared timestamp counter, updating timestamps for objects being committed to memory, performing a final validation of the transaction's consistency, and atomically effecting the the transaction's changes to the outside world. The order and manner in which these actions are performed can affect both the correctness of the STM implementation and the overall system performance.

We identify several commit sequence designs, prove their correctness, and analyze their performance. We identify cases where timestamps do not have to be unique for different transactions committing concurrently, and cases where unnecessary updates of the global shared counter — which can trigger extra validations in other transactions, hurting performance — can be avoided. We evaluate these commit sequence designs on a set of benchmarks on a 16 processor SunFire SMP machine. We show that a carefully chosen commit sequence can improve overall system performance by up to 14% over the state of the art single counter timestamp-based validation techniques, and we show that it is possible to obtain high performance without incurring space overhead proportional to the number of objects in the system.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'08, June 14–16, 2008, Munich, Germany.
Copyright 2008 ACM 978-1-59593-973-9/08/06 ...$5.00.

Keywords

Transactional Memory, Timestamp, Commit Sequence

1. INTRODUCTION

Transactional Memory (TM) [7, 12] has emerged as a promising high-level programming model for the new and omnipresent multicore machines. TM allows the programmer to think in terms of *transactions*, operations that are guaranteed to perform *atomically* without introducing expensive and deadlock-prone coarse-grain locking. TMs are particularly attractive for complex and dynamic data structures, where different processes can operate in parallel on the data structure, and where coarse-grain locking would introduce serialization on the data structure (and unacceptable performance penalties), while fine-grain locking is difficult to implement, error-prone and deadlock-prone.

Software Transactional Memory (STM) [2, 3, 4, 5, 6, 8, 11] systems implement the TM functionality exclusively in software. These systems have to ensure that the state of the shared objects in memory is consistent at all times, for all transactions. A widely accepted method in current STM implementations is *validation*: The transaction rescans the shared objects it has accessed to confirm that they remain consistent. Validation can be an expensive operation that, performed unnecessarily, can degrade the system's performance significantly. One study [11] has shown that for some applications, STM can spend up to 50% of the time in validation, leaving only 20% for useful work (30% of the time is spent on other overhead).

Timestamp-based validation techniques [1, 9, 13] attempt to reduce the validation overhead by introducing a global shared counter and/or a small field (a *timestamp*) into each object. These timestamps allow the STM system to quickly determine when a full validation of the transaction is unnecessary when performing critical operations such as opening new objects. Timestamp-based validation techniques such as Transactional Locking II [1], Lazy Snapshot [9], and Global Commit Counter [13] significantly reduce unnecessary validations at the cost of a slight increase in memory requirements and of potential contention on a shared timestamp counter.

Timestamp-based Software Transactional Memory systems associate a timestamp with each shared object to indicate when an object was last modified. A transaction also acquires a timestamp indicating its current candidate linearization point: the time when the transaction's effects become visible to the outside world. This time information allows comparisons of the order of execution among in-flight transactions, providing the STM system an opportunity to skip validation when opening an object that was last modified before the transaction's linearization point.

The design of a timestamp-based STM system faces many trade-offs. For example, the value of the global counter chosen to be the linearization point of the transaction, the value of the global counter chosen to be the timestamp of the objects written by the transaction, the implementation of the shared global counter and the timestamps, and the requirements that timestamps, linearization points and global counter have to satisfy can all have a significant impact on correctness and performance of the STM system.

Current time-based STM systems exhibit some issues:

- In Transactional Locking II, Lazy Snapshot, and Global Commit Counter, every update transaction requires a unique timestamp (hence a total ordering) for the objects being updated. We show that this requirement is unnecessary, and present commit sequences that relax this requirement and either scale better or consume less memory space.

- Existing timestamp-based techniques update the shared global counter more times than absolutely necessary. These extraneous updates can in turn force unnecessary validations and increase the contention on the counter itself. We demonstrate new commit strategies that reduce the number of unnecessary counter updates and that improve overall system performance by up to 9%.

The remainder of this paper is organized as follows. In Section 2 we describe the current state of the art in timestamp-based validation techniques. In Section 3 we present several variants of the commit sequence in a timestamp-based validation scheme for single-version STMs. In Section 4 we present experimental results that illustrate the impact of commit sequences on overall system performance. In Section 5 we conclude the paper and suggest directions for future research.

2. RELATED WORK

Validation is a technique designed to prevent a transaction from observing an inconsistent state in shared data. It is closely related to the conflict detection scheme used in the STM systems. An exhaustive conflict detection system that detects all possible w/w, r/w, and w/r conflicts would not need validation, but it would also be expensive.

To increase concurrency, different conflict detection strategies have been developed. *Invisible reads* [6] hide the read of an object from concurrent transactions which are thereafter able to modify the object. This makes write after read no longer a conflict. *Lazy writes* [2] hide the write from other transactions and allows modifications to them by other transactions, making read after write and write after write no longer a conflict. To avoid entering an inconsistent state, a transaction needs to validate the objects it has read (including the writes that are lazily acquired) at appropriate times in program execution.

Incremental validation [6] is a strategy that validates previously-accessed objects each time the transaction opens a new object. If any change in the past is detected, the validation fails. This strategy guarantees a consistent state but imposes a substantial overhead [13], quadratic in the number of objects opened in a transaction.

In this section, we discuss three existing state of the art timestamp-based STM systems: Transactional Locking II (TL II) [1], Lazy Snapshot (LSS) [9] and Global Commit Counter (GCC) [13]. We will focus on the design of the shared global counter, method to update this counter, the format of the timestamp, and the impact of the different design choices these techniques make on performance.

Riegel, Fetzer, and Felber [10] use perfectly synchronized real-time clocks or externally synchronized physical clocks as the time base for better scalability. In contrast to our work, their scheme requires external support and focuses on the scalability of the time base itself. Our work focuses on reducing the number of unnecessary updates to the time base to improve the scalability of the system and requires no external hardware support.

For convenience, we use some short variable names for algorithms in this paper. We explain the meaning of these variables here. T is a transaction. O_i is an object. m is the mode to open an object. m can be read-only or read-write. $T.update$ is the flag to indicate if the transaction is a read-only transaction or update transaction. $T.ts$ is the transaction's timestamp. It is used to reason about the ordering of the transaction and the objects. TS is the variable to save the timestamp to be written into the objects being updated. TSC is the timestamp counter. It is a shared integer. $T.O$ is a list keeping the objects that need to be verified when doing a validation. $VALIDATE(T)$ checks if the invisible reads and lazy writes have been modified or acquired by other transactions. If they are, transaction T is aborted, otherwise it proceeds forward.

2.1 Transactional Locking II (TL II)

TL II can save the overhead of bookkeeping invisible reads for read-only transaction. The reason is it fixes its linearization point at the beginning of read-only transactions and does not validate.

TL II always performs a validation in the commit phase for update transactions. It updates the shared counter and acquires a unique timestamp using either a looped CAS for a single counter or a non-looped CAS for a tuple (consisting of a timestamp and the thread ID). It performs unnecessary updates to the shared counter when the validation fails.

TL II also suggests one timestamp implementation to reduce the contention on the global shared counter by pairing the original timestamp with a thread ID for each shared object. Unfortunately, the paper does not give any experimental results on this suggested timestamp implementation. The tuple of the thread ID and the timestamp is unique for each update transaction and satisfies the uniqueness requirement, allowing the commit phase to avoid the use of CAS in a loop to force the shared counter update.

Having the commit sequence always perform validation is a sufficient condition to relax the requirement for timestamp uniqueness. It also saves the per-object space overhead for storing the last updater's thread ID. We discuss this variant and evaluate its performance in Sections 3 and 4.

To more directly compare different commit sequences, we changed TL II's $OPEN$ function to make it compatible with other versions. This change consists of updating the transaction's timestamp and revalidating, instead of aborting, when an `OPEN` detects that an object has been modified at a time later than the transaction's candidate linearization point.

Function `TL II OPEN(`T, O_i, m`)` single counter version

```
if $m = write$ then
    $T.update \leftarrow true$;
if $O_i.ts > T.ts$ then
    $T.ts \leftarrow TSC$;
    $VALIDATE(T)$;
$T.O \leftarrow T.O \cup O_i$;
```

Function `TL II COMMIT(`T`)` single counter version

```
if $T.update$ then
    $ACQUIRE(T)$;
    $TS \leftarrow TSC$;
    while $TS$ != $CAS(\&TSC, TS, TS+1)$ do
        $TS \leftarrow TSC$;
    $TS \leftarrow TS + 1$;
    $VALIDATE(T)$;
    foreach $O_i$ in $T.O$ do
        $O_i.ts \leftarrow TS$;
$CAS(\&TX\_STATUS, ACTIVE, COMMITTED)$;
```

Function `TL II OPEN(`T, O_i, m`)` tuple version

```
if $m = write$ then
    $T.update \leftarrow true$;
if $O_i.ts > T.ts$ or ($O_i.ts = T.ts$ and $O_i.id$ != $ID$) then
    $T.ts \leftarrow TSC$;
    $VALIDATE(T)$;
$T.O \leftarrow T.O \cup O_i$;
```

2.2 Lazy Snapshot (LSS)

LSS skips validation in the commit phase for read-only transactions. It generates a unique timestamp using an atomic fetch-and-increment instruction for each update transaction, thereby serializing all update transactions when incrementing the shared counter. In our experiments we use a looped CAS to simulate the atomic fetch-and-increment instruction. LSS also skips validation when the timestamp counter is unchanged since its last validation: If no competing transaction has committed, then no new conflicts can have been generated. However, if a final validation should fail, LSS will have unnecessarily updated the shared counter.

2.3 Global Commit Counter (GCC)

GCC is not a typical timestamp-based STM because it does not keep timestamps in shared objects. It uses a single counter as the time base. It skips the validation in the commit phase when there is no change to the counter since last time it was visited. It uses a looped CAS to update the shared counter (and ensure that the counter *does* get updated by the current transaction, in addition to any other updates that may happen to it at the same time by other transactions). It uses a unique timestamp for each update transaction. It performs a full validation when read-only transaction opens an object and detects a change to the global counter. As detailed in Section 3.4, this design admits a subtle race condition in which conflicting transactions may incorrectly commit together.

Overall, the various timestamp designs use two different methods to update the shared counter.

Forced update is used when the shared counter *has* to be updated. This is often implemented using a looped cas.

Function `TL II COMMIT(`T`)` tuple version

```
if $T.update$ then
    $ACQUIRE(T)$;
    if $T.ts = TSC$ then
        $TS_{new} \leftarrow CAS(\&TSC, T.ts, T.ts + 1)$;
        if $TS_{new}$ != $T.ts$ then
            $TS \leftarrow TS_{new}$;
    else
        $TS \leftarrow TSC$;
    $VALIDATE(T)$;
    foreach $O_i$ in $T.O$ do
        $O_i.ts \leftarrow TS$;
        $O_i.id \leftarrow ID$;
$CAS(\&TX\_STATUS, ACTIVE, COMMITTED)$;
```

Function `Lazy Snapshot OPEN(`T, O_i, m`)`

```
if $m = write$ then
    $T.update \leftarrow true$;
if $O_i.ts > T.ts$ then
    $T.ts \leftarrow TSC$;
    $VALIDATE(T)$;
$T.O \leftarrow T.O \cup O_i$;
```

Function `Lazy Snapshot COMMIT(`T`)`

```
if $T.update$ then
    $ACQUIRE(T)$;
    $TS \leftarrow TSC$;
    while $TS$ != $CAS(\&TSC, TS, TS+1)$ do
        $TS \leftarrow TSC$;
    if $TS$ != $T.ts$ then
        $VALIDATE(T)$;
    $TS \leftarrow TS + 1$;
    foreach $O_i$ in $T.O$ do
        $O_i.ts \leftarrow TS$;
$CAS(\&TX\_STATUS, ACTIVE, COMMITTED)$;
```

Function `GCC OPEN(`T, O_i, m`)`

```
if $m = write$ then
    $T.update \leftarrow true$;
if $T.ts$ != $TSC$ then
    $T.ts \leftarrow TSC$;
    $VALIDATE(T)$;
$T.O \leftarrow T.O \cup O_i$;
```

Function GCC COMMIT(T)

```
$ACQUIRE(T)$;
if $!TRY\_COMMIT(T)$ then
    $VALIDATE(T)$;
    $TS \leftarrow TSC$;
    while $TS = CAS(\&TSC, TS, TS + 1)$ do
        $TS \leftarrow TSC$;
$CAS(\&TX\_STATUS, ACTIVE, COMMITTED)$;
```

Function GCC TRY_COMMIT(T)

```
if $T.update$ then
    $TS_{new} \leftarrow CAS(\&TSC, T.ts, T.ts + 1)$;
    if $TS_{new} = T.ts$ then
        $result \leftarrow TRUE$;
    else
        $result \leftarrow FALSE$;
else
    if $TSC = T.ts$ then
        $result \leftarrow TRUE$;
    else
        $result \leftarrow FALSE$;
return $result$
```

If used in a design where each update transaction uses a unique timestamp, all update transactions are serialized at updating the shared counter. This can become a bottleneck in highly parallel system where many transactions can reach their commit phase around the same time.

Non-forced updates can be performed when only an attempt to update the shared counter is sufficient to guarantee consistency. If the attempt succeeds (the CAS to update the shared counter succeeds), the comit sequence can continue as planned. If the attempt fails, a validation is needed to ensure consistency. The contention for the shared counter is greatly reduced in this case. TL II's suggested tuple implementation uses this scheme to update the counter part of the timestamp.

Another issue associated with existing designs is of unnecessary updates to the shared counter, which can force other transactions to perform an avoidable final validation at commit time.

3. COMMIT SEQUENCE DESIGN

In this section we present several alternatives to the the commit sequences in the existing time-based STM systems. In particular, we relax the requirement for the *uniqueness* of timestamps and address the *false update* problem that can force transactions to perform unneeded validations (and by slowing their completion, increase the window for potential conflicts with other transactions). False updates occur when a transaction leads to a shared counter update that does not correspond to the completion of any transaction.

3.1 Non-unique Timestamps

In existing timestamp-based STM systems, every write transaction needs to acquire a unique timestamp for the objects it updates. This uniqueness is achieved via a single integer timestamp or a tuple that has non-unique timestamp and a thread ID. Using a single unique integer requires an atomic increment to a shared timestamp counter for each update transaction, which serializes update transactions and reduces the scalability of the STM system. Using a tuple with thread ID reduces contention because the thread ID is not shared; this avoids serializing all update transactions on the shared counter, but incurs additional space overhead.

Timestamp uniqueness is a sufficient condition for enforcing an order among transactions. However, as we demonstrate in this paper, it is a more powerful property than strictly necessary, and leads to space and/or contention overheads. We present a method for relaxing the uniqueness requirement for timestamps and show that it can yield reduced contention on a shared counter without incurring the space overhead.

Our method is based on the observation that disjoint transactions can share a common timestamp if they commit concurrently. Within the commit sequence, an ACQUIRE operation ensures that any write/write inter-transactional conflicts are detected. Read/write conflicts, meanwhile, can be subsequently detected by a VALIDATE operation; so if two transactions have both successfully completed ACQUIRE, they may safely share their timestamp *provided* they both validate their read set. Any moment in time between the ACQUIRE and the final commit can be used as the effective timestamp for a transaction. Although a transaction can still be aborted by a competitor after its ACQUIRE, this case is handled by the aborted transaction's status word update: when updating from ACTIVE to COMMITTED fails, the new versions of objects, with updated timestamps, are never effected as current. We argue more carefully the correctness of these claims later in this section.

Many authors have noted that, where possible, skipping validation during commit can improve transactional system throughput; this is done when the shared timestamp counter has not changed. When the counter *has* changed, we can choose either to abort and restart the transaction, or to force another update of the counter and validate the transaction.

3.2 False Update Avoidance

We consider two scenarios for false updates in timestamp-based STMs that occur after the shared counter is updated as part of the commit sequence. The first scenario occurs when the transaction's own subsequent validation fails and it aborts. The second occurs when another transaction aborts (i.e. it detects a conflict) the one that has updated the timestamp. Both scenarios can potentially lead to redundant validation in otherwise uninvolved transactions when the STM system tries to skip a final commit-time validation. Since false update can potentially trigger a validation in each concurrently executing transaction, in a worst-case scenario the total amount of extra validation work performed can grow linearly with the number of transactional threads in a system — a clear scalability problem.

3.3 Commit Sequence Design Alternatives

The design space for commit sequences may be organized around several individual axis; we introduce nomenclature for them here.

We term *hard validation* the case where a VALIDATE operation is performed unconditionally in the commit sequence. In contrast, it may be skipped under certain circumstances in *soft validation*. If we always perform a validation, the

```
Function V1, V2, V4 OPEN($T$, $O_i$, $m$)
if $m = write$ then
    $T.update \leftarrow true$;
if $O_i.ts > T.ts$ then
    $T.ts \leftarrow TSC$;
    $VALIDATE(T)$;
$T.O \leftarrow T.O \cup O_i$;
```

update to the time base may be placed virtually anywhere in the commit sequence, and the expected value for the CAS may be either the candidate linearization point or a fresh read of the global counter. In order to skip commit-time validation (in cases where conditions are met for doing so), an update to the timestamp must occur before the validation attempt.

With a *hard increment*, the timestamp counter is always incremented during the commit sequence by the thread executing the transaction (typically via a looped CAS). With a *soft increment*, the timestamps can be shared, so the counter update can sometimes be skipped. In some of our proposed variations to the commit sequence, it suffices for a committing transaction to observe that an update has been made to the shared counter, whether the transaction's executing thread was the one that successfully performed the increment or not. This avoids full serialization of all transactions and reduces contention on the timestamp counter, but it also leads to non-unique timestamps.

We characterize as *eager (or lazy) timestamp acquire* the case where the base value used to update the timestamp counter is read before (or after) the ACQUIRE operation. Finally, having *side effects* means that the OPEN operation needs to perform validations not only when the object's timestamp is greater than transaction's timestamp, but also in some other cases. We summarize these differences in table 1.

Version 1: This design uses a single timestamp counter. It does not necessarily assign a unique timestamp for each transaction; instead it can share a timestamp across concurrently committing transactions. This allows it to perform only one CAS operation on the shared timestamp counter; even if the CAS fails, the counter is guaranteed to have been updated. Scalability is improved as contention on the counter is reduced, and the strict serialization of transactions on the counter is relaxed, as in LSS, GCC, and TL2C. Unlike TL2T, this version avoids using additional space for a thread ID. One disadvantage of this design is that validation must always be performed during the commit phase.

Version 2 (V2): This design also uses a shared timestamp counter. It attempts to avoid false updates in the timestamp counter by validating past reads each time a CAS fails, getting a fresh read of the counter for each attempted update. The main disadvantage of this design is that every transaction must force an update to the timestamp counter, which increases contention.

The OPEN functions in V1 and V2 are identical.

Version 3 (V3): This design is similar in most respects to V1. The key difference from V1 is in updating the shared timestamp counter: Where V1's CAS uses as its expected value a fresh read obtained after performing the ACQUIRE operation, V3's CAS uses the most recently read value. In our correctness proof for this design, we show that the ACQUIRE operation is a critical point; this difference forces V3 to perform additional validations on open when the object and transaction timestamps match. This version is suitable for systems with abundance of parallelism and where the transactions have a very good chance to succesfully commit. The *CAS* in V3 has a high probability failing and the commit can take the quick path in the if statement. HashTable is one benchmark where V3's performance stands out. V3's performance matches TL II tuple version, without the additional space requirements. Due to the additional validations when opening a object with an equal timestamp, V3's performance suffers when the validation overhead is high, as demonstrated in LinkedList and RandomGraph.

```
Function V1 COMMIT($T$)
if $T.update$ then
    $ACQUIRE(T)$;
    $VALIDATE(T)$;
    $TS \leftarrow TSC$;
    if $TS = TSC$ then
        $TS_{new} \leftarrow CAS(\&TSC, TS, TS + 1)$;
        if $TS_{new} = TS$ then
            $TS \leftarrow TS + 1$;
        else
            $TS \leftarrow TS_{new}$;
    else
        $TS \leftarrow TSC$;
    foreach $O_i$ in $T.O$ do
        $O_i.ts \leftarrow TS$;
$CAS(\&TX\_STATUS, ACTIVE, COMMITTED)$;
```

```
Function V2 COMMIT($T$)
if $T.update$ then
    $ACQUIRE(T)$;
    $TS \leftarrow T.ts + 1$;
    while $T.ts$ != $TSC$ or $T.ts$ = $CAS(\&TSC,$ $T.ts,$ $TS)$ do
        $T.ts \leftarrow TSC$;
        $VALIDATE(T)$;
        $TS \leftarrow T.ts + 1$;
    foreach $O_i$ in $T.O$ do
        $O_i.ts \leftarrow TS$;
$CAS(\&TX\_STATUS, ACTIVE, COMMITTED)$;
```

```
Function V3 OPEN($T$, $O_i$, $m$)
if $m = write$ then
    $T.update \leftarrow true$;
if $O_i.ts >= T.ts$ then
    $T.ts \leftarrow TSC$;
    $VALIDATE(T)$;
$T.O \leftarrow T.O \cup O_i$;
```

Version 4 (V4): Our fourth design combines the ability to share timestamps (and the corresponding reduction in shared counter contention) from V1 with the ability to skip

	validation	increment	TS acquire	thread id	false update	side effect
V1	hard	soft	lazy	no	commit CAS fails	no
V2	soft	hard	eager	no	commit CAS fails	no
V3	hard	soft	eager	no	validation or commit CAS fails	yes
V4	soft	soft	eager	no	commit CAS fails	no
TLIIT	hard	soft	lazy	yes	validation or commit CAS fails	yes
TLIIC	hard	hard	lazy	no	validation or commit CAS fails	no
LSS	soft	hard	eager	no	validation or commit CAS fails	no

Table 1: Commit Sequence Comparison

Function `V3 COMMIT(`T`)`

```
if $T.update$ then
    $ACQUIRE(T)$;
    if $TSC = T.ts$ then
        $TS \leftarrow T.ts + 1$;
        $TS_{new} \leftarrow CAS(\&TSC, T.ts, TS)$;
        if $TS_{new}$ != $T.ts$ then
            $TS \leftarrow TS_{new}$;
    else
        $TS \leftarrow TSC$;
    $VALIDATE(T)$;
    foreach $O_i$ in $T.O$ do
        $O_i.ts \leftarrow TS$;
$CAS(\&TX\_STATUS, ACTIVE, COMMITTED)$;
```

Function `V4 COMMIT(`T`)`

```
if $T.update$ then
    $ACQUIRE(T)$;
    if $T.ts$ != $TSC$ then
        $T.ts \leftarrow TSC$;
        $VALIDATE(T)$;
    $TS \leftarrow CAS(\&TSC, T.ts, T.ts + 1)$;
    if $TS = T.ts$ then
        $TS \leftarrow TS + 1$;
    else
        $VALIDATE(T)$;
    foreach $O_i$ in $T.O$ do
        $O_i.ts \leftarrow TS$;
$CAS(\&TX\_STATUS, ACTIVE, COMMITTED)$;
```

validation in the commit sequence from V2. As such, it has potentially the lowest overhead of any of our commit sequences. Like V2, V4 shares a common `OPEN` function with V1.

3.4 Theoretical Considerations

The following lemma presents a key insight into sequencing of `ACQUIRE` and `VALIDATE` operations for timestamp-based validation techniques.

Lemma 1. *Suppose that at a moment in time t_3, a transaction T_1 opens an object O for reading that was written by a transaction T_2. Let t_1 be the moment in time that transaction T_2 successfully performed its* `ACQUIRE` *operation during its commit phase. If there was a moment in time t_2 at which transaction T_1 performs a successful validation, and if $t_3 > t_2 > t_1$, then T_1 does not need to validate its state at the moment t_3 when it opens the object O.*

Proof. If there are more than one successful validations performed by T_1 between moments t_1 and t_3, we will assume without a loss of generality that t_2 is the moment when that happens for the first time. Let t_4 be the moment in time when T_1 opens the object O for the first time. There are three possible scenarios:

- $t_4 < t_1$: the $ACQUIRE$ action performed by T_2 will fail, since it will attempt to obtain ownership of O, which is already open by T_1 for reading. The conditions that the lemma requires are not met.

- $t_1 < t_4 < t_2$: if T_2 hasn't committed yet, then the $OPEN$ action by T_1 in the moment t_4 will fail, since it will attempt to open an object that has been locked by T_2. If T_2 has committed at a moment t_5, then $t_5 < t_4 < t_2$ and validation at t_2 by T_1 will include O and ensure that all subsequent reads of O (including the one at t_3) is consistent with the state of the transaction.

- $t_3 \leq t_4 < t_2$: let t_5 be the time that T_2 commits. If $t_5 > t_4$ then the $OPEN$ at t_4 will fail. If $t_4 > t_5 > t_2$ then the success of the validation at t_2 guarantees that T_1 and T_2 do not share any objects at t_2, allowing T_1 to serialize after T_2 and use the object O written by T_2. If $t_5 < t_2$ then the validation at t_2 ensures that there are no conflicts between T_1 and T_2 and that T_1 can serialize after T_2, which allows T_1 to open O for reading without validation.

□

Informally, Lemma 1 states that for any two transactions T_1 and T_2, if T_1 performs a validation after T_2 performs the `ACQUIRE` command, then after the validation, T_1 can open any objects written by T_2 without validation.

At this point, we need to emphasize an important property of the timestamp-based validation techniques: every time a transaction updates its linearization point $T.ts$ to the current value of the shared global counter TSC, it also performs a full validation. This happens when the transaction tries to open an object with a timestamp $O.ts > T.ts$, and also at the beginning of the transaction. For simplicity in following explanation, we can assume that a call to $VALIDATE(T)$ is made at the beginning of each transaction as well, since this call would not do anything because there are no objects open by the transaction at that time. Therefore, every update to $T.ts$ is accompanied by a call to $VALIDATE(T)$. Also, without loss of generality, we will assume that the update to $T.ts$ and the full validation happen at the same single point in time t_{Tts}, at the end of the call to $VALIDATE(T)$.

Theorem 1. *Algorithm $V1$ satisfies the consistency requirement of transactional memory.*

Proof. Since $V1$ always performs a validation during the commit phase, we only need to prove that $V1$ does not lead to inconsistency when opening an object.

Without loss of generality, suppose O is the object written by transaction T_1 and read by transaction T_2. We prove that for all possible timestamps written into O, there will not be an inconsistency in T_2. Let t_{AQ} be the time that V_1 performs its $ACQUIRE$ action on line 2.

Since any scenario where $O.ts > T_2.ts$ forces a validation on $OPEN(O)$, we only need to consider cases where $O.ts \leq T_2.ts$. Let t_{T2ts} be the point in time when the $T_2.ts$ gets updated to its current value. According to Lemma 1, to prove consistency, it is sufficient to prove that $t_{T2ts} > t_{AQ}$.

Let $t(line\ X)$ be the point in time when line X in function $V1\ COMMIT(T)$ gets executed. There are three possible places where the timestamp that gets written into O can be generated: line 8, line 10 and line 12 in function $V1\ COMMIT(T)$.

- $O.ts$ is generated at line 8: CAS at line 6 was successful. To obtain a value from TSC that is greater or equal to TS computed at line 8, it has to be $t_{T2ts} \geq t(line\ 6) \Rightarrow t_{T2ts} > t_{AQ}$.
- $O.ts$ is generated at line 10: CAS at line 6 failed $\Rightarrow$ someone else has updated TSC between $t(line\ 4)$ and $t(line\ 6)$. To obtain a value from TSC that is greater or equal to $TSNew$, it has to be $t_{T2ts} \geq t(line\ 4) \Rightarrow t_{T2ts} > t_{AQ}$.
- $O.ts$ is generated at line 12: the comparison at line 5 failed $\Rightarrow$ someone else has updated TSC between $t(line\ 4)$ and $t(line\ 5)$. To obtain a value from TSC that is greater or equal to TSC from line 12, it has to be $t_{T2ts} \geq t(line\ 4) \Rightarrow t_{T2ts} > t_{AQ}$.

□

Theorem 2. *Algorithm $V2$ satisfies the consistency requirement of transactional memory.*

Proof. The proof that opening an object when $O.ts \leq T.ts$ without a validation is identical to the proof of Theorem 1, with an exception that the possible places to generate the timestamp written into O are line 3 and line 7 in function $V2\ COMMIT(T)$.

- $O.ts$ is generated at line 3: the CAS at line 4 succeeds $\Rightarrow$ the only way for $T_2.ts$ to obtain a value of TSC that is greater or equal to TS at line 3 is if $t_{T2ts} \geq t(line\ 4)$ $\Rightarrow t_{T2ts} > t_{AQ}$.
- $O.ts$ is generated at line 7: the only way to exit the loop at line 4 is if both $T.ts = TSC$ comparison and CAS instructions succeed. Therefore, TSC is incremented every time an update transaction commits $\Rightarrow$ every update transaction has its own unique timestamp written to the objects it updates. Since $O.ts \leq T_2.ts$: at the moment t_{T2ts}, T_1 has updated TSC which means it has finished its while loop $\Rightarrow t_{T2ts} \geq t(line\ 8) \Rightarrow t_{T2ts} > t_{AQ}$.

Since algorithm $V2$ does not always perform a validation during the commit phase, we also need to prove that it does not violate consistency when test on line 4 fails immediately and the body of the while loop never gets executed.

They only way for the *while* loop at line 4 to exit immediately is if both conditions on line 4 fail $\Rightarrow T_1.ts = TSC \Rightarrow$ TSC has not changed since $t_{T1ts} \Rightarrow$ no objects were updated in the whole system since t_{T1ts}. Since T_1 has already performed a validation at t_{T1ts}, no validation is necessary at time $t(line\ 4)$. □

Theorem 3. *Algorithm $V3$ satisfies the consistency requirement of transactional memory.*

Proof. Since algorithm $V3$ always performs a validation during the commit phase, we only need to prove that it does not violate consistency when opening an object with a timestamp $O.ts < T.ts$. Note that Algorithm $V3$ forces a validation when opening an object even when $O.ts = T.ts$, which is not the case with Algorithms $V1$ and $V2$.

The proof that opening an object when $O.ts \leq T.ts$ without a validation is identical to the proof of Theorem 1, differing points being the possible places to generate the timestamp written into O are line 4, line 7 and line 9 in function $V3\ COMMIT(T)$.

- $O.ts$ is generated at line 4: the CAS at line 5 succeeded. Since $T_2.ts > T_1.ts + 1 \Rightarrow t_{T2ts} > t(line\ 5) \Rightarrow$ $t_{T2ts} > t_{AQ}$
- $O.ts$ is generated at line 7: the CAS at line 5 failed. Since $T_2.ts > TSNew \Rightarrow t_{T2ts} > t(line\ 5) \Rightarrow t_{T2ts} >$ t_{AQ}
- $O.ts$ is generated at line 9: since $T_2.ts > TSC$ at $t(line\ 9) \Rightarrow t_{T2ts} > t(line\ 9) \Rightarrow t_{T2ts} > t_{AQ}$

We also note that testing for equality when opening an object is necessary. The reason is the assignment at line 9. It is possible that $t_{T2ts} < t_{AQ}$ and $T_2.ts = O.ts = TS$ (at line 9), which would allow $T1$ to modify the object O and commit, and also allow T_2 to open the modified O without validation if the equality was allowed in function $V3\ OPEN$. This could lead to a *write-read* conflict if T_2 had already opened before t_{T2ts} a different object O_2 that is also being written by T_1. □

Theorem 4. *Algorithm $TL\ II$ counter version satisfies the consistency requirement of transactional memory.*

Proof. Algorithm *$TL\ II$ counter version* (the Commit phase of the Transactional Locking II validation strategy) is a more conservative version of the algorithm $V1$, which allows the transaction an opportunity to abort when the CAS operation fails, if it has been invalidated in the meantime by some other transaction. The consistency proof is nearly identical to the proof of Theorem 1. □

Theorem 5. *Algorithm $V4$ satisfies the consistency requirement of transactional memory.*

Proof. The proof that opening an object when $O.ts \leq T.ts$ without a validation is similar to the proof of Theorem 1, with an exception that the possible places to generate the timestamp written into O are line 6 and line 8 in function $V4\ COMMIT(T)$.

- $O.ts$ is generated at line 6: the CAS at line 6 failed $\Rightarrow TSC$ has been changed (by some other thread) between $t(line\ 3)$ and $t(line\ 6) \Rightarrow$ the only way for $T_2.ts$ to obtain a value of TSC that is greater or equal to TS at line 6 is if $t_{T2ts} \geq t(line\ 3) \Rightarrow t_{T2ts} > t_{AQ}$.
- $O.ts$ is generated at line 8: the CAS at line 6 was succesfull $\Rightarrow$ the only way for $T_2.ts$ to obtain a value of TSC that is greater or equal to TS at line 8 is if $t_{T2ts} \geq t(line\ 6) \Rightarrow t_{T2ts} > t_{AQ}$.

Since algorithm $V4$ does not always perform a validation during the commit phase, we also need to prove that it does not violate consistency when both tests on line 3 and line 7 succeed.

If the test on line 3 was sucessful then the TSC has not changed since $t_{T1ts} \Rightarrow$ no objects were updated in the whole system since t_{T1ts}. Since T_1 has already performed a validation at t_{T1ts}, no validation is necessary at time $t(line\ 3)$.

If the test on line 7 was sucessful then the CAS was successfull as well $\Rightarrow$ no objects were updated in the whole system since t_{T1ts}. Since T_1 has already performed a validation at t_{T1ts}, no validation is necessary at time $t(line\ 7)$. □

The correctness proofs for $TL\ II\ tuple\ version$ and LSS are done in a very similar manner as $V1$.

While studying the different sequences of operations in the commit phase, we have also made an interesting discovery: the most recent implementation of the Global Commit Counter commit phase as distributed with RSTM does *not* guarantee consistency. We can show this with an example of sequence of events:

Suppose a transaction T_1 reads object O_1 at time t_1 and writes object O_2 at time t_2, while a transaction T_2 writes O_1 after t_1 and reads O_2 before t_2. These two transactions clearly conflict and it should not be allowed that both commit successfully.

Suppose that $T_1.ts$ before entering commit phase was 3. Suppose that T_1 fails at `TRY_COMMIT` because the global counter has been changed to 4 by another transaction unrelated to T_1 and T_2. Then, before T_2 performs the `ACQUIRE`, T_1 successfully validates since T_2 has not acquired O_1 yet. Then before T_1 commits, T_2 updates $T_2.ts$ to 4 (when reading some new object), performs the `ACQUIRE` operation and using `CAS` changes the global counter from 4 to 5 successfully. After this both T1 and T2 can commit successfully.

In short, if a commit phase for an update transaction attempts to avoid performing the last validation, the update to the shared counter has to happen *before* the last validation. One solution is to move lines 4 through 6 ahead of line 3, making the commit sequence for GCC similar to LSS.

4. EXPERIMENTAL RESULTS

We tested our commit sequence designs using Release 2 of the Rochester Software Transactional Memory (RSTM) [8], which offers significant performance gains relative to its predecessor [13]. We evaluated each design alternative with multiple benchmarks and various numbers of threads.

RSTM is a nonblocking library built to support transactional memory in C++. It features visible and invisible read support, eager and lazy object acquire support, and deferred updates. We extend RSTM by adding a timestamp field to the header of a transactional object, and by adding a candidate linearization point field to the transaction descriptor. Figure 1 depicts our modifications to the RSTM metadata.

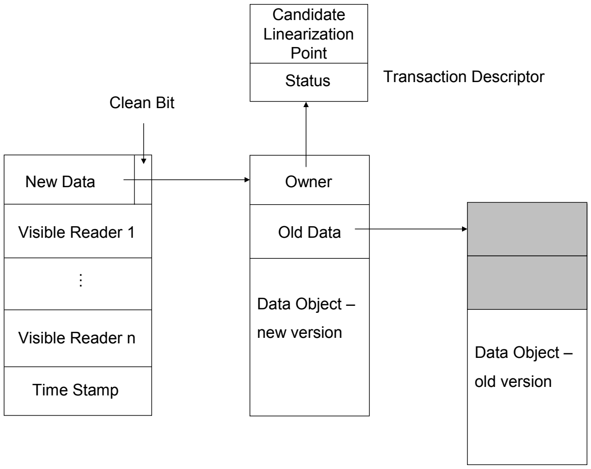

Figure 1: Metadata

We use the following benchmarks in our experiments: a sorted linked list (LinkedList), a sorted linked list with hand-coded early release (LinkedListRelease), a red black tree (RBTree), an undirected graph (RandomGraph), and a hash table (HashTable). These benchmarks are part of the RSTM Release 2 distribution; we have only added a read-only lookup operation to RandomGraph.

For each benchmark, we evaluated two operation mixes: a balanced workload with a mix of one-third each of lookup, insert, and remove operations; and a read-heavy workload with 80% lookups, 10% inserts, and 10% removes.

The RandomGraph benchmark consists of an adjacency list based implementation of a graph. Insert operations add a new node to the graph and connect randomly chosen neighbors to it; delete operations remove a single node from the graph. Transactions in RandomGraph exhibit a high probability of conflicting with one another.

HashTable implements a 256-bucket table that uses overflow chains. The red black tree and linked list benchmarks contains sets of integer values in the range of 0 to 255. LinkedListRelease uses early release to reduce contention for early list nodes.

4.1 Test Methodology

Our test platform is a SunFire 6800 server with 16 UltraSPARC III processors running at 1.2 GHz. We ran each benchmark with various numbers of threads and (where applicable) operation mix ratios, using the standard RSTM 2.0 test driver. We tested with invisible reads, lazy object acquisition, and the Polka contention manager. We report results averaged across three five-second runs; spot checking confirms that using longer test runs does not noticeably alter the results. Due to space limitations, we only list part of our experiment results here (Figure 2). Remaining results are reported in the extended version of this paper [14].

4.2 Discussion

Transactions in the **LinkedList** benchmark spend a large percentage of their execution time in validation, as the lookup, insert and remove functions all must traverse the list from the beginning to validate past reads and lazy writes. In LinkedList, versions 1, 2, 4, TL II counter, and LSS achieve

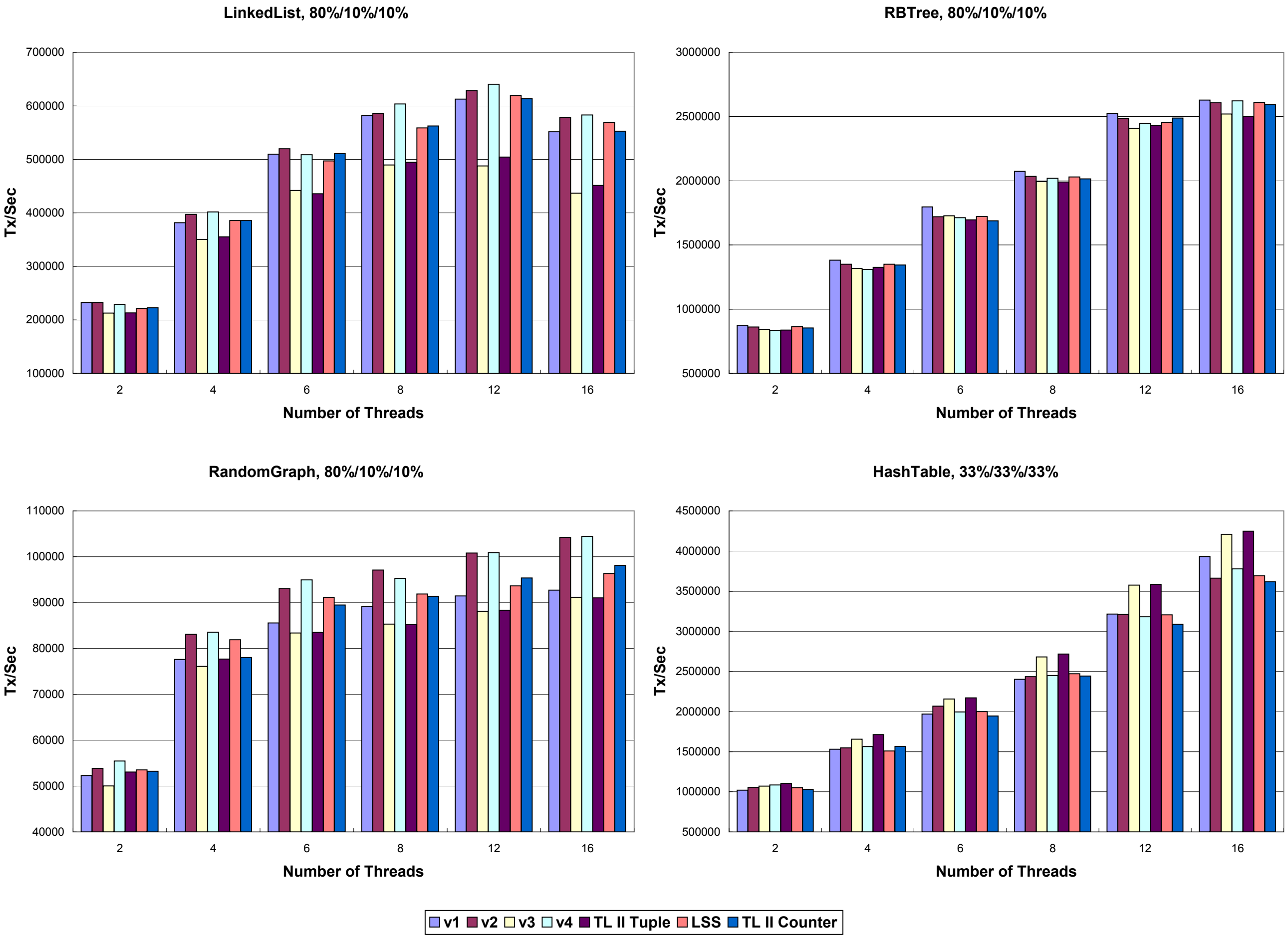

Figure 2: Achieved throughput: LinkedList, RBTree, RandomGraph, and HashTable

better performance then version 3 and TL II tuple. We attribute the poor performance of version 3 to its need to validate when opening an object with a timestamp matching the transaction's current candidate linearization point; these extra validations are not needed in versions 1, 2, 4, TL II counter version and LSS. TL II tuple version performs validations for objects with the same timestamps but different thread IDs; the extra validations manifest in the results as decreased throughput. The biggest performance difference we observe in LinkedList is 33% between V3 and V4 at 16 threads in the read heavy workload.

HashTable features a very large degree of parallelism, since transactions are usually short and disjoint. Transactions typically access only a small constant number of objects; hence validation is inexpensive, especially compared with benchmarks such as LinkedList. This leads to better performance from commit sequences that do not serialize all updates and that minimize runtime overhead.

We observe a difference of about 16% between TL II tuple version and TL II counter version at 16 threads in the balanced workload. We atrtibute the large performance drop from TL II counter version to its serialization of transactions when updating the counter. Designs that reduce contention on the shared counter by allowing shared timestamps result in an improvement; the value of storing a thread ID stored with each object in TL II is clearly visible. Also allowing shared timestamps, version 3 nearly matches the performance of TL II; yet it does not incur the same space overhead from adding IDs to objects. Similarly, version 1 also uses non-unique timestamps. Version 2 and LSS update the time counter with every update transaction, a performance bottleneck at higher levels of concurrency.

LinkedListRelease uses early release to trim the working set size of transactions; the cost of validation reflects this smaller number of objects. This reduces the inherent cost of validation operations which in turn means that designs that attempt to eliminate a final validation in the commit sequence see little payoff. Further, the overall length of transactions is long enough that contention on the shared timestamp counter is very limited, even at high levels of concurrency. For these reasons, all designs give similar performance on this benchmark.

The tree structure of **RBTree** enables greater parallelism among transactions when they modify disjoint subtrees. Since the tree's red-black properties limit its maximum leaf depth to a logarithmic factor of the number of nodes, transactions are very short. We observe some performance loss at 16 threads with versions 3 and TL II tuple, due to their need for an additional validation when initially opening an object. **RandomGraph**'s behavior is similar to LinkedList: Its operations involve linear searches over adjacency lists in the graph data structure; hence, validation contributes a large part of its overhead. Further, its transactions have a high probability of conflict, and taking the time to perform a validation increase a window of opportunity in which another transaction can discover conflict with one that is in the process of committing and abort it. So designs that seek to eliminate the final validation are particularly beneficial here. We observe around a 15% difference in throughput across levels of concurrency.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an in-depth analysis of the Commit phase in timestamp-based Software Transactional Memory implementations. We have shown that forcing the increment of the global shared counter in the commit phase may force unnecessary extra validation in some cases. We have also shown that performing the final validation for write transactions may be unnecessary in some cases.

We have presented several variants of the commit sequence that either avoid forced updates to the shared global counter (and thereby reduce contention on it), or avoid performing validation in cases when it is safe to do so, or both. We have shown that all the proposed variants of the commit phase preserve an important property of the STM systems: a transaction must never observe an inconsistent state of the shared memory. We have also shown that the most current implementation of the Global Commit Counter validation strategy has an invalid commit phase sequence that potentially allows transactions to observe an inconsistent state of the shared memory, and have proposed a different commit phase sequence that preserves consistency.

We have evaluated the proposed commit sequence variants on a set of transactional memory benchmarks, and shown variations that result in up to a 33% difference in overall system throughput.

As future research, we plan to explore a run-time tuning strategy that chooses an appropriate commit sequence variant based on the overall system performance, current workload characteristics, and the contention levels.

6. ACKNOWLEDGMENTS

This work has been supported by Microsoft fund R62710-792. We thank Sun Microsystems for donating to the University of Rochester the SunFire 6800 machine used for experiments in this paper. We also thank the Rochester Synchronization Group for helping us with the source code for their RSTM 2 system.

7. REFERENCES

[1] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*. Springer, Sep 2006.

[2] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[3] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*, pages 303–323. LNCS, Springer, Sep 2005.

[4] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[5] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[6] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[8] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT 06': Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[9] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, Sep 2006.

[10] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 221–228, New York, NY, USA, 2007. ACM.

[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] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[13] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC06: 20th Intl. Symp. on Distributed Computing*, 2006.

[14] R. Zhang, Z. Budimlić, and W. N. Scherer III. Commit phase variations in timestamp-based software transactional memory. Technical Report TR08-03, Rice University, February 2008.