Enhancing Permissiveness in Transactional Memory via Time-Warping

Nuno Diegues
INESC-ID/IST
ndiegues@gsd.inesc-id.pt

Paolo Romano
INESC-ID/IST
romano@inesc-id.pt

September 2013
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 a target correctness criterion. This theoretically powerful property is often neglected by state-of-the-art TMs, which, in order to maximize implementation’s efficiency, resort to aborting transactions under overly conservative conditions. As a result, they risk rejecting a significant number of safe histories.

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 that can be implemented with minimal bookkeeping and computational overheads, thus maximizing efficiency and ensuring practical viability. TWM also ensures mv-permissiveness by guaranteeing 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: we compare TWM with four other TMs, representative of typical alternative design choices, and on a wide variety of benchmarks. This study shows an average performance improvement across all considered workloads and TMs of 65% in high concurrency scenarios, with gains extending up to 9× with the most favourable benchmarks. These results are a consequence of TWM’s ability to achieve drastic reduction of aborts in scenarios of non-minimal contention, while introducing negligible overheads (approximately 10%) even in worst-case, synthetically designed scenarios (i.e., no contention or contention patterns that cannot be optimized using TWM).

Keywords: Software Transactional Memory, Spurious Abort, Permissiveness
1. Introduction

The advent of multicore has motivated the research of paradigms aimed at simplifying parallel programming. Transactional Memory [19] (TM) is probably one of the most prominent proposals in this sense, especially given the fact that it has been recently adopted by major industrial players [34, 21]. 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. The TM is then responsible for guaranteeing correctness, by aborting transactions that would otherwise generate unsafe histories [30, 17, 20]. 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 implementations of TM tend to suffer of spurious aborts, i.e. they can abort transactions unnecessarily, even when they did not threat 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 [16] 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 [23] and in practice [15]. Indeed, most state of the art TM systems [10, 11, 13, 14, 9] are far from being permissive, and resort to using concurrency control algorithms that can generate a large number of spurious aborts, but which have the key 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.

One widely used form of reducing spurious aborts is by serializing a read-only transaction $R$ before any concurrent update transaction. The intuition behind this idea is that read-only transactions do not abort and therefore do not cause spurious aborts.
write to shared variables and, consequently, are not visible to other transactions in the execution (both concurrent or future). Thus $T_1$ is allowed to commit in the example — many STMs skip validation for read-only transactions at commit-time [10, 11, 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. [10, 11, 9, 14]. In order 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 [6, 2]), for which reason we refer it as classic validation rule. In the example, when $T_3$ is validated at commit time, the next pointer of element $A$ is found to have been updated after $T_3$ read it, causing $T_3$ to abort. It should be noted, 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. Consequently, the key problem to address to minimize spurious aborts is to design algorithms capable of deciding efficiently (i.e., without checking the full transactions’ 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 algorithm 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 [33, 23], 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). Also noteworthy is that TWM ensures abort-freedom for read-only transactions.
Furthermore, our TWM algorithm provides Virtual World Consistency (VWC) [20], 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 multicore machine with 64 cores using several TM benchmarks. The results highlight gains up to 9×, 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 [10, 14, 11]. 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 [32], and has been used in both single-versioned [3] and multi-versioned [31, 14, 26] TM algorithms. In this paper 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 [16]. 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 [33, 15, 23] that instantiate the full transactions’ conflict graph and ensure consistency by ensuring its acyclicity [30]; ii) algorithms [16, 4, 8] 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, 23]) to introduce a too large overhead to be used in practical systems. Analogous considerations apply to interval-based algorithms: as previously shown [16], 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. As a consequence SSI can rely 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 a TM system.

Finally, TWM draws inspiration from Jefferson’s Virtual Time and Time-Warp concepts [22], 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 $\mathcal{H}(S, \prec)$ 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, $\prec$, 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) $A \xrightarrow{\text{rr}} B$ when $B$ read-depends on $A$ because it read one of $A$’s updates; (2) $A \xrightarrow{\text{ww}} B$ when $B$ write-depends on $A$ because it overwrote one of $A$’s updates; (3) $A \xrightarrow{\text{rw}} B$ 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 the versions produced by \( T \) after all versions created by transactions committed before \( t \); and (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 a conservative one, 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 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 timelines 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 \( \mathbb{N}(T) \), resp. \( \mathcal{T}\mathbb{W}(T) \), 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 \( T \prec_{\mathbb{N}} T' \), resp. \( T \prec_{\mathcal{T}\mathbb{W}} T' \), whenever \( \mathbb{N}(T) < \mathbb{N}(T') \), resp. \( \mathcal{T}\mathbb{W}(T) < \mathcal{T}\mathbb{W}(T') \).

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 \( B \), namely \( \mathcal{T}\mathbb{W}(B) \), 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 \( \mathcal{T}\mathbb{W}(B) = \mathbb{N}(A) \), 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 \( \mathcal{T}\mathbb{W}(B) \) 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 $TW$ and natural commit order $N$ 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 $N$), and is assigned a serialization order $TW(B) = N(A_1) = 1$.

**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, which we denote as $S(T)$. 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 $TW(T') \leq S(T)$. If $T$ is an update transaction, it is additionally required that $N(T') < S(T)$. 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.

### 3.3. Pseudo-Code Description

The pseudo-code of the TWM algorithm is reported in Algs. 3 and 4. In Table 3 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 $S(tx)$. In the READ operation we first check if the variable to be read was already written by the transaction (as writes are buffered). If the reader is a read-only transaction, it invokes function $\text{visibleRead}$ in line 15. For a practical implementation of $\text{visibleRead}$ we used a single scalar associated with each variable (attribute $\text{readStamp}$). This scalar represents the latest global clock at which some transaction read the variable. Conceptually adding a transaction to the set of readers is implemented by applying the current clock only if it is larger than the latest visible read (line 27). To conclude the read operation, we iterate through the versions ordered by $TW$ until a condition is satisfied that reflects Rule $3$.

TWM avoids any validation for read-only transactions. For update transactions, the $\text{commit}$ function starts by validating the writes and reads as per Rule $2$. When validating a write (function $\text{handleWrite}$) 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 $\text{handleRead}$) we make the visible read (line 39). 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 $\text{source}$ 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.

<table>
<thead>
<tr>
<th>Struct</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>readStamp</td>
<td>$ts$ of $globalClock$ when this $\text{Var}$ was last read pointer to the most recent $\text{Ver}$ of this $\text{Var}$</td>
</tr>
<tr>
<td></td>
<td>latestVersion</td>
<td></td>
</tr>
<tr>
<td>Ver</td>
<td>value</td>
<td>the value of the version</td>
</tr>
<tr>
<td></td>
<td>natOrder</td>
<td>$ts$ of the natural commit order of the version</td>
</tr>
<tr>
<td></td>
<td>twOrder</td>
<td>$ts$ of the time-warp order of the version</td>
</tr>
<tr>
<td></td>
<td>nextVersion</td>
<td>pointer to the version overwritten by this one</td>
</tr>
<tr>
<td>Tx</td>
<td>writeTx</td>
<td>false when this $\text{Tx}$ is identified as read-only not used in read-only $\text{Tx}$ not used in read-only $\text{Tx}$</td>
</tr>
<tr>
<td></td>
<td>readSet</td>
<td></td>
</tr>
<tr>
<td></td>
<td>writeSet</td>
<td></td>
</tr>
<tr>
<td></td>
<td>start</td>
<td></td>
</tr>
<tr>
<td></td>
<td>source</td>
<td></td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
</tr>
<tr>
<td></td>
<td>natOrder</td>
<td>$ts$ of the natural commit order of this $\text{Tx}$</td>
</tr>
<tr>
<td></td>
<td>twOrder</td>
<td>$ts$ of the time-warp order of this $\text{Tx}$</td>
</tr>
</tbody>
</table>

Table 1. Data structures used in TWM ($ts$ stands for timestamp).
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 45). This is used to implement the time-warp commit according to Rule 1 (see line 72). At this point \( tx \) aborts only if it raised both flags (source and target in line 77 and exemplified by Fig. 2(b) with \( B \) conducting the validation). Otherwise, \( N(tx) \) is computed by atomically incrementing the global clock and reading it. The new writes are committed and stamped with both \( TW(tx) \) (as its version) and \( N(tx) \) (as the time at which it was created). Function createNewVersion places each committed write in the list of versions by using \( TW(tx) \) to establish the order. Because this order is non-strict, there may occur time-warp clashes between transactions, i.e., \( A =_{TW} B \). For a set of transactions that time-warp clash and write to the same variable \( k \), createNewVersion keeps only the update to \( k \) of the transaction \( T \) that has the least value for \( N \) (the other transactions execute line 108). In other words, the transactions in a time-warp clash are serialized in the inverse order of \( N \), 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 [26, 14, 31]). The key idea of those algorithms is the following: assume that \( T \) is the oldest active transaction, with \( S(T) = k \); 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 \( TW(U) \prec k \). For that to happen, there must exist some transaction \( Z \) concurrent with \( U \) such that: \( U \xrightarrow{rw} Z \) and \( N(Z) \prec k \). 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 [27]. This implies that a transaction 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 [26] (using, if possible, explicitly identified privatizing operations to minimize waiting time). 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 [16, 14], for which reason in the prototype implementation we have used the lock-free commit procedure of [14]. As this concern is orthogonal to the focus of this paper, we preserved a simpler presentation by using locks in our descriptions, and delegate details on the lock-free implementation for the appendix file.
Pseudo-code 1 of TWM (1/2).

1: BEGIN(Tx tx, boolean isWriteTx):
2: tx.start ← globalClock ▷ corresponds to 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: VISIBLERead(tx, var, globalClock) ▷ performed by read-only txs
11: end
12: wait until not locked(var)
13: Ver version ← var.latestVersion
14: while (version.twOrder > tx.start) ∨ ▷ rule 3 for read-only tx
15: (tx.writeTx ∧ version.natOrder > tx.start) do ▷ write tx
16: abort(tx) ▷ early abort update tx due to rule 2
17: version ← version.nextVersion
18: return version.value

19: VISIBLERead(Tx tx, Var var, long ts):
20: long lastRead ← var.readStamp
21: if lastRead < ts then
22: CAS(var.readStamp, lastRead, ts)

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 ▷ write tx
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⟩ ▷ ...as the latest version
34:▷ insert according to time-warp order...
35: if newerVersion = ⊥ then
36: var.latestVersion ← version
37: else
38: newerVersion.nextVersion ← version ▷ ...or as an older version

4. Correctness Discussion

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 [20].
4.1. Rejecting Non-Serializable Histories

To prove serializability, we first define a strict total order (\(O\)) on the transactions in the committed projection of \(H\) (noted \(H|C\)), and then we show that any edge between two transactions in DSG(\(H|C\)) is compliant with \(O\). The strict total order \(O\) is obtained from the non-strict total order defined by \(TW\), which we recall can have ties in presence of time-warp clashes, breaking ties as follows. We order update transactions in \(O\) using the time-warp order and, whenever there is a time-warp clash, i.e., \(A =_{TW} B\), we use the natural commit order \(N\) as a tie breaker and serialize \(B\) before \(A\) in \(O\) if \(A \prec_{N} B\). This results in a strict total order because \(N\) defines a strict total order as well. Any read-only transaction \(T\) is serialized in \(O\) according to \(S(T)\), which surely makes them coincide with some update transaction in \(O\). To tie-break, we place the read-only transactions always later than coinciding update transactions in \(O\). 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(\(H|C\)), we show that for any committed transactions \(A\) and \(B\) such that \(A \prec_O B\), 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 \(B \xrightarrow{wr} A\) \(\in\) DSG(\(H|C\)). According to function CREATENEWVERSION this is possible iff \(B \prec_{TW} A\). This, however, directly contradicts the initial assumption \(A \prec_O B\), because it implies that \(A \preceq_{TW} B\).

Now let us consider that \(B \xrightarrow{wr} A\). First suppose that \(A\) is an update transaction. Then, according to line 20 \(A\) can read a version created by \(B\) iff \(N(B) \not\subseteq S(A)\). However, the time-warp commit timestamp of a transaction is always less or equal than its natural commit timestamp (\(TW(B) \preceq N(B)\)); also, an update transaction \(A\) can only time-warp due to concurrent transactions, meaning they commit after \(S(A)\) and thus \(S(A) \prec TW(A)\). Hence, we obtain \(TW(B) \prec TW(A)\), contradicting the initial assumption. Now consider that \(A\) is a read-only transaction. Then, according to line 20 \(A\) can read a version created by \(B\) (concurrent with \(A\)’s execution) iff \(TW(B) \not\subseteq S(A)\). Given that \(A\) is a read-only transaction, \(TW(A) = S(A)\), hence \(TW(B) \not\subseteq TW(A)\). The case \(TW(B) \prec TW(A)\) clearly contradicts the initial assumption. If \(TW(B) = TW(A)\), then we note that \(A\) is a read-only transaction that clashes with \(B\); according to the rules we used to derive \(O\) then \(A\) is ordered after \(B\) in \(O\), which again contradicts the initial assumption (\(A \prec_O B\)).

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 \(S(B) \prec TW(A)\). But since \(B\) is read-only, then \(S(B) = TW(B)\), 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 \(B \prec_N A\). Then \(B\) performs some visible read in line 39 later \(A\) triggers the condition in line 35 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 67. Then \(TW(B) \prec (N(A) = TW(A))\), which is a contradiction with the initial assumed order. Lastly, consider the second case where \(A \prec_N B\). Then \(B\) triggers the condition in line 42. If \(A\) time-warp commits, then \(B\) aborts in line 44. Otherwise, \(B\) adds \(A\) to it’s antiDeps set which results in \(TW(B) \not\subseteq (N(A) = TW(A))\) (according to line 72). The case where \(B \prec_{TW} A\) trivially violates our initial assumption. The tie-break in the time-warp clash, where \(B \Rightarrow_{TW} A\), is broken in the inverse natural commit order (recall that \(A \prec_N B\)), which also contradicts the initial 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 [10, 20]. 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, 17] and it is guaranteed by Virtual World Consistency [20].

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(\(H\)) 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 \(\hat{S}(T)\) of \(past(T)\) where every transaction \(T' \in past(T)\) observes values written by the most recent transaction that precedes \(T'\) in \(\hat{S}(T)\).
Recall that we have argued the absence of cycles in DSG($\mathcal{H}|\mathcal{C}$). 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 $S(T)$ of past($T$) must exist. $S(T)$ is legal because transactions read the most recent version committed according to $\mathcal{T}W$ (see line [20]). But, since past($T$) respects the $\mathcal{T}W$ 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 [17], 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 [20] 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 [44] 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 [16] 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 (available in [12]), TL2 and NOrec from their respective ports to the Deuce framework (available in [24]), 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. Next we consider two more complex and realistic benchmarks, namely STMBench7 [18] and the STAMP suite of benchmarks [29]. STAMP typically contains smaller and less conflicting transactions than STMBench7, although every transaction contains writes, which is a disadvantage particularly for the multi-versioned 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 T₁ modifying an element near the end of the list need not abort only because a concurrent transaction T₂ modified an element in the beginning of the list and committed; TWM can automatically, and safely, commit T₁ before T₂, whereas classic validation would have precluded the commit of T₁.

For this micro-benchmark we used the source code available in the IntSet benchmark in the Deuce framework [24]. 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 bellow that is competitive with the other TMs. At 64 cores TWM achieves the following speedups: 2.8× for TL2; 9.4× for NOrec; 4.3× for JVSTM; and
1.8× 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, which we discuss in detail next.

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 circumstances 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 the fact 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. We begin with STMBench7 configured to enable structural mod-

![Graphs showing throughput and abort rate with various thread counts for different TM algorithms](Figure 5. STMBench7 with structural modifications.)
Figure 6. Scalability in the STAMP benchmarks.

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

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

Despite the presence of read-only transactions, the high contention makes it very difficult for any TM to scale. For the TMs using classic validation we can see that performance starts dropping at 8 threads.
TWM and AVSTM, instead, scale further to 16 threads. TWM achieves an average improvement of $1.37\times$, $1.46\times$, $1.57\times$ and $1.26\times$ with respect to TL2, NOrec, JVSTM and AVSTM. As shown in Fig. 5(b), the gains obtained by TWM are due to reduced spurious aborts, except for AVSTM, which suffers from a more costly algorithm as shown in the previous section.

We have also studied the performance of these TMs in STAMP, for which we used an existing port to Java [25]. Fig. 6 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. 6(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 merits of time-warping with an average improvement of 65% in high concurrency scenarios and gains extending up to $9\times$. 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 here is to create an efficient integration of both systems, which becomes
more challenging with the additional metadata and validations conducted in such STM algorithms [28]. We hope to provide answers to this problem in our future work.

## A Lock-Free Implementation

To simplify presentation, the TWM algorithm was described with a lock-based approach. As shown by recent works [16, 14, 26], however, in order to maximize scalability it is desirable to adopt lock-free synchronization schemes. For this reason we now present an adaptation of our algorithm to use lock-free procedures. The following adaptation does not have an impact on the main objective of time-warping, which is that of reducing spurious aborts. Therefore this should be seen as an orthogonal concern.

### 1.1. Overview of the Changes

To make TWM lock-free we leveraged on the techniques presented in [14] and adapted them to fit TWM. The main idea is that the commit of a transaction \( T \) now considers three phases: (1) validate \( T \); (2) enqueue \( T \) in \( Q \), a queue of (eventually) committed transactions; and (3) ensure that every transaction in the queue (including \( T \)) finalizes applying their write-sets. As we shall see, the queue \( Q \) is at the core of the lock-free commit procedure and, naturally, is implemented as a lock-free queue. In fact, we no longer use the \textit{globalClock} scalar, and instead use the queue to derive the \textit{natural commit order}: when a transaction \( T \) enqueues successfully, after \( P \) in the queue, \( T \) computes \( N(T) \) simply as \( N(P) + 1 \). In other words, the \textit{natural commit order} stems from the order acquired by each transaction in \( Q \).

To complement the pseudo-code description we rely on Table 3 which contains the structures used in our pseudo-code. Note that this updated table highlights (in bold) the new structures and attributes from the lock-based description in the main paper. In the following sections we shall walk through the algorithms that compose the lock-free TWM.

<table>
<thead>
<tr>
<th>Struct</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>readStamp</td>
<td>timestamp when this Var was last read</td>
</tr>
<tr>
<td>Var</td>
<td>latestVersion</td>
<td>pointer to the most recent Ver of this Var</td>
</tr>
<tr>
<td>Var</td>
<td>committingTx</td>
<td>pointer to a Tx attempting to commit to this Var</td>
</tr>
<tr>
<td>Ver</td>
<td>value</td>
<td>the value of the version</td>
</tr>
<tr>
<td>Ver</td>
<td>natOrder</td>
<td>timestamp of the \textit{natural commit order} of the version</td>
</tr>
<tr>
<td>Ver</td>
<td>twOrder</td>
<td>timestamp of the \textit{time-warp order} of the version</td>
</tr>
<tr>
<td>Ver</td>
<td>nextVersion</td>
<td>pointer to the version overwritten by this one</td>
</tr>
<tr>
<td>Tx</td>
<td>writeTx</td>
<td>false when this Tx is identified as read-only</td>
</tr>
<tr>
<td>Tx</td>
<td>readSet</td>
<td>not used in read-only Tx</td>
</tr>
<tr>
<td>Tx</td>
<td>writeSet</td>
<td>not used in read-only Tx</td>
</tr>
<tr>
<td>Tx</td>
<td>start</td>
<td>timestamp of the when this Tx started</td>
</tr>
<tr>
<td>Tx</td>
<td>target</td>
<td>true when another tx anti-depends on Tx</td>
</tr>
<tr>
<td>Tx</td>
<td>natOrder</td>
<td>true when Tx anti-depends on another tx</td>
</tr>
<tr>
<td>Tx</td>
<td>twOrder</td>
<td>timestamp of the \textit{time-warp order} of this Tx</td>
</tr>
<tr>
<td>Tx</td>
<td>isCommitted</td>
<td>true when the transaction is written-back</td>
</tr>
<tr>
<td>Tx</td>
<td>next</td>
<td>pointer to the next (more recent) Tx enqueued in ( Q ) after this Tx</td>
</tr>
<tr>
<td>Q</td>
<td>lastCommit</td>
<td>pointer to Tx that was written-back (may not be the last, due to a race)</td>
</tr>
<tr>
<td>Q</td>
<td>lastEnqueued</td>
<td>pointer to Tx that was last enqueued for commit (head of the queue)</td>
</tr>
</tbody>
</table>

Table 3. Data structures used in the lock-free TWM.
1.2. Begin, Read and Write in a Transaction

The `begin` function is slightly changed, although its core idea remains the same: to compute \( S(tx) \). However this is now derived from the latest committed transaction in \( Q \). The complexity of lines 2-7 is only due to performance concerns, and shall be more clear later in this document.

Recall that, in our main presentation of TWM, a read-only transaction could be blocked when reading a `Var` concurrently being committed by an update transaction. In the lock-free implementation we keep a pointer in each variable to its current writer (`var.committingTx`), which allows a read-only transaction to know if it is reading a variable that is being committed.

We must now consider what a read-only `R` has to do when faced with a variable undergoing commit by `T` (line 17). To preserve this procedure non-blocking, we make `R` help decide the fate of `T` — this is implemented as invoking `Commit` for `T`, which, naturally, is tailored to have concurrent threads attempting to commit the same transaction. This is the fundamental idea of transactions helping each other, which is typical in lock-free algorithms. This allows `R` to ensure progress for `T` and decide if the write of `T` is visible to the snapshot of `R` or not, and in the positive case, safely read it.

### Pseudo-code 3 - access operations

```plaintext
1: BEGIN(Tx tx, boolean isWriteTx):
2: Tx lastCommitted ← Q.lastCommit
3: while lastCommitted.next ≠ null do
4:   Tx moreRecent ← lastCommitted.next
5: if moreRecent.isCommitted then
6:   lastCommitted ← moreRecent
7: tx.start ← lastCommitted.natOrder ≡ corresponds to \( S(tx) \)
8: tx.writeTx ← isWriteTx

9: READ(Tx tx, Var var):
10: if tx.writeTx then
11: if ∃ ‹var, value› ∈ tx.writeSet then
12: return value ≡ tx had already written to var
13: tx.readSet ← tx.readSet ∪ var ≡ performed by update txs
14: else
15: VISIBLEREAD(tx, var)
16: Tx writer ← var.committingTx
17: if writer ≠ null ∧ not writer.isCommitted then
18: COMMIT(writer) ≡ help commit the writer
19: Ver version ← var.latestVersion
20: while (version.twOrder > tx.start) ∨
21:     (tx.writeTx ∧ version.natOrder > tx.start) do
22: version ← version.nextVersion
23: return version.value

24: VISIBLEREAD(Tx tx, Var var):
25: long lastRead ← var.readStamp
26: if lastRead < ts then
27: long ts ← Q.lastEnqueued.natOrder ≡ use latest timestamp
28: CAS(var.readStamp, lastRead, ts)

29: WRITE(Tx tx, Var var, Value val):
30: tx.writeSet ← (tx.writeSet \ ‹var, _› ) ∪ ‹var, val›
```
1.3. Lock-Free Commit

The main changes in the commit have to do with helping mechanisms. In fact, the key idea of the validation that powers time-warping remains unchanged in its implementation within methods handleWrite and handleRead.

Recall that the COMMIT procedure may be invoked by different threads attempting to commit the same Tx. In the pseudo-code we omit some complexities for ease of understanding — a helping thread performs most of the writes in thread-local memory, and not directly to the attributes of the Tx structure as shown in the pseudo-code.

At the start of the COMMIT the thread starts by helping commit every transaction enqueued in Q but not yet committed. As a result, it returns the most recently known committed transaction, which will be used as the expected value for the head of Q in the compare-and-swap operation in line 58. Line 52 sets \( N(tx) \) ephemerally to the value of \( N \) of the last known committed transaction — this is merely for internal usage, and will be re-computed in line 68 before the transaction commits.

This commit procedure implements two different parts: lines 53-57 validate transaction \( tx \) against the most recently known committed transaction; after that the transaction is enqueued in Q; however, if that fails, due to some concurrent transactions that got enqueued in the meantime, then an extra validation (lines 59-63) is necessary to ensure that the newly enqueued transactions do not invalidate \( tx \).

During the normal validation we can see that handleWrite has been slightly changed. Lines 31-34 implement the mechanism that replaces the lock of each \( \text{Var} \), and which was mentioned during the read operation for read-only transactions. These \( \text{var.committingTx} \) pointers are set (with CAS operations) by a committing transaction before enqueuing. To preserve lock-freedom, this step is performed over an ordered write-set. If a write transaction \( T \) attempting to commit fails to CAS itself as the writer of variable \( k \), it is because some \( T' \) is also attempting the same. In such case, \( T \) helps \( T' \) validate and enqueue in Q before proceeding with its own commit (line 34). Finally these pointers shall be cleared when the variables are written-back, to allow more transactions to commit to those variables.

The validation rule that assesses the validity of \( T \) is refactored in function UPDATETIMESTAMPs (because now it can be called multiple times) — however, its logic remains the same, as explained before. If \( T \) is deemed valid, it attempts to acquire its position at the head of Q using a CAS (line 58). This step may fail when some (possibly set of) concurrent transaction(s) enqueued instead in that position (one at a time). In such event, \( T \) has to guarantee it is still valid, by taking into account the transactions that won the race to Q. This is achieved by using the read- and write-sets of the transactions that enqueued concurrently (function INCREMENTALVALIDATION). We perform a selective validation between \( T \) and each \( C_i \in \{C_1, ..., C_N\} \) among the \( N \) transactions that won the race for the enqueue:

- \( C_i \xrightarrow{rw} T \) exists when \( T\text{.writeSet} \cap C_i\text{.readSet} \neq \emptyset \).
- \( T \xrightarrow{rw} C_i \) exists when \( T\text{.readSet} \cap C_i\text{.writeSet} \neq \emptyset \)

Note that the first check may be skipped if \( T \) was already known to be the target of a similar edge. Besides the implementation nature, we can see that the logic governing these validations is the same as presented for the normal validation.

After \( T \) is in Q, its position reveals its natural commit order, i.e., \( N(T) \). At this point, in line 64 a lock-free helping mechanism is used to commit each transaction in Q that is pending finalization.
Pseudo-code 4 - commit procedure

30: **HANDLEWRITE**(Tx tx, Var var):
   ▶ check if tx is the target of an edge
31: while not CAS(var.committingTx, null, tx) do
32:   Tx otherWriter = var.committingTx
33:   if otherWriter ≠ null then
34:     COMMIT(otherWriter)
35:   if var.readStamp ≥ tx.start then
36:     ▶ detect concurrent transactions that read var
37:     tx.target ← true
38: **HANDLEREAD**(Tx tx, Var var):
   ▶ check if tx is the source of an edge
39: **VISIBLEREAD**(tx, var)
40: ▶ check writes committed concurrently to tx’s execution
41: Ver version ← var.latestVersion
42: while version.natOrder > tx.start do
43:   abort(tx)
44:   ▶ rule 3
45:   tx.antiDeps.add(version.natOrder)
46: tx.source ← true
47: version ← version.nextVersion
48: **COMMIT**(Tx tx):
49: if !tx.writeTx then
50:   return
51:   ▶ read-only txs never abort
52: Tx lastCommit ← HELPCLIENTALL()
53: tx.natOrder = lastCommit.natOrder
54: ∀ var ∈ tx.writeSet do: HANDLEWRITE(tx, var)
55: ∀ var ∈ tx.readSet do: HANDLEREAD(tx, var)
56: ▶ not the final value for N(tx)
57: if Q contains tx then
58: return
59: while not CAS(Q.lastEnqueued, lastCommit, tx) do
60:   lastCommit ← INCREMENTALVALIDATION(tx, lastCommit)
61:   tx.natOrder ← lastCommit.natOrder
62:   UPDATETIMESTAMPS(tx)
63: if Q contains tx then
64:   return
65: HELPCLIENTALL()
66: ▶ some helper succeeded first
67: ▶ ensure tx is written-back

(which is the case for T). We delve in those details in the next section.

1.4. Helping a Transaction

The HELPCLIENTALL function simply traverses the transactions in Q not yet committed and attempts to help each one. The implementation of HELPCLIENT resembles that of [14]. Briefly, this entails applying the contents of the write-set of tx to the respective variables, and this is performed in parallel by any transaction that concurrently tries to help by splitting the write-set in buckets — note that the pseudo-code omit these details for ease of presentation. Moreover, each helper verifies that all the buckets are processed before considering T as finished. This idea of splitting the write-set in buckets is in fact shown here by iterating over the whole write-set in line 96. To create a new version we use a very similar algorithm in function CREATENEWVERSION. The only
difference now is that the version may already be in place due to concurrent helpers. Therefore the placement of the version in the list of versions uses a compare-and-swap operation (lines 113 and 116). These new versions are effectively available to new transactions after line 100 is processed by at least one helper. This moves the pointer of \( Q \) to the newly committed transaction, which can be used by a new transaction to compute its \( S \). However, note that there is a race condition in changing the \( \text{lastCommit} \) attribute of \( Q \). We avoid the use of a compare-and-swap operation here because new transactions can cope with this race condition merely by traversing the \( \text{next} \) pointers of the transactions in the queue until they reach the effective lastly committed transaction (lines 27).

1.5. Discussion

Concluding, the read and write operations of an update transaction \( T \) never repeat or block. Its commit operation may repeat when another write transaction succeeded on enqueuing in \( Q \), but that means there was global progress. \( T \) may also repeat the commit procedure if it fails to place itself as the writer of some variable \( k \), because it is already being written by some \( T' \). But in such case, \( T \) retreats by removing itself from the writer of variables, and helps \( T' \) with validation and enqueue. Because the write-sets are canonically ordered, it is impossible for a cycle of helping dependencies to exist. Therefore there will always exist some write transaction \( T \) enqueuing successfully and ensuring global progress.

With regard to read-only transactions, they never block, and do not have commit or write procedure. The read operation of a read-only transaction \( R \) helps at most the validation and enqueue of a write transaction for each read that \( R \) performs. Each helping can only fail and repeat in the presence of global progress (as explained in the last paragraph). Thus, this algorithm makes TWM lock-free.

References


Pseudo-code 5 - auxiliary commit procedures

65: \textbf{UPDATETIMESTAMPS}(Tx tx):
66: \textbf{if } tx.target \land tx.source \textbf{ then}
67: \hspace{1em} abort(tx) \hspace{1em} \text{\texttt{▷ rule 3}}
68: tx.natOrder \leftarrow tx.natOrder + 1 \hspace{1em} \text{\texttt{▷ compute } N(tx)}
69: \textbf{if } (tx.antiDeps = \emptyset) \textbf{ then}
70: tx.twOrder \leftarrow tx.natOrder \hspace{1em} \text{\texttt{▷ compute } TW(tx)}
71: \textbf{else}
72: tx.twOrder \leftarrow \min(tx.antiDeps)

73: \textbf{INCREMENTALVALIDATION}(Tx tx, Tx lastCommit):
74: \hspace{1em} \textbf{consider each new transaction}
75: \hspace{1em} \textbf{while } toCommit \neq \text{null} \textbf{ do}
76: \hspace{2em} \textbf{if not } tx.target \land
77: \hspace{3em} \text{tx.writeSet} \cap \text{lastCommit.readSet} \neq \emptyset \textbf{ then}
78: \hspace{4em} tx.target \leftarrow true
79: \hspace{2em} \textbf{if } tx.readSet \cap \text{lastCommit.writeSet} \neq \emptyset \textbf{ then}
80: \hspace{3em} \textbf{if } toCommit.source \textbf{ then}
81: \hspace{4em} abort(tx)
82: \hspace{4em} tx.source \leftarrow true
83: \hspace{3em} lastCommit \leftarrow toCommit
84: \hspace{3em} toCommit \leftarrow toCommit.next
85: \hspace{2em} return lastCommit

86: \textbf{HELPCOMMittAll}():
87: \hspace{1em} Tx lastCommit \leftarrow Q.lastCommit
88: \hspace{1em} Tx toCommit \leftarrow lastCommit.next
89: \hspace{1em} \textbf{while } toCommit \neq \text{null} \textbf{ do}
90: \hspace{2em} \textbf{HELPCOMMitt(toCommit)}
91: \hspace{2em} lastCommit \leftarrow toCommit
92: \hspace{2em} toCommit \leftarrow toCommit.next
93: \hspace{1em} return lastCommit

94: \textbf{HELPCOMMitt(Tx toCommit)}:
95: \hspace{1em} \textbf{if not } toCommit.isCommitted \textbf{ then}
96: \hspace{2em} \textbf{for all } \langle \text{var, value} \rangle \in \text{toCommit.writeSet} \textbf{ do}
97: \hspace{3em} \textbf{CREATENEWVERSION}(toCommit, \text{var, val})
98: \hspace{2em} \textbf{CAS}(\text{var.committingTx, toCommit, null})
99: \hspace{2em} toCommit.isCommitted \leftarrow true
100: \hspace{1em} Q.lastCommit \leftarrow toCommit

101: \textbf{CREATENEWVERSION}(Tx tx, Var var, Value val):
102: \hspace{1em} Ver newerVersion \leftarrow \bot
103: \hspace{1em} Ver olderVersion \leftarrow \text{var.latestVersion}
104: \hspace{1em} \textbf{while } tx.twOrder < olderVersion.twOrder \textbf{ do}
105: \hspace{2em} newerVersion \leftarrow olderVersion
106: \hspace{2em} olderVersion \leftarrow olderVersion.nextVersion
107: \hspace{2em} \textbf{if } tx.twOrder = olderVersion.twOrder \textbf{ then}
108: \hspace{3em} return
109: \hspace{2em} Ver version \leftarrow \langle \text{val, tx.natOrder, tx.twOrder, tx, olderVersion} \rangle
110: \hspace{2em} \textbf{\texttt{▷ insert according to time-warp order...}}
111: \hspace{2em} \textbf{if } newerVersion = \bot \textbf{ then}
112: \hspace{3em} \textbf{\texttt{▷ \texttt{...as the latest version}}}
113: \hspace{3em} \textbf{CAS(\text{var.latestVersion, olderVersion, version})}
114: \hspace{2em} \textbf{else}
115: \hspace{3em} \textbf{\texttt{\texttt{...or as an older version}}}
116: \hspace{3em} \textbf{CAS(newerVersion.nextVersion, olderVersion, version)