Time-Warp: Lightweight Abort Minimization in Transactional Memory

Nuno Diegues Paolo Romano

INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal

ndiegues@gsd.inesc-id.pt romano@inesc-id.pt

Abstract

The notion of permissiveness in Transactional Memory (TM) translates to only aborting a transaction when it cannot be accepted in any history that guarantees correctness criterion. This property is neglected by most TMs, which, in order to maximize implementation's efficiency, resort to aborting transactions under overly conservative conditions.

In this paper we seek to identify a sweet spot between permissiveness and efficiency by introducing the Time-Warp Multi-version algorithm (TWM). TWM is based on the key idea of allowing an update transaction that has performed stale reads (i.e., missed the writes of concurrently committed transactions) to be serialized by "committing it in the past", which we call a *time-warp* commit. At its core, TWM uses a novel, lightweight validation mechanism with little computational overheads. TWM also guarantees that read-only transactions can never be aborted. Further, TWM guarantees *Virtual World Consistency*, a safety property that is deemed as particularly relevant in the context of TM. We demonstrate the practicality of this approach through an extensive experimental study, where we compare TWM with four other TMs, and show an average performance improvement of 65% in high concurrency scenarios.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques - Concurrent Programming

Keywords Software Transactional Memory; Spurious Abort; Permissiveness; Multi-Version

1. Introduction

The advent of multi-cores has motivated the research of paradigms aimed at simplifying parallel programming. Transactional Memory [18] (TM) is probably one of the most prominent proposals in this sense. With TM, programmers are only required to identify *which* code blocks should run atomically, and not *how* concurrent access to shared state should be synchronized to enforce isolation. TMs guarantee correctness by aborting transactions that would otherwise generate unsafe histories [16, 19, 25].

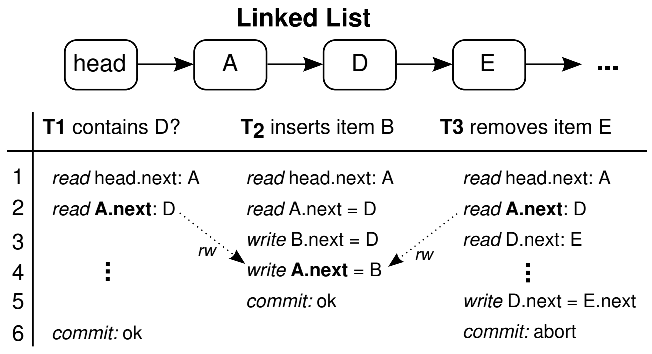
TM implementations achieve this by tracking transparently which memory locations are accessed by transactions. This information is then used to detect conflicts, and possibly abort transactions with the objective of guaranteeing a safe execution. However, to minimize instrumentation's overhead, practical TM implementations suffer of *spurious aborts*, i.e. they abort transactions unnecessarily, even when they did not threaten correctness.

Indeed, existing literature on Software Transactional Memory (STM) has highlighted an inherent trade-off between the efficiency of a TM algorithm, and the number of spurious aborts it produces — the notion of permissiveness [17] was proposed precisely to capture this trade-off. A TM is permissive if it aborts a transaction only when the resulting history (without the abort) does not respect some target correctness criterion (e.g., serializability). Achieving permissiveness, however, comes at a non-negligible cost, both theoretically [21] and in practice [15]. Indeed, most state of the art TMs [9, 10, 13, 14] are far from being permissive, and resort to concurrency control algorithms that generate a large number of spurious aborts, but which have the advantage of allowing highly efficient implementations.

1.1 Problem

To illustrate the problem, consider an example consisting of a sorted linked-list as shown in Fig. 1. This list is accessed by update transactions that insert or remove an element, and by read-only transactions that try to find if a given element is in the list. Let us consider three transactions: a read-only transaction T_1 that seeks element D in the list; an update transaction T_2 that inserts item D; and an update transaction T_3 that removes item E. In the figure we also show a possible execution for the operations of each transaction, and the corresponding result, in a typical STM.

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.
Copyright © 2014 ACM 978-1-4503-2656-8/14/02…$15.00.
http://dx.doi.org/10.1145/2555243.2555259



	T1 contains D?	T2 inserts item B	T3 removes item E
1	*read* head.next: A	*read* head.next: A	*read* head.next: A
2	*read* **A.next**: D	*read* A.next = D	*read* **A.next**: D
3		*write* B.next = D	*read* D.next: E
4	⋮	*write* **A.next** = B	⋮
5		*commit:* ok	*write* D.next = E.next
6	*commit:* ok		*commit:* abort

Figure 1. Possible executions accessing a sorted list.

One widely used form of reducing spurious aborts is by serializing a read-only transaction R before any concurrent update transaction. The intuition is that read-only transactions do not write to shared variables and, consequently, are not visible to other transactions. Thus T_1 is allowed to commit in the example — many STMs skip validation for read-only transactions at commit-time [10, 13, 14] because they can be safely serialized in the past.

Let us now consider T_3, which is an update transaction that modifies shared variables. The execution shown for T_3 dictates its abort in state of the art, *practical* STM algorithms, e.g. [9, 13]. To minimize overheads, these algorithms rely on a simple validation scheme, which allows update transactions to commit only if they can be serialized at the present time (6), i.e. after every other so far committed transaction. This validation mechanism has been systematically adopted by a number of STM algorithms (and database concurrency control schemes [2, 6]), for which reason we refer it as *classic validation* rule. In the example, when T_3 is validated at commit, the *next* pointer of element A is found to have been updated after T_3 read it, causing T_3 to abort. Notice, however, that this abort is spurious, given that T_3 could have been safely serialized "in the past", namely before T_2, yielding the equivalent sequential history $T_1 \rightarrow T_3 \rightarrow T_2$.

On the other hand, serializing update transactions in the past is not always possible, as their effects could have been missed by concurrently committed update transactions. This would be the case, for instance, if T_3 had also attempted to insert element C, missing the concurrent update of T_2 and overwriting B. In such a scenario, a cycle in the serialization graph would arise, and T_3 could not be spared from aborting. Overall, minimizing spurious aborts, in a practical way, requires designing algorithms capable of deciding *efficiently* (i.e., without checking the full conflict graph) when update transactions can be serialized in the past.

1.2 Contribution

In this paper we present an algorithm to efficiently tackle the problem identified above: Time-Warp Multi-version (TWM) is a multi-versioned STM that strikes a new balance between permissiveness and efficiency to reduce spurious aborts.

The key idea at the basis of TWM is to allow an update transaction that missed the write committed by a concurrent transaction T' to be serialized "in the past", namely before T'. Unlike TM algorithms that ensure permissiveness [21, 28], TWM exclusively tracks the direct conflicts (more precisely, anti-dependencies [2]) developed by a committing transaction, avoiding onerous validation of the entire conflicts' graph. Thus, TWM's novel validation is sufficiently lightweight to ensure efficiency, but it can also accept far more histories than state of the art, efficient TM algorithms that only allow the commit of update transactions "in the present" (using the *classic validation* rule).

Furthermore, our TWM algorithm provides *Virtual World Consistency* (VWC) [19], which is a safety criterion that provides consistency guarantees even on the snapshots observed by transactions that abort. This means that TWM prevents typical problems (such as infinite loops and run time exceptions) due to observing inconsistent values not producible in *any* sequential execution.

We present an extensive experimental study comparing TWM with four other STMs representative of different designs, guarantees and algorithmic complexities. This study was conducted on a large multi-core machine with 64 cores using several TM benchmarks. The results highlight gains up to $9\times$, with average gains across all benchmarks and compared TMs of 65% in high concurrency scenarios. The remainder of the paper is structured as follows. In Section 2 we discuss related work. Then we focus on describing the TWM algorithm in Section 3. We elaborate on the correctness of TWM in Section 4 and present our experimental study in Section 5. We conclude in Section 6.

2. Related Work

The growing interest in TM research has led to the development of STMs designed to maximize single-thread performance and reduce bookkeeping overhead [9, 10]. As a consequence, these algorithms are optimized for uncontended scenarios and end up rejecting a large number of serializable schedules (i.e., producing many spurious aborts). An interesting strategy in STMs has been to reduce spurious aborts only for read-only transactions. This idea has been formally characterized as mv-permissiveness [26], and has been used in both single-versioned [3] and multi-versioned [11, 22, 27] TM algorithms. Here, we seek to reduce spurious aborts even further than mv-permissiveness.

Several proposals were designed with the main concern of reducing spurious aborts, ultimately achieving permissiveness [17]. These works target different consistency criteria (serializability, virtual-world consistency, opacity) and pursue permissiveness using both probabilistic and deterministic techniques. Clearly, these design decisions have a strong impact on several important details of these algorithms. Nevertheless, it is still possible to coarsely distinguish them into two classes: i) algorithms [15, 21, 28] that instantiate the full transactions' conflict graph and ensure consistency by ensuring its acyclicity [25]; ii) algorithms [4, 8, 17] that determine the possible serialization points of transactions by using time intervals, whose bounds are dynamically adjusted based on the conflicts developed with other concurrent transactions. Concerning the first class of algorithms, which rely on tracking the full conflict graph, these are generally recognized (often by the same authors [15, 21]) to introduce a too large overhead to be used in practical systems. Analogous considerations apply to interval-based algorithms: as previously shown [17], and confirmed by our evaluation study, these algorithms have costly commit procedures, which hinder their viability in various practical scenarios.

The TWM algorithm leverages on the lessons learnt from prior art and identifies a sweet spot between efficiency (i.e., avoiding costly bookkeeping operations) and the ability to avoid spurious aborts: (1) TWM deterministically accepts many common patterns rejected by practical TM algorithms, by tracking only *direct* conflicts between transactions; and (2) it exploits multi-versioning to further reduce aborts and achieve mv-permissiveness.

TWM also shares commonalities with SSI [7], a technique that enhances snapshot isolation [5] DBMSs to provide serializability. In particular, both schemes track direct (anti-dependency [2]) conflicts between transactions to detect possible serializability violations. However, the two algorithms differ significantly both from a theoretical and a pragmatic standpoint. First, unlike TWM, SSI does not ensure mv-permissiveness (i.e., SSI can abort read-only transactions). Further, SSI was designed to be layered on top, and guarantee interoperability with, a snapshot isolation concurrency control mechanism designed to operate in disk-based DBMS environments. Hence, SSI relies on techniques (e.g., a global lock-table that needs to be periodically garbage collected to avoid spurious aborts) that would have an unbearable overhead in a disk-less environment, such as in TM.

Finally, TWM draws inspiration from Jefferson's Virtual Time and Time-Warp concepts [20], which also aim at decoupling the real-time ordering of events from their actual serialization order. In Jefferson's work, however, Time-Warp is used to reconstruct a safe global state. In TWM, instead, the time-warp mechanism injects "back in time" the versions produced by transactions that observed an obsolete snapshot (to avoid aborting them).

3. The TWM algorithm

Before presenting TWM we introduce preliminary notations.

3.1 Preliminary Notations and Assumptions

As in typical Multi-Version Concurrency Control (MVCC) schemes, TWM maintains a set of versions for each data item k. We refer to data items as variables. A history $\mathcal{H}(\mathcal{S}, \ll)$ over a set of transactions S consists of two parts: a partial order among the set of operations generated by the transactions in S and a version order, $\ll$, that is a total order on the committed versions of each k.

We denote with DSG($\mathcal{H}$) a Direct Serialization Graph over a history $\mathcal{H}$, i.e., a direct graph containing: a vertex for each committed transaction in $\mathcal{H}$; an edge from a vertex corresponding to a transaction T_i to a vertex corresponding to transaction T_j, if there exists a read/write/anti-dependency from T_i to T_j. These edges are labelled with the type of the dependency: (1) $A \xrightarrow{wr} B$ when B read-depends on A because it read one of A's updates; (2) $A \xrightarrow{ww} B$ when B write-depends on A because it overwrote one of A's updates; (3) $A \xrightarrow{rw} B$ when B anti-depends on A because A read a version of a variable for which B commits a new version.

Throughout the description of the algorithm we consider a model with strong atomicity [1]. Considering weak atomicity in multi-versioned TMs is an issue largely orthogonal to the focus of this paper (reducing spurious aborts), and therefore we assume that all accesses to shared variables are transactional and governed by the TM algorithm to ease presentation. We also assume that transactions are statically identified as being read-only. Dynamic, or compiler-assisted, identification of such transactions may be used to this purpose, and is also orthogonal to this work.

3.2 Algorithm Overview

Typical MVCC algorithms [6] allow read-only transactions to be serialized "in the past", i.e., before the commit event of any concurrent update transaction. Conversely, they serialize an update transaction T committing at time t "in the present", by: (1) ordering versions produced by T after all versions created by transactions committed before t; (2) performing the classic validation, which ensures that the snapshot observed by T is still up-to-date considering the updates generated by all transactions that committed before t. We note that this approach is conservative, as it guarantees serializability by systematically rejecting serializable histories in which T might have been safely serialized before T'.

The key idea in TWM is to allow an update transaction to sometimes commit "in the past", by ordering the data versions it produces before those generated by already committed, concurrent transactions. In this case we say that T performs a *time-warp* commit. An example illustrating the benefits of time-warp commits is shown in Fig. 2(a): by adopting a classic validation scheme, B would be aborted because it misses the writes issued by the two concurrent transactions A_1 and A_2; however, B could be safely serialized before both transactions that anti-depend on it, which is precisely what TWM allows for, by time-warp committing B.

To implement the time-warp abstraction efficiently, TWM orders the commit events of update transactions according to two totally ordered, but possibly diverging, time lines. The first time line reflects the *natural commit order* of transactions (or, briefly, *commit order*), which is obtained by

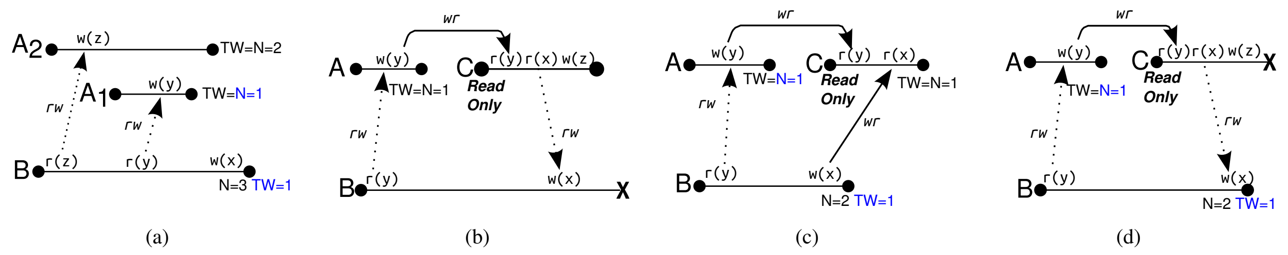


Figure 2. Example histories. The edges are labelled according to the operations they connect.

monotonically increasing a shared logical (i.e., scalar) clock and assigning the corresponding value to each committed transaction. TWM uses this time line to identify concurrent transactions and to establish the visible snapshot for a transaction upon its start. The actual transaction serialization order (and hence the version order) is instead determined by means of a second time line, which reflects what we call the *time-warp commit order* and that diverges from natural commit time order whenever a transaction performs a time-warp commit. TWM keeps track of the two time lines by associating each version of a variable with two timestamps, namely $natOrder$ and $twOrder$, which reflect, respectively, the natural commit and the time-warp commit order of the transaction that created it.

We denote as $\mathcal{N}(T)$, resp. $\mathcal{TW}(T)$, the function (having the set of transactions that commit in $\mathcal{H}$ as domain, and $\mathbb{N}$ as co-domain) that defines the total order associated with the natural, resp. time-warp, commit order. Further, we write $T \prec_{\mathcal{N}} T'$, resp. $T \prec_{\mathcal{TW}} T'$, whenever $\mathcal{N}(T) < \mathcal{N}(T')$, resp. $\mathcal{TW}(T) < \mathcal{TW}(T')$.

We start by discussing how to determine the serialization order of transactions that perform a time-warp commit. Next we describe the transaction validation logic. Finally, we explain how read and write operations are managed.

Time-warp Commit: TWM establishes the time-warp order of a committed update transaction B ($\mathcal{TW}(B)$) as follows:

***Rule* 1.** Consider that B misses the writes of a set of committed transactions A_S that executed concurrently with B (i.e., the transactions in A_S anti-depend on B). Let A be the first transaction in A_S according to the natural commit order. Then $\mathcal{TW}(B) = \mathcal{N}(A)$, which effectively orders B before the transactions in A_S, namely those whose execution B did not witness. The versions of each variable updated by B are timestamped with $\mathcal{TW}(B)$ and added to the versions' list according to the time-warp order.

The above rule is exemplified by the history illustrated in Fig. 2(a): as both A_1 and A_2 perform a regular commit, their time-warp order $\mathcal{TW}$ and natural commit order $\mathcal{N}$ coincide; conversely, as B time-warp commits due to anti-dependency edges developed towards A_1 and A_2, then B is serialized by TWM before A_1 (which commits before A_2 according to $\mathcal{N}$), and is assigned a serialization order $\mathcal{TW}(B) = \mathcal{N}(A_1) = 1$.

Validation Rule: As we will see shortly, the version visibility rule of read-only transactions ensures that these can always be correctly serialized, without the need for any validation phase. Update transactions, conversely, undergo a validation scheme that aims at detecting a specific pattern in the DSG, named *triad*. A triad exists whenever there is transaction T that is both the source and target of anti-dependency edges from two transactions T' and T'' that are concurrent with T (where, possibly, $T' = T''$). We call T a *pivot*, and define the TWM validation scheme as follows:

***Rule* 2.** A transaction fails its validation if, by committing, it would create a triad whose *pivot* time-warp commits.

In other words, TWM deterministically rejects schedules in which two conditions must happen: 1) a pivot transaction T misses the updates of a concurrent transaction T'; and 2) a concurrent transaction T'' (possibly T') misses in its turn the updates of the pivot transaction T. Note that the first condition corresponds to the classic validation rule, and that the second condition (which restricts the set of histories rejected by TWM) is what allows to reduce spurious aborts with respect to state of the art STMs.

Fig. 2(b) exemplifies Rule 2: when B is validated during its commit phase, TWM detects that B is the *pivot* of a triad including also A and C, and it would have to time-warp commit before A. Consequently, B is aborted; this history is indeed non-serializable. Note that B reaches that conclusion by checking solely the direct anti-dependencies it developed, hence avoiding expensive checks for cycles in the entire DSG. Moreover, a pivot must be an update transaction because a read-only transactions is never the target of an anti-dependency.

Read and Write operations: It remains to discuss how TWM regulates the execution of read and write operations. Write operations are privately buffered during transaction's execution phase, and are applied only at commit time, in case the transactions is successfully validated. To determine which versions of a variable a transaction should observe, TWM attributes to a transaction, upon its start, the current value of the shared logical clock. We call this value the start of a transaction ($\mathcal{S}(T)$). TWM uses distinct version visibility rules for read-only and update transactions:

***Rule* 3.** If a read-only transaction T issues a read operation on a variable x, it returns the most recent version of x (according to the time-warp order) created by a transaction T', such that $\mathcal{TW}(T') \leq \mathcal{S}(T)$. If T is an update transaction, it is additionally required that $\mathcal{N}(T') \leq \mathcal{S}(T)$. This prevents update transactions from observing versions produced by concurrently time-warp committed transactions.

The rationale underlying the choice of using different visibility rules for read-only and update transactions is of performance nature. TWM is designed to guarantee that read-only transactions are never aborted. As a consequence, in order to preserve correctness, TWM must ensure that the snapshot observed by a read-only transaction T includes all transactions serialized before T, including time-warp committed ones (see transaction C in Fig. 2(c)). The trade-off is that, in order to be sheltered from the risk of abort, a read-only transaction T must perform visible reads to ensure that concurrent update transactions can detect anti-dependencies originating from T (necessary to implement Rule 2). Fig. 2(b) shows a scenario in which the read-only transaction C commits and, using visible reads, allows pivot B to detect a potential violation of Rule 2, and, hence, to abort. Fig. 2(d) highlights that, without visible reads, B would have no means of detecting a violation of Rule 2, and thus would have committed. In such case, it would be necessary to abort the read-only C (violating mv-permissiveness).

On the other hand, adopting visible reads for update transactions would not render them immune to aborts. Hence, TWM spares them from the cost of visible reads during their execution. Conversely, TWM adopts a lightweight approach ensuring that the snapshot visible for an update transaction T is determined upon its start, and prevent it from reading versions created by concurrent transactions that time-warped. This guarantees that the snapshot observed by T is equivalent to one producible by a serial history defined over a subset of the transactions in $\mathcal{H}$, even if T aborts.

Struct	Attribute	Description
`Var`	readStamp	*ts* of *globalClock* when this `Var` was last read
	latestVersion	pointer to the most recent `Ver` of this `Var`
`Ver`	value	the value of the version
	natOrder	*ts* of the *natural commit order* of the version
	twOrder	*ts* of the *time-warp order* of the version
	nextVersion	pointer to the version overwritten by this one
`Tx`	writeTx	false when this `Tx` is identified as read-only
	readSet	not used in read-only `Tx`
	writeSet	not used in read-only `Tx`
	start	*ts* of the *globalClock* when this `Tx` started
	source	true when another tx anti-depends on `Tx`
	target	true when `Tx` anti-depends on another tx
	natOrder	*ts* of the *natural commit order* of this `Tx`
	twOrder	*ts* of the *time-warp order* of this `Tx`

Table 1. Data structures used in TWM (*ts* = timestamp).

3.3 Pseudo-Code Description

The pseudo-code of the TWM algorithm is reported in Algs. 1 and 2. In Table 1 we describe the metadata used in the pseudo-code for ease of readability.

Any transaction tx starts by reading the global logical clock (*globalClock*), which defines $\mathcal{S}(tx)$. In the READ operation we first check for a read-after-write. Otherwise, if the reader is a read-only transaction, it invokes function SEMIVISIBLEREAD in line 10. For a practical implementation we used a scalar for each variable (attribute *readStamp*). This scalar represents the latest global clock at which some transaction read the variable. This corresponds to a semi visible read scheme, as we do not track individually each reader as other more onerous approaches [4, 15, 17, 21]. To conclude the read operation, we iterate through the versions ordered by $\mathcal{TW}$ until a condition is satisfied that reflects Rule 3.

TWM avoids any validation for read-only transactions. For update transactions, the COMMIT function starts by validating the writes and reads as per Rule 2. When validating a write (function HANDLEWRITE) we first lock the variable and then verify if there existed a concurrent transaction that read any of the variables written by tx, meaning there is an anti-dependence from that reader to tx. When validating a read (function HANDLEREAD) we make the visible read (line 47). Then tx is said to be the source of an edge if tx read a variable and there exists a version for it that was committed after tx started. This means that such version was not in the snapshot of tx and thus an anti-dependency exists from tx to the transaction (say B) that produced that version. In such case, tx tries to *time-warp* commit and serialize before B. In the case that B had set its *source* flag during its previous commit, then tx now fails to commit (as exemplified in Fig. 2(d) with transaction C conducting the validation). In that case, note that B had time-warp committed, so if tx now committed as well, B would become a *pivot* breaking Rule 2. This check is also performed for update transactions during the read operation (line 16) in order to early abort them.

Also note that each anti-dependency, of which tx is the source, is stored locally during the commit procedure (line 54). This is used to implement the *time-warp* commit according to Rule 1 (see line 69). At this point tx aborts only if it raised both flags (`source` and `target` in line 64 and exemplified by Fig. 2(b) with B conducting the validation). Otherwise, $\mathcal{N}(tx)$ is computed by atomically incrementing the global clock and reading it. The new writes are committed and stamped with both $\mathcal{TW}(tx)$ (as its version) and $\mathcal{N}(tx)$ (as the time at which it was created). Function CREATENEWVERSION places each committed write in the list of versions by using $\mathcal{TW}(tx)$ to establish the order. Because this order is non-strict, there may occur *time-warp clashes* between transactions, i.e., $A =_{\mathcal{TW}} B$. For a set of transactions that *time-warp clash* and write to the same variable

Pseudo-code 1 of TWM (1/2).

```
 1: BEGIN(Tx tx, boolean isWriteTx):
 2:   tx.start ← globalClock                              ▷ corresponds to $\mathcal{S}(tx)$
 3:   tx.writeTx ← isWriteTx

 4: READ(Tx tx, Var var):
 5:   if tx.writeTx then
 6:     if ∃ ‹var, value› ∈ tx.writeSet then
 7:       return value                                    ▷ tx had already written to var
 8:     tx.readSet ← tx.readSet ∪ var                     ▷ performed by update txs
 9:   else
10:     SEMIVISIBLEREAD(tx, var, globalClock)             ▷read-only txs
11:     ▷ ensure a concurrent committer sees the visible read
12:     wait until not locked(var)
13:   Ver version ← var.latestVersion
14:   while (version.twOrder > tx.start) ∨               ▷ rule 3 for read-only tx
          (tx.writeTx ∧ version.natOrder > tx.start) do   ▷ write tx
15:     if (tx.writeTx ∧ version.natOrder ≠ version.twOrder) then
16:       abort(tx)                                       ▷ early abort update tx due to rule 2
17:     version ← version.nextVersion
18:   return version.value

19: SEMIVISIBLEREAD(Tx tx, Var var, long ts):
20:   do
21:     long lastRead ← var.readStamp
22:   while lastRead < ts ∧ CAS(var.readStamp, lastRead, ts) = failed

23: WRITE(Tx tx, Var var, Value val):
24:   tx.writeSet ← (tx.writeSet \ ‹var, _› ) ∪ ‹var, val›

25: CREATENEWVERSION(Tx tx, Var var, Value val):
26:   Ver newerVersion ← ⊥
27:   Ver olderVersion ← var.latestVersion
28:   while tx.twOrder < olderVersion.twOrder do
29:     newerVersion ← olderVersion
30:     olderVersion ← olderVersion.nextVersion
31:   if tx.twOrder = olderVersion.twOrder then
32:     return                                            ▷ no tx will ever read this value, skip it
33:   Ver version ← ‹val, tx.natOrder, tx.twOrder, tx, olderVersion›
34:   ▷ insert according to time-warp order...
35:   if newerVersion = ⊥ then
36:     var.latestVersion ← version                       ▷ ...as the latest version
37:   else
38:     newerVersion.nextVersion ← version                ▷ ...or as an older version
```

Pseudo-code 2 of TWM (2/2).

```
39: HANDLEWRITE(Tx tx, Var var):                          ▷ check if tx is the target of an edge
40:   lock(var)
41:   ▷ lines 10, 12 and 40 ensure readers are visible to tx or blocked
42:   if var.readStamp ≥ tx.start then
43:     ▷ detect concurrent transactions that read var
44:     tx.target ← true

45: HANDLEREAD(Tx tx, Var var):                           ▷ check if tx is the source of an edge
46:   ▷ tx can now do visible reads without affecting its validation
47:   SEMIVISIBLEREAD(tx, var, globalClock)
48:   wait until not locked(var) by $tx' \neq$ tx
49:   ▷ check writes committed concurrently to tx's execution
50:   Ver version ← var.latestVersion
51:   while version.natOrder > tx.start do
52:     if version.natOrder ≠ version.twOrder then
53:       abort(tx)                                       ▷ rule 2
54:     tx.antiDeps.add(version.natOrder)                 ▷ used to compute $\mathcal{TW}(tx)$
55:     tx.source ← true
56:     version ← version.nextVersion

57: COMMIT(Tx tx):
58:   if !tx.writeTx then
59:     return                                            ▷ read-only txs never abort
60:   ▷ check for rw edges from/to concurrent txs
61:   $\forall var$ ∈ tx.writeSet do: HANDLEWRITE(tx, var)
62:   $\forall var$ ∈ tx.readSet do: HANDLEREAD(tx, var)
63:   if tx.target ∧ tx.source then
64:     abort(tx)                                         ▷ rule 2
65:   tx.natOrder ← incAndFetch(globalClock)              ▷ compute $\mathcal{N}(tx)$
66:   if (tx.antiDeps = ∅) then
67:     tx.twOrder ← tx.natOrder                          ▷ $\mathcal{TW}(tx) = \mathcal{N}(tx)$
68:   else
69:     tx.twOrder ← min(tx.antiDeps)                     ▷ compute $\mathcal{TW}(tx)$
70:   ∀ ‹$var, value$› ∈ tx.writeSet do:
71:     CREATENEWVERSION(tx, var, value)
72:     releaseLock(var)
```

k, CREATENEWVERSION keeps only the update to k of the transaction T that has the least value for $\mathcal{N}$ (the other transactions execute line 32). In other words, the transactions in a *time-warp clash* are serialized in the inverse order of $\mathcal{N}$, because the one that happened earlier according to the natural commit order was missed by all others in the clash.

3.4 Garbage collection, privatization and lock-freedom

Garbage Collection: The *time-warp* commit mechanism does not raise particular issues for the garbage collection of versions. Indeed, it can rely on standard garbage collection algorithms for MVCC schemes that maintain any version that can possibly be read by an active transaction (as in different implementations in [14, 22, 27]). The key idea of those algorithms is the following: assume that T is the oldest active transaction, with $\mathcal{S}(T) = k$; then versions up to (and excluding) k can be garbage collected — note that the newest version is preserved regardless of this condition.

One may argue that a problematic scenario may arise if some update transaction U *time-warp committed* such that $\mathcal{TW}(U) < k$. For that to happen, there must exist some transaction Z concurrent with U such that: $U \xrightarrow{rw} Z$ and $\mathcal{N}(Z) < k$. But this is impossible because we assumed that T was the oldest active transaction, so Z could not be concurrent with U and obtain *natural commit order* k.

Privatization Safety: Recall that, in our assumptions, we precluded non-transactional accesses to simplify presentation. However, another relevant concern is that of privatization safety [23]. This implies that a transaction P should be able to safely make some shared data only available to it (privatizing it) and work on it without transactional barriers. The challenge here is to ensure that the thread executing P and concurrent transactions do not interfere with each other. However, similarly to the concern of garbage collection, *time-warping* does not present additional challenges to privatization. Existing approaches to support privatization, in fact, are based on the notion of *quiescence*, which forces privatizing transactions to wait for concurrent transactions to finish [22] (using, if possible, explicitly identified privatizing operations to minimize waiting). These techniques suffice to ensure that, once a privatizing transaction P has committed, no transaction can time-warp commit and serialize before P.

Lock-Freedom: Finally, recent work has motivated the adoption of lock-free synchronization schemes to obtain maximum scalability [14, 17], for which reason in the prototype implementation we have used the lock-free commit procedure of [14]. As this concern is orthogonal to our focus, we preserved a simpler presentation with locks, and delegate additional details to our technical report [12].

4. Correctness Arguments

In this section we provide arguments on the correctness of the TWM algorithm. We begin by discussing the serializability of committed transactions in TWM, by showing that the serializability graph of histories accepted by the TWM algorithm is acyclic.Next, we discuss the consistency guarantees provided also to non-committed transactions, namely Virtual World Consistency [19].

4.1 Rejecting Non-Serializable Histories

To prove serializability, we first define a strict total order ($\mathcal{O}$) on the transactions in the committed projection of $\mathcal{H}$ (noted $\mathcal{H}|C$), and then we show that any edge between two transactions in DSG($\mathcal{H}|C$) is compliant with $\mathcal{O}$. The strict total order $\mathcal{O}$ is obtained from the non-strict total order defined by $\mathcal{TW}$, which we recall can have ties in presence of time-warp clashes, breaking ties as follows. We order update transactions in $\mathcal{O}$ using the *time-warp order* and, whenever there is a *time-warp clash*, i.e., $A =_{\mathcal{TW}} B$, we use the natural commit order $\mathcal{N}$ as a tie breaker and serialize B before A in $\mathcal{O}$ iff $A \prec_{\mathcal{N}} B$. This results in a strict total order because $\mathcal{N}$ defines a strict total order as well. Any read-only transaction T is serialized in $\mathcal{O}$ according to $\mathcal{S}(T)$, which surely makes them coincide with some update transaction in $\mathcal{O}$. To tie-break, we place the read-only transactions always later than coinciding update transactions in $\mathcal{O}$. If two read-only transactions obtain the same value (because they started on the same snapshot), any deterministic function suffices as a tie break (for instance, the identifier of the thread that executed the transaction).

In order to prove the acyclicity of DSG($\mathcal{H}|C$), we show that for any committed transactions A and B such that $A \prec_{\mathcal{O}} B$, there cannot be any edge from B to A in the DSG. We prove this claim by contradiction, considering individually each type of edge. First, let us assume that $B \xrightarrow{ww} A \in$ DSG($\mathcal{H}|C$). According to function CREATENEWVERSION this is possible iff $B \prec_{\mathcal{TW}} A$. This, however, contradicts the assumption $A \prec_{\mathcal{O}} B$, because it implies that $A \preccurlyeq_{\mathcal{TW}} B$.

Now let us consider that $B \xrightarrow{wr} A$. First suppose that A is an update transaction. Then, according to line 14, A can read a version created by B iff $\mathcal{N}(B) \leq \mathcal{S}(A)$. However, the time-warp commit timestamp of a transaction is always less or equal than its natural commit timestamp ($\mathcal{TW}(B) \leq \mathcal{N}(B)$); also, an update transaction A can only time-warp due to concurrent transactions, meaning they commit after $\mathcal{S}(A)$ and thus $\mathcal{S}(A) < \mathcal{TW}(A)$. Hence, we obtain $\mathcal{TW}(B) \prec \mathcal{TW}(A)$, contradicting the initial assumption. Now consider that A is a read-only transaction. Then, according to line 14, A can read a version created by B (concurrent with A's execution) iff $\mathcal{TW}(B) \leq \mathcal{S}(A)$. Given that A is a read-only transaction, $\mathcal{TW}(A) = \mathcal{S}(A)$, hence $\mathcal{TW}(B) \leq \mathcal{TW}(A)$. The case $\mathcal{TW}(B) < \mathcal{TW}(A)$ clearly contradicts the initial assumption. If $\mathcal{TW}(B) = \mathcal{TW}(A)$, then we note that A is a read-only transaction that clashes with B; according to the rules we used to derive $\mathcal{O}$ then A is ordered after B in $\mathcal{O}$, which again contradicts the initial assumption ($A \prec_{\mathcal{O}} B$).

Finally we consider that $B \xrightarrow{rw} A$. First assume that B is a read-only transaction. Then the version written by A is not visible to B iff $\mathcal{S}(B) < \mathcal{TW}(A)$. But since B is read-only, then $\mathcal{S}(B) = \mathcal{TW}(B)$, and we once again contradict the initial assumption. Assume now that B is an update transaction, for which we have two possible cases depending on whether B commits before or after A in the *natural commit order*. Consider the first case where $B \prec_{\mathcal{N}} A$. Then B performs some visible read in line 47; later A triggers the condition in line 42 and sets $A.target \leftarrow true$. Consequently A cannot *time-warp commit* or else both *target* and *source* flags would be true and A would abort in line 64. Then $\mathcal{TW}(B) < \mathcal{N}(A) = \mathcal{TW}(A)$, which is a contradiction with the initial assumed order. Lastly, consider the second case where $A \prec_{\mathcal{N}} B$. Then B triggers the condition in line 51. If A *time-warp commits*, then B aborts in line 53. Otherwise, B adds A to it's *antiDeps* set which results in $\mathcal{TW}(B) \leq (\mathcal{N}(A) = \mathcal{TW}(A))$ (according to line 69). The case where $B \prec_{\mathcal{TW}} A$ trivially violates our assumption. The tie-break in the *time-warp clash*, where $B =_{\mathcal{TW}} A$, is broken in the inverse *natural commit order* (recall $A \prec_{\mathcal{N}} B$), which also contradicts the assumption.

4.2 Virtual World Consistency

So far we have argued that TWM ensures serializability for committed transactions. But running (or already aborted) transactions are equally important in TWM because certain phenomena must be prevented with regard to them [16, 19]. If a transaction executing alone is correct, then it should be correct when faced with concurrency under a TM algorithm. This translates to a sense of consistency sufficiently strong in which hazards, such as infinite loops or divisions by zero, are avoided — this is considered an imperative requirement in TM algorithms [10, 16] and it is guaranteed by Virtual World Consistency [19].

VWC is a correctness criterion stronger than serializability, as it prevents transactions from observing snapshots that cannot be generated in any sequential history. Besides serializability for committed transactions, VWC also requires that, for every aborted or running transaction T, there is a legal linear extension of partial order *past(T)*, where *past(T)* is obtained from the sub-graph of DSG($\mathcal{H}$) containing all the transactions on which T transitively depends, and removing any anti-dependencies. A legal linear extension of *past(T)* is a linear extension $\widehat{S}(T)$ of *past(T)* where every transaction $T' \in past(T)$ observes values written by the most recent transaction that precedes T' in $\widehat{S}(T)$.

Recall that we have argued the absence of cycles in DSG($\mathcal{H}|C$). Note that *past(T)* is a subgraph of DSG($\mathcal{H}$), on which non-committed transactions are also considered; but they must be sinks in that subgraph (because anti-dependencies are removed) and thus we also argue that *past(T)* is also acyclic. It then follows that a linear extension $\widehat{S}(T)$ of *past(T)* must exist. $\widehat{S}(T)$ is legal because transactions read the most recent version committed according to $\mathcal{TW}$ (see line 14). But, since *past(T)* respects the $\mathcal{TW}$ order, we get that T must be legal and so we argue that TWM provides VWC.

Another similar, albeit stronger, correctness criterion is that of opacity. In the following we discuss why TWM does not guarantee opacity [16], and then explain how TWM might be adapted to ensure this property. Briefly, the opacity specification requires 2 properties: O.1) the existence of an equivalent serial history $\mathcal{H_S}$ that preserves the real-time order of $\mathcal{H}$; O.2) that every transaction in $\mathcal{H_S}$ is legal. TWM does respect property O.1 (not shown here for space constraints). Concerning property O.2, we note that two concurrent transactions R and W can perceive two different serialization orders — this is a consequence of the different version visibility conditions in line 14 according to the nature of the transaction. These two orders, denoted respectively as $\mathcal{H}_S^R$ and $\mathcal{H}_S^W$ for transactions R and W, exist in case a third concurrent transaction A time-warp commits before R and W. In this case, A may be included in $\mathcal{H}_S^R$ but not included in $\mathcal{H}_S^W$. But then, in such case, TWM would abort W due to line 53, thus not endangering serializability. Then, this makes $\mathcal{H}_S^W$ a legal sequential history, but it is incompatible with the serial history equivalent to $\mathcal{H}$, which we denoted as $\mathcal{H_S}$. This is why TWM does not abide by property O.2.

We stress that the fact that $\mathcal{H}_S^R$ and $\mathcal{H}_S^W$ may not be compatible is acceptable by VWC. This is because any transaction in $\mathcal{H}_S^W$ that is not compatible with $\mathcal{H_S}$ aborts, and in VWC aborted transactions can observe legal linear extensions of different causal pasts. We also remark that it would be indeed relatively straightforward to adapt TWM to ensure property O.2, and hence opacity: it would be sufficient to homogenize the logic governing the execution of read operations for both read-only and update transactions, allowing update transactions to observe the snapshots generated by concurrent transactions and forcing them to use visible reads, just like read-only transactions. As discussed in Section 3, the choice of using non-visible reads for update transactions is motivated by performance considerations. Indeed, by adopting VWC rather than opacity as reference correctness criterion, it is possible to maximize its efficiency via lightweight conflict tracking mechanisms, while still providing robust guarantees concerning the avoidance of unexpected errors due to inconsistent/partial reads.

5. Evaluation

In this section we experimentally evaluate the performance of a Java-based implementation of TWM. To access its merit, we compare it with four other STMs representative of different designs: (1) JVSTM [14] is multi-versioned and guarantees abort-freedom for read-only transactions; (2) TL2 [10] is a simpler TM based on timestamps and locks; (3) NOrec [9] uses a single word for metadata (a global lock), thus being even simpler than TL2; and (4) AVSTM [17] is also single-version, but on top of that it is also probabilistically permissive with regard to opacity. This allows to contrast TWM directly against a different design that minimizes spurious aborts (AVSTM); against TMs representative of single-thread efficient designs (TL2 and NOrec); and against a multi-versioned TM (JVSTM). Note that both JVSTM and AVSTM are lock-free (similarly to our prototype of TWM, as mentioned in Section 3.4), whereas TL2 and NOrec are lock-based. Finally, TWM and AVSTM exploit alternative mechanisms to validate transactions, whereas the others rely on the *classic validation*.

We used Java implementations for all the STMs, by obtaining the code for JVSTM from its public repository, TL2 and NOrec from their respective ports to the Deuce framework, and by porting AVSTM to Java. All implementations were modified to share a common interface that uses manual instrumentation relying on a concept similar to that of `VBoxes` [14]. This means that the benchmarks were manually instrumented to identify shared variables and transactions, resulting in an equal and fair environment for comparison of all TMs. We also identified read-only transactions in the benchmarks, and allowed implementations to take advantage of this when possible. This means that TWM, JVSTM and TL2 do not maintain read-sets for such transactions and their commit procedure needs no validation. NOrec requires the read-set for re-validation of a transaction T when the global clock has changed, and AVSTM requires it for an update transaction T that is committing to update the validity interval of concurrent transactions T' that read items committed by T.

In the following experimental study we seek to answer the following questions: (1) What is the performance difference of TWM to each of the other design class of STM? (2) Where does the difference in performance come from? (3) What is the overhead in reducing aborts with respect to the classic validation?

To answer the above questions, we conducted experiments on a variety of benchmarks and workloads. We first present results with a classic micro-benchmark for TM, namely Skip List. Then, we consider the more complex and realistic benchmarks from the STAMP suite [24][1]. STAMP contains a variety of transactions, with different sizes and contention levels. However, contrarily to the Skip List micro-benchmark, STAMP does not have read-only transactions, which is a disadvantage to multi-version TMs. The following results were obtained on a machine with four AMD Opteron 6272 processors (64 total cores), 32GB of RAM, running Ubuntu 12.04 and Oracle's JVM 1.7.0_15 and each data point corresponds to the average of 10 executions. Finally, we use use the geometric mean when we show averages over normalized result and use as abort rate metric the ratio of number of restarts to the number of executions (encompassing committed and restarted transactions).

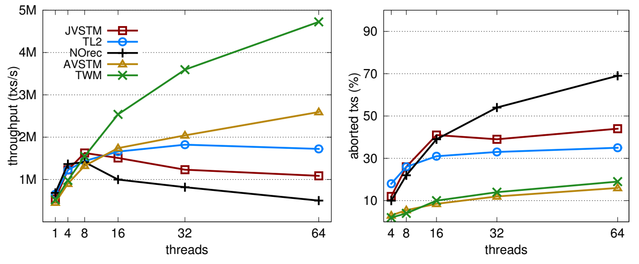
5.1 Skip List

We begin by studying the behavior of time-warping in a traditional data-structure. As described in Section 1.1, concurrent traversals and modifications in data-structures, such as a skip-list, are perfect examples of the advantages of time-warping: a transaction T_1 modifying an element near the end of the list need not abort because a concurrent transaction T_2 modified an element in the beginning of the list and committed; TWM can automatically, and safely, commit T_1 before T_2, whereas *classic validation* precludes the commit of T_1.

For this micro-benchmark we used the source code available in the IntSet benchmark in the Deuce framework. We set up the skip-list with 100 thousand elements and 25% update transactions that either insert or remove an element. Fig. 3(a) shows the scalability results for this workload, where TWM performs best after 16 threads, and below that is competitive with the other TMs. At 64 cores TWM achieves the following speedups: $2.8\times$ for TL2; $9.4\times$ for NOrec; $4.3\times$ for JVSTM; and $1.8\times$ for AVSTM. It is actually interesting to assess that, at a low thread count, NOrec performs best. However, this difference quickly fades at a low thread count and its performance plunges due to the overly pessimistic validation procedure — this is visible on Fig. 3(b) where its abort rate quickly grows to approximately 70%. Note that JVSTM's update transactions incur in a significant cost due to the multi-version maintenance — this cost is amplified by the non-negligible percentage of update transactions, which have no advantage in the availability of multi-versions. TWM, instead, takes advantage of multi-versions even for update transactions due to time-warping.

Overall, as we can see in Fig. 3(b), the source of our gains is two-fold: TWM clearly aborts much less transactions than *classic validation* TMs; on the other hand, despite TWM aborting slightly more than AVSTM, it introduces a much lower overhead, as we will discuss in detail next.

[1] Additional experiments are available in our technical report [12].



(a) Throughput in SkipList. (b) Abort percentage in SkipList.

Figure 3. SkipList with 25% modifications.

5.2 Overhead Assessment

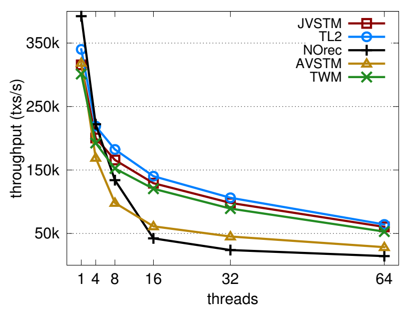
To better understand the nature of each design, we conducted worst-case experiments to assess the cost of reducing spurious aborts. We first conducted an experiment with two shared variables, both incremented once by every transaction, to create a scenario with very high contention, whose conflict patterns cannot be accommodated by the TWM algorithm (as well as by the other considered TMs).

We can see the throughput for this experiment in Fig. 4(a), where the slowdown of TWM is comparable to that of JVSTM and TL2, being 7% and 12% worse with respect to those two TMs. Both AVSTM and NOrec perform worse beyond 8 threads due to the internal validation procedures — we shall see this in detail next.

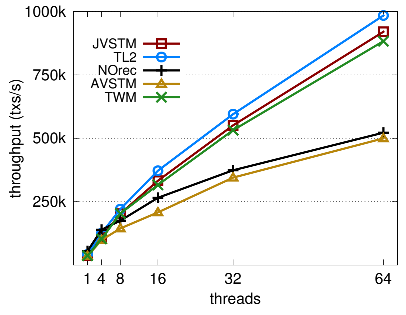
We also modified the SkipList micro-benchmark to have each thread modify an independent skip-list. Consequently, no transaction ever runs into conflicts, although they still activate the validation procedures as every transaction performs some writes. The results of this experiment are shown in Fig. 4(b). As expected, every TM is able to scale as this scenario is conflict-free. Moreover, the relative trends are consistent with those observed for the highly-contended scenario with the shared counters.

To better understand the previous results, we instrumented the prototypes to collect the time spent by transactions on each phase of the TM algorithm. Fig. 4(c) shows the results relative to the previous experiment. We considered four different phases: the *read* corresponds to time spent in read barriers; *readSet-val* and *writeSet-val* are the validations conducted by the transaction, including those at a commit-time and when executed during the transaction execution in the case of NOrec — note that the write-set validation only exists in the case of TWM and AVSTM; and finally *commit* corresponds to the rest of the time spent in the commit phase (for instance, writing-back, or helping other transactions in the case of lock-free schemes).

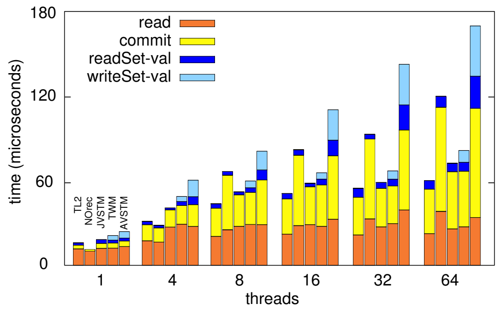
In this plot, we see that the commit is generally the main source of overhead as the threads increase. TL2 obtains the least overhead because transactions are conflict-free and the workload is write-intensive, which implies extra costs for schemes that minimize aborts and for multi-version algorithms. Initially, NOrec also benefits from these circum-



(a) Incrementing two shared counters.



(b) SkipList without contention.



(c) Overhead breakdown for SkipList.

Figure 4. Overhead assessment with 100% writes.

stances to yield the least overhead. However, it quickly becomes less efficient as the global commit lock becomes a bottleneck and the commit time increases significantly due to threads waiting to commit. Moreover, its read-set validation time also increases because transactions re-validate the read-set when they notice the global clock has changed (due to an update transaction committing).

On the other hand, the lock-free schemes also incur in some overhead right from the start. Both TWM and AVSTM conduct additional validations that are useless in this scenario, as it is conflict-free, and are noticeably making them more expensive. Yet, TWM preserves the overheads rather low as the scale increases, whereas AVSTM suffers considerably as we reach 64 threads, making it the most expensive TM at that scale, slightly above NOrec. The main culprit for this cost in AVSTM is that a committing update transaction must possibly update metadata of every concurrent transaction. As a result of this onerous check, the commit and validations cost grow considerably with the number of threads.

Finally, we highlight that TWM's overheads are consistently close to those of JVSTM. They are also both higher than those of TL2 due to the management of multi-versions and lock-freedom guarantees. Yet, TWM manages to reduce spurious aborts with respect to both JVSTM and TL2. This illustrates the appeal of TWM, which escapes the overheads of aiming for permissiveness, while improving performance in high concurrency scenarios.

5.3 Application Benchmarks

In this section we present additional experiments with larger benchmarks to demonstrate the ability of TWM to reduce spurious aborts.

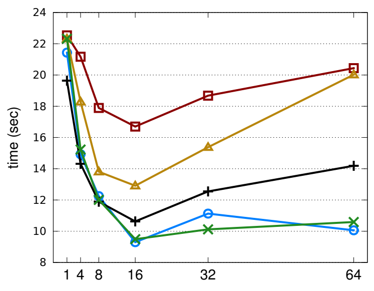
For that, we studied the performance of these TMs in STAMP, for which we used an existing port to Java in Deuce framework. Fig. 5 presents the time to complete each benchmark, excluding Yada (not available in the Java port) and Bayes (excluded given its non-determinism). Note that in these plots lower is better.

TWM behaves slightly worse than JVSTM and TL2 in both Intruder and Kmeans with an average slowdown of 7%. On the other benchmarks, it is either on par with the best TM (Genome, SSCA2, Vacation (low)) or it obtains improvements over all TMs (Labyrinth and Vacation (high)). We have manually inspected each benchmark to understand if there are opportunities for time-warp to reduce spurious aborts: this is the case for Genome, Labyrinth and Vacation. The other three only generate simple conflict patterns that cannot be surpassed with time-warping. Yet, TWM manages to perform among the best TMs in every benchmark. Conversely, AVSTM only obtains considerable improvements in Vacation (high), although it still performs worse than TWM.

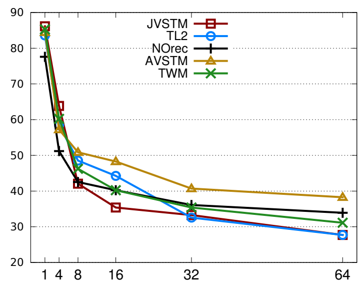
Fig. 5(i) shows the geometric mean (and deviation) of the speedup of TWM relative to the other TMs across all the STAMP benchmarks. The overall trend is that TWM is more beneficial than *classic validation* TMs as the thread count increases. The average improvement across all the benchmarks is 31% over JVSTM, 12% over TL2, 16% over NOrec and 21% over AVSTM. Additionally, if we only consider the benchmarks with possibility of time-warping, TWM obtains an average improvement of 36% over JVSTM, 37% over TL2, 41% over NOrec, and 37% over AVSTM. Note that the gains over AVSTM are mostly due to a more efficient algorithm, rather than by abort reduction (as shown in Table 2).

6. Final remarks

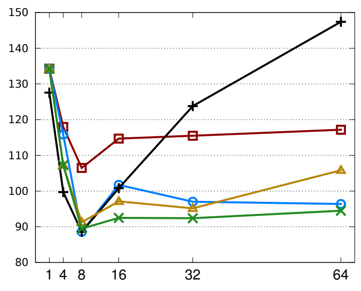
This paper presented TWM, a novel multi-version algorithm that aims at striking a balance between permissiveness and efficiency. TWM exploits the key idea of allowing update transactions to be serialized "in the past", according to what we called a *time-warp* time line. This time line diverges from the natural commit order of transactions in order to allow update transactions to commit successfully (but in the past) despite having performed stale reads. Past solutions have tried to maximize permissiveness via costly and inefficient procedures. TWM explored a new validation strategy that results in less aborts, without hindering efficiency (e.g., by avoiding expensive checks of the transactions' dependency graph). Furthermore, TWM ensures mv-permissiveness and VWC. Our experiments comparing a variety of TMs evidenced the



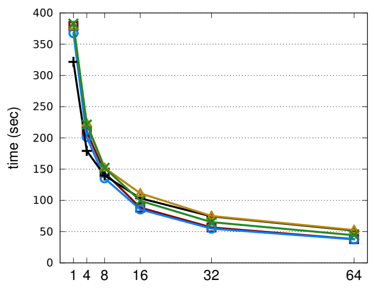
(a) Genome.



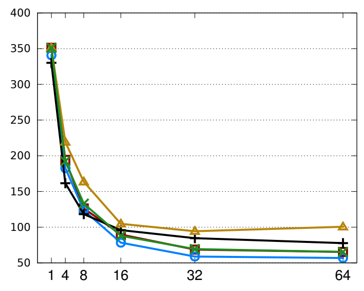
(b) Intruder.



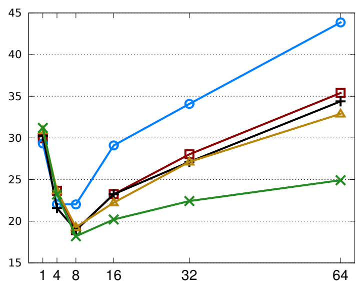
(c) SSCA2.



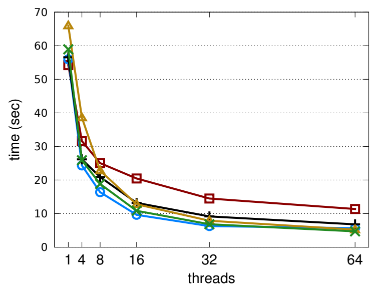
(d) Kmeans (low).



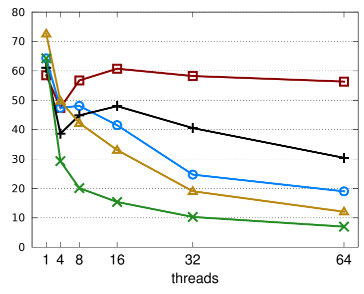
(e) Kmeans (high).



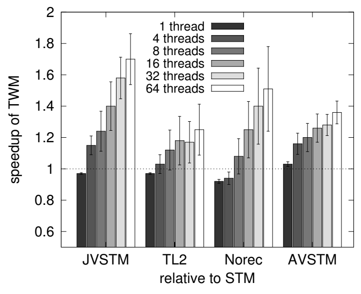
(f) Labyrinth.



(g) Vacation (low).



(h) Vacation (high).



(i) Geometric mean speedups.

Figure 5. Scalability in the STAMP benchmarks.

	Benchmark							
STM	genome	intruder	kmeans-l	kmeans-h	labyrinth	ssca2	vac-l	vac-h
TWM	3.8	3.8	1.4	4.2	8.8	10.5	6.4	17.8
JVSTM	15.4	3.2	1.6	4.9	12.3	11.3	12.1	41.1
TL2	12.1	4.8	3.8	3.4	13.8	11.7	10.0	41.4
NOrec	21.1	6.0	3.8	6.4	27.6	14.9	19.9	55.0
AVSTM	13.0	3.5	2.6	4.8	10.4	11.5	9.4	18.9

	Threads				
STM	4	8	16	32	64
TWM	1.2	4.4	6.6	9.9	15.7
JVSTM	1.8	7.0	10.2	15.7	21.2
TL2	2.6	6.5	11.4	16.1	20.9
NOrec	3.4	9.6	18.6	24.9	34.0
AVSTM	2.5	5.5	8.6	12.7	17.6

Table 2. Average abort rate (%) across each STAMP benchmark (left) and each thread count (right).

merits of time-warping with an average improvement of 65% in high concurrency scenarios and gains extending up to 9×. Further, we showed that TWM introduces very limited overheads when faced with contention patterns that cannot be optimized using TWM. We opted for a multi-version scheme for time-warping, although part of our ideas can also be applied to single-versioned TMs. The extent to which that can be advantageous is interesting future work.

The recent release of hardware support for TM is another source of interesting open questions. Hardware vendors have opted for a paradigm of best-effort semantics for the first generation of Hardware TMs, in which no guarantee is given that a transaction will ever complete successfully. One of the main reasons for such weak semantics is the difficulty in dealing with arbitrarily large transactions, while preserving a simple hardware design. The proposed alternative is thus to use software fallback paths, namely to an STM implementation. Consequently, it is interesting to explore the integration of STM implementations with reduced spurious aborts, such as TWM, in the fallback paths for hardware implementations. The difficulty is in the efficient integration of both systems, which becomes more challenging with the metadata and validations conducted in such STM algorithms. We hope to provide answers to this problem in our future work.

Acknowledgements.

This work was supported by national funds through FCT — Fundação para a Ciência e Tecnologia — under project PEst-OE/EEI/LA0021/2013, and by specSTM project (PTDC/EIA-EIA/122785/2010).

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 185–196, 2009.

[2] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[3] H. Attiya and E. Hillel. Single-version STMs can be multi-version permissive. In *Proceedings of the 12th International Conference on Distributed Computing and Networking*, ICDCN, pages 83–94, 2011.

[4] U. Aydonat and T. Abdelrahman. Relaxed Concurrency Control in Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 23(7):1312–1325, 2012.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of SIGMOD*, pages 1–10, 1995.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing, Boston, MA, USA, 1987.

[7] M. Cahill, U. Röhm, and A. Fekete. Serializable isolation for snapshot databases. In *Proceedings of SIGMOD*, pages 729–738, 2008.

[8] T. Crain, D. Imbs, and M. Raynal. Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP, pages 244–257, 2011.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 67–78, 2010.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th Symposium on Distributed Computing*, DISC, pages 194–208, 2006.

[11] N. Diegues and J. Cachopo. Practical Parallel Nesting for Software Transactional Memory. In *Proceedings of the 27th Symposium on Distributed Computing*, DISC, pages 149–163, 2013.

[12] N. Diegues and P. Romano. Enhancing permissiveness in transactional memory via time-warping. Technical Report 16, december 2013.

[13] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 237–246, 2008.

[14] S. M. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 179–188, 2011.

[15] V. Gramoli, D. Harmanci, and P. Felber. On the Input Acceptance of Transactional Memory. *Parallel Processing Letters*, 20(1), 2010.

[16] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 175–184, 2008.

[17] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in Transactional Memories. In *Proceedings of the 22nd Symposium on Distributed Computing*, DISC, pages 305–319, 2008.

[18] M. Herlihy and J. E. B. Moss. Transactional Memory: architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA, pages 289–300, 1993.

[19] D. Imbs and M. Raynal. Virtual World Consistency: A condition for STM systems. *Theoretical Computer Science*, 444 (0):113–127, 2012.

[20] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages Systems*, 7(3):404–425, July 1985.

[21] I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. In *Proceedings of the 21st Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 59–68, 2009.

[22] L. Lu and M. L. Scott. Generic Multiversion STM. In *Proceedings of the 27th Symposium on Distributed Computing*, DISC, pages 134–148, 2013.

[23] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, ICPP, pages 67–74, 2008.

[24] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Symposium on Workload Characterization*, IISWC, pages 35–46, 2008.

[25] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[26] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th Symposium on Principles of Distributed Computing*, PODC, pages 16–25, 2010.

[27] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: Selective Multi-Versioning STM. In *Proceedings of the 25th Symposium on Distributed Computing*, DISC, pages 125–140, 2011.

[28] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an stm. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 163–172, 2009.