Boosting Transactional Memory with Stricter Serializability

Pierre Sutra[2], Patrick Marlier[1], Valerio Schiavoni[1], and François Trahay[2]

[1] University of Neuchâtel, Switzerland
{patrick.marlier,valerio.schiavoni}@unine.ch
[2] Télécom SudParis, France
{pierre.sutra,francois.trahay}@telecom-sudparis.eu

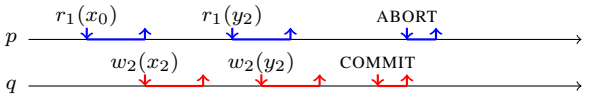
Abstract. Transactional memory (TM) guarantees that a sequence of operations encapsulated into a transaction is atomic. This simple yet powerful paradigm is a promising direction for writing concurrent applications. Recent TM designs employ a time-based mechanism to leverage the performance advantage of invisible reads. With the advent of many-core architectures and non-uniform memory (NUMA) architectures, this technique is however hitting the synchronization wall of the cache coherency protocol. To address this limitation, we propose a novel and flexible approach based on a new consistency criteria named stricter serializability (SSER$^+$). Workloads executed under SSER$^+$ are opaque when the object graph forms a tree and transactions traverse it top-down. We present a matching algorithm that supports invisible reads, lazy snapshots, and that can trade synchronization for more parallelism. Several empirical results against a well-established TM design demonstrate the benefits of our solution.

1 Introduction

The advent of chip level multiprocessing in commodity hardware has pushed applications to be more and more parallel in order to leverage the increase of computational power. However, the art of concurrent programming is known to be a difficult task [29], and programmers always look for new paradigms to simplify it. Transactional Memory (TM) is widely considered as a promising step in this direction, in particular thanks to its simplicity and programmer's friendliness [13].

The engine that orchestrates concurrent transactions run by the application, i.e., the concurrency manager, is one of the core aspects of a TM implementation. A large number of concurrency manager implementations exists, ranging from pessimistic lock-based implementations [1, 23] to completely optimistic ones [24], with [31] or without multi-version support [3]. For application workloads that exhibit a high degree of parallelism, these designs tend to favor optimistic concurrency control. In particular, a widely accepted approach consists in executing tentatively invisible read operations and validating them on the course of the transaction execution to enforce consistency. Disjoint-access parallelism (DAP) [14] ensures that concurrent transactions operating on disjoint part of the application do not contend in the concurrency manager. This property is key to ensures that the system scales with the numbers of cores.

From a developer's point of view, the interleaving of transactions must satisfy some form of correctness. Strict serializability (SSER) [26] is a consistency criteria commonly encountered in database literature. This criteria ensures that committed transactions behave as if they were executed sequentially, in an order compatible with real-time. However, SSER does not specify the conditions for aborted transactions. To illustrate this point, let us consider history h_1 where transaction $T_1 = r(x); r(y)$ and $T_2 = w(x); w(y)$ are executed respectively by processes p and q. In this history, T_1 aborts after reading inconsistent values for x and y. Yet, h_1 is compliant with SSER.



(h_1)

Opacity (OPA) was introduced [19] to avoid the side-effects of so-called *doomed transactions*, *i.e.*, transactions which eventually abort (such as T_1 in history h_1).[3] In addition to SSER, OPA requires that aborted transactions observe a prefix of the committed transactions. This is the usual consistency criteria for TM.

Achieving OPA is known to be expensive, even for weak progress properties on the transactions [32]. In particular, ensuring that a transaction always sees a consistent snapshot when read are invisible require to either validate the read set after each read operation, or to rely on a global clock. The former approach has a quadratic-time validation complexity. The latter approach is expensive in multi-core/multi-processors architecture, due to a synchronization wall.

In this paper, we address these shortcomings with a new consistency criteria, named *stricter serializability* (SSER$^+$). This criteria extends strict serializability by avoiding specifically the inconsistency depicted in history h_1. We describe in detail a corresponding TM algorithm that ensures invisible reads, and permits transactions to commit as long as they do not contend with conflicting transactions. We further validate our design by means of a full implementation of SSER$^+$ and several experiments. Our results show that when the workloads are strongly parallel, SSER$^+$ offers close-to-optimum performance.

Outline. This paper is organized as follows. Section 2 introduces our system model assumptions and defines precisely SSER$^+$. The corresponding transactional memory algorithm and a formal proof of correctness are presented in Section 3. We present the evaluation of our prototype against several benchmarks in Section 4. We survey related work in Section 5, before concluding in Section 6.

2 A new consistency criteria

This section is organized in two parts. The first part (Section 2.1) present the elements of our system model, as well as the notions of contention and bindings (Section 2.2).

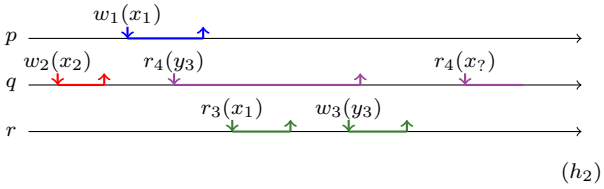
[3] Allowing T_1 to return both x_0 and y_2 may have serious consequences in a non-managed environment. As shown in [19], transaction T_1 may compute a division by 0, leading the program to crash.

In the second part (Section 2.3 and later sections), we formulate our notion of stricter serializability.

2.1 System Model

Transactional memory (TM) is a recent paradigm that allows multiple processes to access concurrently a memory region. Each process manipulates *objects* in the shared memory with the help of transactions. When a process begins a new transaction, it calls function $begin$. Then, it executes a sequence of $read$ and $write$ operations on the shared objects according to some internal (not modeled) logic. Operation $read(x)$ takes as input an object x and returns either a value in the domain of x, or a flag ABORT to indicate that the transition aborts. A write $read(x, v)$ changes x to the value v in the domain of x. This operation does not return any value and it may also abort. At the end of the execution, the process calls $tryCommit$ to terminate the transaction. This calls returns either COMMIT, to indicate that the transaction commits, or ABORT if the transaction fails.

A *history* is a sequence of invocations and responses of transactional operations by one or more processes. As illustrated below, a history is commonly represented with parallel timelines, where each timeline represents the transactions executed by some processes. In history h_2, process p, q and r execute respectively transactions $T_1 = w(x)$, $T_2 = w(x)$ then $T_4 = r(y); r(x)$, and $T_3 = r(x)$. All the transactions but T_4 completes in this history. For simplicity, complete transactions that are not explicitly aborted in a history commit immediately after their last operations. We note $com(h)$ the set of transactions that commit during history h. In the case of history h_2, we have $com(h_2) = \{T_1, T_2, T_3\}$.



(h_2)

A history induces a real-time order between transactions (denoted $\prec_h$). This order holds between two transactions T_i and T_j when T_i terminates in h before T_j begins. For instance in history h_2, transaction T_1 precedes transaction T_3. When two transactions are not related with real-time, they are *concurrent*.

A *version* is the state of a shared object as produced by the write of a transaction. This means that when transaction T_i writes to some object x, an operation denoted $w_i(x_i)$, it creates the version x_i of x. Versions allow to uniquely identify the state of an object as observed by a read operation, *e.g.*, $r_3(x_1)$ in h_2. When a transaction T_i reads version x_j, we say that T_i *read-from* transaction T_j.

Given some history h and some object x, a version order on x for h is a total order relation over the versions of x in h. By extension, a version order for h is the union of all the version orders for all objects (denoted $\ll_h$). For instance, in history h_2 above, we may consider the version order $(x_1 \ll_{h_2} x_2)$.

Consider an history h and some version order $\ll_h$. A transaction T_i *depends on* some transaction T_j, written $T_i \trianglerighteq T_k$ when T_j precedes T_i, T_i reads-from T_j, or such a relation holds transitively. Transaction T_i *anti-depends* from T_j on object x, when T_i reads a version x_k, T_j writes object x, and x_k precedes x_j in the version order ($x_k \ll_h x_j$). An anti-dependency between T_i and T_j on object x is a *reverse-commit anti-dependency* (for short, RC-anti-dependency) [22] when T_j commits before T_i and T_i writes some object $y \neq x$.

To illustrate the above definitions, consider again history h_2. In this history, transaction T_3 depends on T_1 and T_2. On the other hand, if $x_2 \ll_h x_1$ holds and T_4 reads x_2, then this transaction exhibits an anti-dependency with T_1. This anti-dependency becomes an RC-anti-dependency if T_4 executes an additional step where it writes some object z.

A transaction observes a *strictly consistent snapshot* [6] when it never misses the effects of some transaction it depends on. In other words, the snapshot of transaction T_i in history h is strictly consistent when *(i)* T_i reads version x_j, *(ii)* T_k writes version x_k, and *(iii)* T_i depends on T_k, then version x_k is followed by version x_j. Formally, $((r_i(x_j) \in h) \wedge (w_k(x_k) \in h) \wedge (T_i \trianglerighteq T_k)) \rightarrow (x_k \ll_h x_j)$, where $\ll_h$ is some version order defined over history h.

2.2 Contention and bindings

Internally, a transaction memory is built upon a set of *base objects*, such as locks or registers. When two transactions are concurrent, their steps on these base objects are interleaving. If the two transactions access disjoint objects and the TM is disjoint-access parallel, no contention occurs. However, in the case they access the base object, they may slow down each other.

A transactional read is *invisible* when it does not change the state of the base objects implementing it. With invisible reads, read contention is basically free. From a performance point of view, this property is consequently appealing, since workloads exhibit in most case a large ratio of read operations.

When two transactions are concurrently writing to some object, it is possible to detect the contention and abort preemptively one of them. On the other hand, when a read-write conflict occurs, a *race condition* occurs between the reader and the writer. If the read operation takes place after the write, the reader is bound to use the version produced by the writer.

Definition 1 (Binding). *When a transaction T_i reads-from a concurrent transaction T_j in a history h, we say that T_i is bound to T_j on x.*

When a transaction T_i is bound to another transaction T_j, to preserve the consistency of its snapshot, T_i must read the updates and causal dependencies of T_j that are intersecting with its read set. This is for instance the case of transaction T_4 in history h_2, where this transaction is bound to T_3 on y. As a consequence, T_4 must return x_1 as the result of its read on x, or its snapshot will be inconsistent.

Tracking this causality relation is difficult for the contention manager as it requires to inspect the readset, rely on a global clock, or use large amount of metadata []. We observe

that this tracking is easier if each version read prior the binding is either, accessed by the writer, or one of its dependencies. In which case, we will say that the binding is fair.

Definition 2 (Fair binding). *Consider that in some history h a transaction T_i is bound to a transaction T_j on some object x. This binding is* fair *when, for every version y_k read by T_i before x_j in h, $T_j \trianglerighteq T_k$ holds.*

Going back to history h_3, the binding of T_4 to T_3 on y is fair. Indeed, this transaction did not read any data item before accessing the version of y written by T_4. When the binding is fair, the reader can leverage the metadata left by the writer to check prior versions it has read and ensure the consistency of later read operations. In the next section, we formalize this idea with the notion of stricter serializability.

2.3 Stricter serializability

Here we introduce and describe in details SSER$^+$, the stricter serializability consistency criteria that we build upon in the remainder of this paper. The SSER criteria requires that committed transactions form a sequential history. In addition, SSER prohibits transactions to view inconsistencies unless one of their bindings is unfair. Similar to related work [7], we give a graph characterization of stricter serializability.

Definition 3 (Strict serialization graph). *Consider some version order $\ll_h$ for h. As defined below, relation $<$ captures all the relations over $com(h)$ induced by $\ll_h$. The serialization graph of history h induced by $\ll_h$, written $SSG(h, \ll_h)$, is $(com(h), <)$.*

$$
\begin{aligned}
&T_i < T_{j \neq i} \triangleq \\
&\vee\ T_i \prec_h T_j && \textit{(1)} \\
&\vee\ \exists x : \vee\ r_j(x_i) \in h && \textit{(2)} \\
&\qquad \vee\ \exists T_k : x_k \ll_h x_j \wedge \vee\ T_k = T_i && \textit{(3)} \\
&\qquad\qquad \vee\ r_i(x_k) \in h && \textit{(4)}
\end{aligned}
$$

In the above definition, (1) is a real-time order between T and T', (2) a read-write dependency, (3) a version ordering, and (4) an anti-dependency.

Definition 4 (Stricter serializability). *A history h is stricter serializable ($h \in$ SSER$^+$) when* (i) *for some version order $\ll_h$, the serialization graph $(com(h), <)$ is acyclic, and* (ii) *for every transaction T_i that aborts in h, either T_i observes a strictly consistent snapshot in h, or one of its bindings is unfair.*

Opacity (OPA) and strict serializability (SSER) coincide when no transaction aborts during an history. As a consequence of the above definition, if in a stricter serializable history all the aborted transaction exhibit fair bindings, the history is opaque.

Corollary 1. *If ($h \in$ SSER$^+$) exhibits no unfair bindings, then ($h \in$ OPA) holds.*

2.4 Applicability

Theorem 1 offers a convenient property on histories that, when it applies, allows to reach opacity. Based on this property, this section details a workload for which this property applies. In other words, we give a robustness criteria [8] against SSER$^+$.

To conduct our analysis, we focus specifically on SSER$^+$ implementations that allow invisible reads. As pointed out earlier, this restriction is motivated by performance since most workloads are read-intensive. In this context, the result of Hans et al. [22] tells us that it is not possible to jointly achieve *(i)* SSER, *(ii)* read invisibility, *(iii)* minimal progressiveness, and *(iv)* accept RC-anti-dependencies. As a consequence, we have to prune histories that exhibits such patterns from our analysis; hereafter, we shall note these histories RCAD.

A robustness criteria The state of an object commonly include a reference to another object. These references between objects form the *object graph* of the application. Processes know initially one (or more) *root* objects in this graph, and upon performing a computation traverse the object graph appropriately.

Consider that a transaction T access in order objects $x_1, \ldots, x_m$. We say that T cross a *simple path* if for all $i \in [1, m-1]$, the state of x_i includes a reference to x_{i+1}. At the light of this definition, we call P the following property on a workload.

($\mathcal{P}$) The object graph forms initially a tree and every transaction maintains this invariant. Moreover, every transaction in the workload crosses a simple path.

The proposition that follows proves that, provided $\mathcal{P}$ holds and the TM does not accept RC-anti-dependencies, then the workload is robust against SSER$^+$.

Proposition 1. *Let $\mathcal{T}$ be some set of transactions for which property P holds. Define $H_{\mathcal{T}}$ the histories built upon transactions in $\mathcal{T}$. It is true that: $(\mathit{SSER}^+ \cap H_{\mathcal{T}} \setminus \mathit{RCAD}) \subset \mathit{OPA}$.*

Proof. Choose some history $h \in H_{\mathcal{T}} \cap$ SSER$^+$, and let T_i be a transaction that aborts in h. In what follows, we prove that all of the bindings of T_i are fair in h. By definition of SSER$^+$, this leads to the fact that $h \in$ OPA.

We proceed by induction. Define x and T_j such that T_i is bound to T_j on x and assume that all the prior bindings of T_i are fair. Since $h \in$ SSER$^+$, the snapshot read by T_i is strictly consistent up to x.

Note λ the linearization of the transactions that commit in h. For some transaction T in λ, let π and π' be the paths to x before and after transaction T. We observe that if transaction T reaches x, then all the objects in π and π' are accessed by T. Indeed, to add an object π, T must read it. Conversely, to unlink a set of objects Y from π, while preserving property $\mathcal{P}$, T must make a bridge between some object y' before Y and some object y'' after it.

Let π_i be the path taken by T_i to access object x. Note $T_{i'} \in \lambda$ the transaction such that π_i is the path to x after transaction $T_{i'}$. (Since T_i observes a strictly consistent snapshot, such a transaction exists.) Name respectively π_j and ${\pi_j}'$ the path before and after T_j. According to the positions of $T_{i'}$ and T_j in λ, we may consider the following cases:

- ($T_j = T_{i'}$) Since T_j reaches x, all the objects in $\pi_i = \pi_j{}'$ are accessed by T_j; thus the binding is fair.
- ($T_j <_\lambda T_{i'}$) Name T the first transaction that modifies the path $\pi_j{}'$. Since T_i observes a strictly consistent snapshot up to x in h, T commits before T_i in h. Observe that T_j is concurrent to T_i in h, thus it is also concurrent to T. Moreover, transaction T commits before T_j, Since T modifies $\pi_j{}'$, history h exhibits and RC-anti-dependency between T and T_j. Contradiction.
- ($T_{i'} <_\lambda T_j$) Let y be an object read by T_i in the path π_i and assume that y is no more in π_j. Name $T \leq_\lambda T_j$ the transaction that removes some set Y of objects from π_i, with $y \in Y$. To preserve property $\mathcal{P}$, T updates some object y' prior to Y in the path to x. Because $h \notin$ RCAD, this removal is not concurrent to T_j. Applying by induction this reasoning, we obtain $T_j \trianglerighteq T$.

3 Algorithm

In this section, we present a transactional memory that attains SSER$^+$. Contrary to several existing TM implementation, our design does not require a global clock. It is weakly-progressive, aborting a transaction only if it encounters a concurrent conflicting transaction. Moreover, reads operations do not modify the base objects of the implementation (read invisibility).

We first give an overview of the algorithm, present its internals and justify some design choices. A correctness proofs follows. We close this section with a discussion on the parameters of our algorithm. In particular, we explain how to tailor it to be disjoint-access parallel.

3.1 Overview

Algorithm 1 depicts the pseudo-code of our construction of the TM interface at some process p. Our design follow the general approach of the lazy snapshot algorithm (LSA) of Felber et al. [16], replacing the central clock with a more flexible mechanism. Algorithm 1 employs a deferred update schema that consists in two steps. A transaction first executes optimistically, buffering its updates. Then, at commit time, the transaction is certified and, if it commits, its updates are applied to the shared memory.

During the execution of a transaction, a process checks that the objects accessed so far did not change. Similarly to LSA, this check is lazily executed. More specifically, Algorithm 1 executes it only if a shared object appears to have been recently updated, or when the transaction terminates.

3.2 Tracking Time

Algorithm 1 tracks time to compute how concurrent transactions interleave during an execution. To this end, the algorithm makes use of logical clocks. We model the interface of a *logical clock* with two operations: $read()$ returns a value in $\mathbb{N}$, and $adv(v \in \mathbb{N})$ updates the clock with value v. The sequential specification of a logical clock guarantees a single property, that the time flows forward: *(Time Monotonicity)* A read operation

always returns at least the greatest value to which the clock advanced so far. Formally, for every history h, $(res(read(), v) \in h) \rightarrow (v \geq \max(\{u : adv(u) \prec_h read()\} \cup \{0\}))$.

Algorithm 1 associates logical clocks with both processes and transactions. To retrieve the clock associated with some object i, the algorithm uses function $clock(i)$. Notice that in the pseudo-code, when it is clear from the context, we write $clock(i)$ as a shorthand for $clock(i).read()$.

The clock associated with a transaction is always local (line 2). In the case of a process, it might be shared or not (line 3). The flexibility of our design comes from this locality choice for $clock(p)$. When the clock is shared, it is linearizable. To implement an (obstruction-free) linearizable clock we employ the following common approach:

(**Construction 1**) Let x be a shared register initialized to 0. When $read()$ is called, we return the value stored in x. Upon executing $adv(v)$, we fetch the value stored in x, say u. If $v > u$ holds, we execute a compare-and-swap to replace u with v; otherwise the operation returns. If the compare-and-swap fails, the previous steps are retried.

3.3 Details

In Algorithm 1, each object x has a *location* in the shared memory, denoted $loc(x)$. This location stores a pair (t, d), where $t \in \mathbb{N}$ is a *timestamp*, and d is the actual content of x as seen by transactions. For simplicity, we shall name hereafter a pair (t, d) a *version* of object x. Since the location of object x is unique, a single version of object x may exist at a time in the memory. As usual, we assume some transaction T_{INIT} that initializes for every object x the location $loc(x)$ to $(0, \perp)$. Furthermore, we consider that each read or write operation to some location $loc(x)$ is atomic.

Algorithm 1 associates a lock to each object. To manipulate the lock-related functions of object x, a process p employs appropriately the functions $lock(x)$, $isLocked(x)$ and $unlock(x)$.

For every transaction T submitted to the system, Algorithm 1 maintains three local data structures: $clock(T)$ is the logical clock of transaction T; $rs(T)$ is a map that contains its read set; and $ws(T)$ is another map that stores the write set of T. Algorithm 1 updates incrementally $rs(T)$ and $ws(T)$ over the course of the execution. The read set serves to check that snapshot of the shared memory as seen by the transaction is strictly consistent. The write set buffers updates. With more details, the execution of a transaction T proceeds as follows.

- When T starts its execution, Algorithm 1 initializes $clock(T)$ to the smallest value of $clock(q)$ for any process q executing the TM. Then, both $rs(T)$ and $ws(T)$ are set to $\varnothing$.
- When T accesses a shared object x, if x was previously written, its value is returned (line 10). Otherwise, Algorithm 1 fetches atomically the version (d, t), as seen in location $loc(x)$. Then, the algorithm checks that *(i)* no lock is held on x, and *(ii)* in case x was previously accessed, that T observes the same version. If one of these two conditions fails, Algorithm 1 aborts transaction T (line 14). The algorithm then checks that the timestamp t associated to the content d is smaller than the clock of

Algorithm 1 A SSER$^+$ transactional memory – code at process p

```
1: Variables:
2:     clock(T), rs(T), ws(T)                                   // local to p
3:     clock(p)                                                 // shared

4: begin(T)
5:     clock(T).adv(min_q clock(q))
6:     rs(T) ← ∅
7:     ws(T) ← ∅

8: read(T, x)
9:     if (d, x) ∈ ws(T) then
10:        return d
11:    (d, t) ← loc(x)
12:    if isLocked(x)
13:        ∨ (∃(t', x) ∈ rs(T) : t ≠ t') then
14:        return abort(t)
15:    if t > clock(T) ∧ ¬ extend(T, t) then
16:        return abort(t)
17:    rs(T)[x] ← t
18:    return d

19: write(T, x, d)
20:    if ¬ lock(x) then
21:        return abort(T)
22:    (_, t) ← loc(x)
23:    if t > clock(T) ∧ ¬ extend(T, t) then
24:        return abort(t)
25:    ws(T)[x] ← d

26: tryCommit(T)
27:    if ¬ extend(T, clock(T)) then
28:        return abort(T)
29:    return commit(T)

// Helpers
30: extend(T, t)
31:    for all (t', x) ∈ rs(T) do
32:        (_, t'') ← loc(x)
33:        if isLocked(x) ∨ t'' ≠ t' then
34:            return false
35:    clock(T).adv(t)
36:    return true

37: abort(T)
38:    for all (_, x) ∈ ws(T) do
39:        unlock(x)
40:    return ABORT

41: commit(T)
42:    if ws(T) ≠ ∅ then
43:        clock(T).adv(clock(T) + 1)
44:        clock(p).adv(clock(T))
45:        for all (d, x) ∈ ws(T) do
46:            loc(x) ← (d, clock(T))
47:            unlock(x)
48:    return COMMIT
```

T. In case this does not hold (line 15), Algorithm 1 tries extending the snapshot of T by calling function $extend()$. This function returns $true$ when the versions previously read by T are still valid. In which case, $clock(T)$ is updated to the value t.

If Algorithm 1 succeeds in extending (if needed) the snapshot of T, d is returned and the read set of T updated accordingly; otherwise transaction T is aborted (line 16).
- Upon executing a write request on behalf of T to some object x, Algorithm 1 takes the lock associated with x (line 20), and in case of success, it buffers the update value d in $ws(T)$ (line 25). The timestamp t of x at the time Algorithm 1 takes the lock serves two purposes. First, Algorithm 1 checks that t is lower than the current clock of T, and if not T is extended (line 23). Second, it is saved in $ws(T)$ to ensure that at commit time the timestamp of the version of x written by T is greater than t.
- When T requests to commit, Algorithm 1 certifies the read set by calling function $extend()$ with the clock of T (line 27). If this test succeeds, transaction T commits (lines 43 to 48). In such a case, $clock(T)$ ticks to reach its final value (line 43). By construction, this value is greater than the timestamps of all the versions read or written by T (lines 14 and 23). Algorithm 1 updates the clock of p with the final value of $clock(T)$ (line 44), then it updates the items written by T with their novel versions (line 46).

3.4 Guarantees

In this section, we assess the core properties of Algorithm 1. First, we show that our TM design is weakly progressive, i.e., that the algorithm aborts a transaction only if it encounters a concurrent conflicting transaction. Then, we prove that Algorithm 1 is stricter serializable.

(Weak-progress) A transaction executes under *weak progressiveness* [20], or equivalently it is *weakly progressive*, when it aborts only if it encounters a conflicting transaction. By extension, a TM is weakly progressive when it only produces histories during which transactions are weakly-progressive. We prove that this property holds for Algorithm 1.

In Algorithm 1, a transaction T aborts either at line 14, 16, 21, 24, or 28. We observe that in such a case either T observes an item x locked, or that the timestamp associated with x has changed. It follows that if T aborts then it observes a conflict with a concurrent transaction. From which we deduce that it is executing under weak progressiveness.

(Stricter serializability) Consider some run ρ of Algorithm 1, and let h be the history produced in ρ. At the light of its pseudo-code, every function defined in Algorithm 1 is wait-free. As a consequence, we may consider without lack of generality that h is complete, i.e., every transaction executed in h terminates with either a commit or an abort event. In what follows, we define that $\ll_h$ as the order in which writes to the object locations are linearized in ρ. We first prove that $<$ is acyclic for this definition of $\ll_h$. Then, we show that, if a transaction does not exhibit any unfair binding, then it observes a strictly consistent snapshot. For some transaction, we shall note $clock(T_i)_f$ the final value of $clock(T)$.

Proposition 2. *Consider two transactions T_i and $T_{j\neq i}$ in h. If either $T_i \trianglerighteq T_j$ or $x_j \ll_h x_i$ holds, then $clock(T_i)_f \geq clock(T_j)_f$ is true. In addition, if transaction T_i commits then the ordering is strict, i.e., $clock(T_i)_f > clock(T_j)_f$.*

Proof. In each of the two cases, we prove that $clock(T_i)_f \geq clock(T_i)_f$ holds before transaction T_i commits.

($T_i \trianglerighteq T_j$) Let x be an object such that $r_i(x_j)$ occurs in h. Since transaction T_i reads version x_j, transaction T_j commits. We observe that T_j writes version x_j together with $clock(T_j)_f$ at $loc(x)$ when it commits (line 46). As a consequence, when transaction T_i returns version x_i at line 18, it assigns $clock(T_j)_f$ to t before at line 11. The condition at line 15 implies that either $clock(T_i) \geq t$ holds, or a call to $extend(T_i, t)$ occurs. In the latter case, transaction T_i executes line 35, advancing its clock up to the value of t.

($x_j \ll_h x_i$) By definition, relation $\ll_h$ forms a total order over all versions of x. Thus, we may reason by induction, considering that x_i is immediately after x_j in the order $\ll_h$. When T_j returns from $w_j(x_j)$ at line 25, it holds a lock on x. This lock is released at line 47 after writing to $loc(x)$. As $\ll_h$ follows the linearization order, T_i executes line 20 after T_j wrote $(x_j, clock(T_j)_f)$ to $loc(x)$. Location $loc(x)$ is not updated between x_j and x_i. Hence, afterT_i executes line 23, $clock(T_i) \geq clock(T_j)$ holds.

Since a clock is monotonic, the relation holds forever. Then, if transaction T_i commits, it must executes line 43, leading to $clock(T_i)_f > clock(T_i)_f$.

Proposition 3. *History h does not exhibit any RC-anti-dependencies ($h \notin RCAD$)*

Proof. Consider T_i, T_j and T_k such that $r_i(x_k), w_j(x_j) \in h$, $x_k \ll_h x_j$ and T_j commits before T_i. When T_j invokes $commit$, it holds a lock on x. This lock is released at line 47 after version x_j is written at location $loc(x)$. Then, consider the time at which T_i invokes $tryCommit$. The call at line 27 leads to fetching $loc(x)$ at line 32. Since T_i reads version x_k in h, a pair $(x_k, clock(T_k)_f)$ is in $rs(T_i)$. From the definition of $\ll_h$ the write of $(x_k, clock(T_k)_f)$ takes place before the write of version $(x_j, clock(T_j)_f)$ in ρ. Hence, $loc(x)$ does not contain anymore $(x_k, clock(T_k)_f)$ Applying Proposition 2, T_i executes line 34 and aborts at line 29.

Proposition 4. *Consider two transactions T_i and $T_{j \neq i}$ in $com(h)$. If $T_i < T_j$ holds, transaction T_i invokes $commit$ before transaction T_j in h.*

Proof. Assume that T_i and T_j conflict of some object x. We examine in order each of the four cases defining relation $<$.

- ($T_i \prec_h T_j$)
  This case is immediate.
- ($\exists x : r_j(x_i) \in h$)
  Before committing, T_j invokes $extend$ at line 27. Since T_j commits in h, it should retrieve $(x_i, \_)$ from $loc(x)$ when executing line 32. Hence, transaction T_i has already executed line 46 on object x. It follows that T_i invokes $commit$ before transaction T_j in history h.
- ($\exists x : x_i \ll_h x_j$)
  By definition of $\ll_h$, the write of version x_i is linearized before the write of version x_j in ρ. After T_i returns from $w_i(x_i)$, it owns a lock on object x (line 46). The object is then unlocked by transaction T_i at line 47. As a consequence, transaction T_i takes

a lock on object x after T_i invokes operation $commit$. From which it follows that the claim holds.

– $(\exists x, T_k : x_k \ll_x x_j \wedge r_i(x_k))$
Follows from Proposition 3.

Theorem 1. *History h belongs to SSER$^+$.*

Proof. Proposition 4 tells us that if $T_i < T_j$ holds then T_i commits before T_j. It follows that the $SSG(h, \ll_h)$ is acyclic.

Let us now turn our attention to the second property of SSER$^+$. Assume that a transaction T_i aborts in h. For the sake of contradiction, consider that T_i exhibits fair bindings and yet that it observes a non-strictly consistent snapshot.

Applying the definition given in Section 2.1, there exist transactions T_j and T_k such that $T_i \trianglerighteq T_j$, $r_i(x_k)$ occurs in h and $x_k \ll_h x_j$. Applying Proposition 4, if $T_j \prec_h T_i$ holds, transaction T_i cannot observe version x_k. Thus, transaction T_j is concurrent to T_i. Moreover, by definition of $T_i \trianglerighteq T_j$, there exist a transaction T_l (possibly, T_j) and some object y such that T_i performs $r_i(y_l)$ and $T_l \trianglerighteq T_j$. In what follows, we prove that T_i aborts before returning y_l.

For starter, relation $<$ is acyclic, thus $x_k \neq y_l$ holds. It then remains to investigate the following two cases:

– $(r_i(y_l) \prec_h r_i(x_k))$
From Proposition 4 and $T_l \trianglerighteq T_j$, transaction T_j is committed at the time T_i reads object x. Contradiction.

– $(r_i(x_k) \prec_h r_i(y_l))$
We first argue that, at the time T_i executes line 11, the timestamp fetches from $loc(y)$ is greater than $clock(T_i)$.

Proof. First of all, observe that T_j is not committed at the time T_i reads object x (since $x_k \ll_h x_j$ holds). Hence, denoting q the process that executes T_j, $clock(q) < clock(T_j)_f$ is true when T_i begins its execution at line 5. From the pseudo-code at line 5, $clock(T_i) < clock(T_j)_f$ holds at the start of T_i. Because T_j is concurrent to T_i, T_l is also concurrent to T_i by Proposition 4. Thus, as $r_i(y_l)$ occurs, T_i is bound to T_l on y. Now, consider some object z read by T_i before y, and name z_r the version read by T_i. Since the binding of T_i to T_l is fair, $T_l \trianglerighteq T_r$ is true. Hence, applying Proposition 2, we have $clock(T_r)_f < clock(T_l)_f$. It follows that the relation $clock(T_i) < clock(T_l)_f$ is true.

From what precedes, transaction T_i invokes $extend$ at line 15. We know that transaction T_j is committed at that time (since T_l is committed and $T_l \trianglerighteq T_j$ holds). Thus, the test at line 33 fails and T_i aborts before returning y_l.

3.5 Discussion

Algorithm 1 replaces the global clock usually employed in TM architectures with a more flexible mechanism. For some process p, $clock(p)$ can be local to p, shared across a subset of the processes, or even all of them.

If processes need to synchronize too often, maintaining consistency among the various clocks is expensive. In this situation, it might be of interest to find a compromise

between the cost of cache coherency and the need for synchronization. For instance, in a NUMA architecture, Algorithm 1 may assign a clock per hardware socket. Upon a call to $clock(p)$, the algorithm returns the clock defined for the socket in which the processor executing process p resides.

On the other hand, when the processes use a global clock, Algorithm 1 boils down to the original TinySTM algorithm of Felber et al. [16]. In this case, a read-only transaction always sees a strictly consistent snapshot. As a consequence, it can commit right after a call to $tryCommit$, i.e., without checking its snapshot at line 38.

A last observation is that our algorithm works even if one of the processes takes no step. This implies that the calls to process clocks (at lines 5 and 44) are strictly speaking necessary and can be skipped without impacting the correctness of Algorithm 1. Clocks are solely used to avoid extending the snapshot at each step where a larger timestamp is encountered. If process clocks are not used, when two transactions access disjoint objects, they do not contend on any base object of the implementation. As a consequence, such a variation of Algorithm 1 is disjoint-access parallel.

4 Evaluation

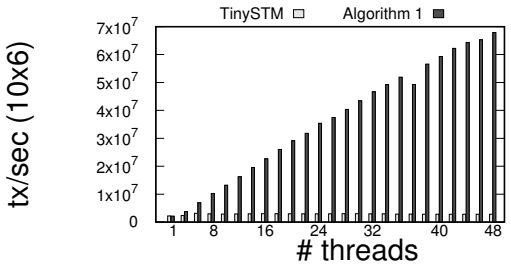
This section presents a performance study of our SSER$^+$ transactional memory depicted in Section 3. To conduct this evaluation we implemented and integrated our algorithm inside TinySTM [16]. Our modifications account for approximatively 500 SLOC. We run Algorithm 1 in disjoint-access parallel (DAP) mode. As explained in Section 3.5, in this variation the clocks of the processes are not accessed. A detailed evaluation of the other variations of Algorithm 1 is subject to future work.

The experiments are conducted on an AMD Opteron48, a 48-cores machine with 256 GB of RAM. This machine has 4 dodeca-core AMD Opteron 6172, and 8 NUMA nodes. To evaluate the performance of our implementation on this multi-core platform, we use the test suite included with TinySTM. This test suite is composed of several TM applications with different transaction patterns. The reminder of this section briefly describes the benchmarks and discuss our results. As a matter of a comparison, we also present the results achieved with the default TinySTM distribution, (v1.0.5).

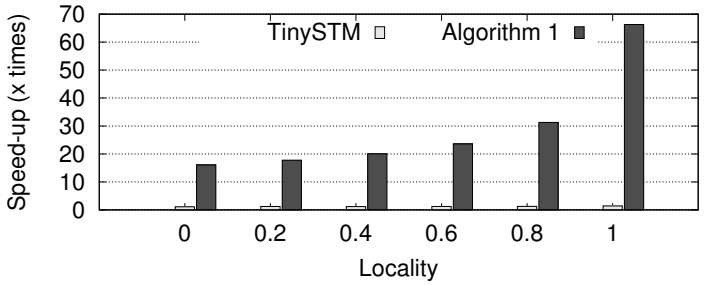
4.1 A bank application

The bank benchmark consists in simulating transfers between bank accounts. A transaction updates two accounts, transferring some random amount of money from one account to another. A thread executing this benchmark perform transfers in closed-loop. Each thread is bound to some branch of the bank, and accounts are spread evenly across the branches. A *locality* parameter allows to tune the accounts accessed by a thread to do a transfer. This parameter serves to adjust the probability that a thread executes consecutive operations on the same data. More specifically, when locality is set to the value ρ, a thread executes a transfer in its branch with probability ρ and between two random accounts with probability $(1 - \rho)$. When $\rho = 1$, the workload is fully parallel.

Figure 1 presents our result for the bank benchmark. In Figure 1(a), we execute a base scenario with 10k bank accounts, and a locality of 0.8. We measure the number of



(a) Base scenario – 2^{10} accounts, locality set to 0.8



(b) Varying locality – using 48 threads

Fig. 1. Bank benchmark

transfers performed by varying the number of threads in the application. In this figure, we observe that the performance obtained with TINYSTM merely improves as the number of thread increases: 48 threads achieve 2.8 million transactions per second (MTPS), scaling-up from 2.2 MTPS with a single thread. Our implementation performs better: with 48 threads Algorithm 1 executes around 68 MTPS, executing $\times 31$ more operations than with one thread.

To understand the impact of data locality on performance, we vary this parameter for a fixed number of threads. Figure 1(b) presents the speedup obtained when varying locality from 0, i.e., all the accounts are chosen at random, up to 1, where they are all chosen in the local branch. In this experiment, we fix the number of threads to 48, *i.e.* the maximum number of cores available on our test machine. As shown in Figure 1(b), our TM implementation leverages the presence of data locality. This is expected, since we use the disjoint-access parallel (DAP) variation of Algorithm 1. When locality increases, the contention in the application decreases. As a consequence of DAP, each thread works on independent data, hence improving performance.

4.2 Linked-list

The linked-list benchmark consists in concurrently modifying a sorted linked-list of integers. Each thread randomly adds or removes an integer from the list. We run this

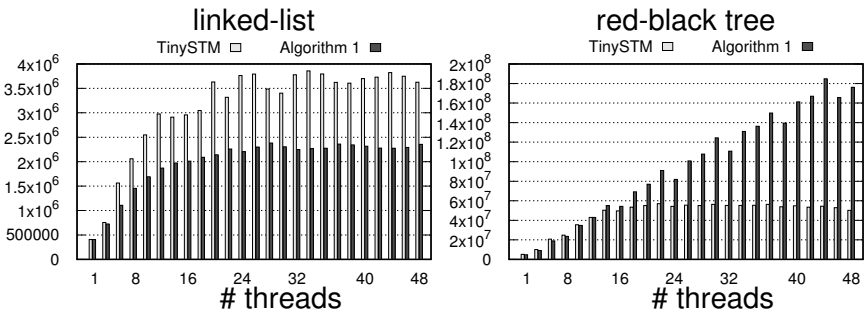


Fig. 2. Linked-list (left) and Red-Black tree (right) benchmarks. Y-axis: transactions/sec.

benchmark for a range of 512 values, *i.e.* a thread randomly selects a value between -255 and $+256$ before doing an insertion/removal. The linked list is initialized to contains 256 integers. We report our results in Figure 2 (left).

In Figure 2, we observe that TINYSTM outperforms our implementation in the linked-list benchmark. This is due to the fact that without proper clock synchronization, transaction tends to re-validate their reads frequently over their execution paths. In this scenario of high contention, it is (as expected) preferable to rely on a frequent synchronization mechanism such as the global clock used in TINYSTM. To alleviate this issue, we plan in a future work to adjust dynamically the clocks used in Algorithm 1 accordingly to contention. Such a strategy can be put at work with the help of a global lock, similarly to the mechanism used to avoid that long transactions abort.

4.3 Reb-black tree

The red-black tree benchmark is similar to the linked-list benchmark except that the values are stored in a self-balancing binary search tree. We run this benchmark with a range of 10^7 values, and a binary tree initialized with 10^5 values. Figure 2 (right) reports our results.

When using the global clock, the performance of this application improves linearly up to 12 threads. It then stalls to approximately 50 millions transactions per second due to contention on the global clock. In this benchmark, the likelihood of a concurrent transaction is very low because of the high number of values in the tree. Leveraging this workload property, our implementation of Algorithm 1, scales this application linearly with the number of threads. Algorithm 1 achieves 176 MTPS with 48 threads, improving performance by a $\times 36$ factor over a single threaded execution.

5 Related Work

Transactional memory (TM) allows to design applications with the help of sequences of instructions that run in isolation one from another. This paradigm greatly simplifies the programming of modern highly-parallel computer architectures.

Ennals [15] suggests to build deadlock-free lock-based TMs rather than non-blocking ones. Empirical evidences [10] and theoretical results [18, 28] support this claim.

At first glance, it might be of interest that a TM design accepts all correct histories; a property named *permissiveness* [21]. Such TM algorithms need to track large dependencies [27] and/or acquire locks for read operations [2]. However, both techniques are known to have a significant impact on performance.

Early TM implementations (such as DSTM [25]) validate all the prior reads when accessing a new object. The complexity of this approach is quadratic in the number of objects read along the execution path. A time-based TM avoids this effort by relying on the use a global clock to timestamp object versions. Zhang et al. [35] compare several such approaches, namely TL2 [11], LSA [33] and GCC[34]. They provide guidelines to reduce unnecessary validations and shorten the commit sequence.

Multi-versioning [12, 17] brings a major benefit: allowing read-only transactions to complete. This clearly boosts certain workloads but managing multiple versions has a non-negligible performance cost on the TM internals. Similarly, invisible reads ensure that read operations do not contend in most cases. However, such a technique limits progress or the consistency criteria satisfied by the TM [4]. In the case of Algorithm 1, both read-only and updates transaction are certain to make progress only in the absence of contention.

New challenges arise when considering multicore architectures and cache coherency strategies for NUMA architectures. Clock contention [9] is one of them. To avoid this problem, workloads as well as TM designs should take into account parallelism [30]. Chan and Wang [9] propose to group threads into zones, and that each zone shares a clock and a clock table. To timestamp a new version, the TL2C algorithm of Avni and Shavit [5] tags it with a local counter together with the thread id. Each thread stores a vector of the latest timestamp it encountered. The algorithm preserves opacity by requiring that a transaction restarts if one of the vector entries is not up to date.

6 Conclusion

Transactional memory systems must handle a tradeoff between consistency and performance. It is impractical to take into account all possible combinations of read and write conflicts, as it would lead to largely inefficient solutions. For instance, accepting RCAD histories brings only a small performance benefits in the general case [22].

This paper introduces a new consistency criteria, named stricter serializability (SSER^+). Workloads executed under SSER^+ are opaque when the object graph forms a tree and transactions traverse it top-down. We present an algorithm to attain this criteria together with a proof of its correctness. Our evaluation based on a fully implemented prototype demonstrates that such an approach is very efficient in weakly-contended workloads.

Bibliography

[1] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *International Symposium on Distributed Computing*, pages 297–311. Springer, 2012.

[2] H. Attiya and E. Hillel. A single-version stm that is multi-versioned permissive. *Theory of Computing Systems*, 51(4):425–446, Nov 2012. ISSN 1433-0490. doi: 10.1007/s00224-012-9406-3.

[3] H. Attiya and E. Hillel. A single-version stm that is multi-versioned permissive. *Theory of Computing Systems*, 51(4):425–446, 2012.

[4] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 69–78, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. doi: 10.1145/1583991.1584015.

[5] H. Avni and N. Shavit. Maintaining consistent transactional states without a global clock. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity*, SIROCCO '08, pages 131–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proc. of the 6th International Conference on Very Large Data Bases*, pages 285–300, Oct. 1980.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] A. Cerone and A. Gotsman. Analysing snapshot isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 55–64, 2016. doi: 10.1145/2933057.2933096.

[9] K. Chan and C.-L. Wang. Trc-mc: Decentralized software transactional memory for multi-multicore computers. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 292–299, 2011.

[10] D. Dice and N. Shavit. What really makes transactions fast?, 2006.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[12] N. Diegues and P. Romano. Time-warp: Lightweight abort minimization in transactional memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 167–178, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555259.

[13] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM Can Be More Than a Research Toy. *Commun. ACM*, 54(4):70–77, Apr. 2011.

[14] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 115–124. ACM, 2012.

[15] R. Ennals. Software transactional memory should not be obstruction-free, 2005.

[16] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, 21(12):1793–1807, 2010.

[17] S. M. Fernandes and J. a. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 179–188, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941579.

[18] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[19] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[20] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 404–415, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480931.

[21] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In G. Taubenfeld, editor, *Distributed Computing*, pages 305–319, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87779-0.

[22] S. Hans, A. Hassan, R. Palmieri, S. Peluso, and B. Ravindran. Opacity vs tms2: Expectations and reality. In C. Gavoille and D. Ilcinkas, editors, *Distributed Computing*, pages 269–283, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53426-7.

[23] T. Harris and K. Fraser. Revocable locks for non-blocking programming. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 72–82. ACM, 2005.

[24] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *ACM SIGPLAN Notices*, volume 49, pages 387–388. ACM, 2014.

[25] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[26] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[27] I. Keidar and D. Perelman. On avoiding spare aborts in transactional memory. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 59–68, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. doi: 10.1145/1583991.1584013.

[28] P. Kuznetsov and S. Ravi. Why transactional memory should not be obstruction-free. *CoRR*, abs/1502.02725, 2015.

[29] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[30] D. Nguyen and K. Pingali. What scalable programs need from transactional memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 105–118, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037750.

[31] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: selective multi-versioning STM. In *Distributed Computing*, pages 125–140. Springer, 2011.

[32] S. Ravi. Lower bounds for transactional memory. *Bulletin of the EATCS*, 121, 2017.

[33] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Distributed Computing*, pages 284–298. Springer, 2006.

[34] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Distributed Computing*, pages 179–193. Springer, 2006.

[35] R. Zhang, Z. Budimlić, and W. N. Scherer III. Commit phase in timestamp-based STM. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*, pages 326–335. ACM, 2008.