Progressive Transactional Memory in Time and Space

Petr Kuznetsov[1] and Srivatsan Ravi[2](✉)

[1] Télécom ParisTech, Paris, France
petr.kuznetsov@telecom-paristech.fr
[2] TU Berlin, Berlin, Germany
srivatsan.ravi@inet.tu-berlin.de

Abstract. Transactional memory (TM) allows concurrent processes to organize sequences of operations on shared *data items* into atomic transactions. A transaction may commit, in which case it appears to have executed sequentially or it may *abort*, in which case no data item is updated.

The TM programming paradigm emerged as an alternative to conventional fine-grained locking techniques, offering ease of programming and compositionality. Though typically themselves implemented using locks, TMs hide the inherent issues of lock-based synchronization behind a nice transactional programming interface.

In this paper, we explore inherent time and space complexity of lock-based TMs, with a focus of the most popular class of *progressive* lock-based TMs. We derive that a progressive TM might enforce a read-only transaction to perform a quadratic (in the number of the data items it reads) number of steps and access a linear number of distinct memory locations, closing the question of inherent cost of *read validation* in TMs. We then show that the total number of *remote memory references* (RMRs) that take place in an execution of a progressive TM in which n concurrent processes perform transactions on a single data item might reach $\Omega(n \log n)$, which appears to be the first RMR complexity lower bound for transactional memory.

Keywords: Transactional memory · Mutual exclusion · Step complexity

1 Introduction

Transactional memory (TM) allows concurrent processes to organize sequences of operations on shared *data items* into atomic transactions. A transaction may *commit*, in which case it appears to have executed sequentially or it may *abort*, in which case no data item is updated. The user can therefore design software having only sequential semantics in mind and let the TM take care of handling

Petr Kuznetsov—The author is supported by the Agence Nationale de la Recherche, ANR-14-CE35-0010-01, project DISCMAT.

© Springer International Publishing Switzerland 2015
V. Malyshkin (Ed.): PaCT 2015, LNCS 9251, pp. 410–425, 2015.
DOI: 10.1007/978-3-319-21909-7_40

conflicts (concurrent reading and writing to the same data item) resulting from concurrent executions. Another benefit of transactional memory over conventional lock-based concurrent programming is *compositionality*: it allows the programmer to easily compose multiple operations on multiple objects into atomic units, which is very hard to achieve using locks directly. Therefore, while still typically *implemented* using locks, TMs hide the inherent issues of lock-based programming behind an easy-to-use and compositional transactional interface.

At a high level, a TM implementation must ensure that transactions are *consistent* with some sequential execution. A natural consistency criterion is *strict serializability* [19]: all committed transactions appear to execute sequentially in some total order respecting the timing of non-overlapping transactions. The stronger criterion of *opacity* [13], guarantees that *every* transaction (including aborted and incomplete ones) observes a view that is consistent with the *same* sequential execution, which implies that no transaction would expose a pathological behavior, not predicted by the sequential program, such as division-by-zero or infinite loop.

Notice that a TM implementation in which every transaction is aborted is trivially opaque, but not very useful. Hence, the TM must satisfy some *progress* guarantee specifying the conditions under which a transaction is allowed to abort. It is typically expected that a transaction aborts only because of *data conflicts* with a concurrent one, e.g., when they are both trying to access the same data item and at least one of the transactions is trying to update it. This progress guarantee, captured formally by the criterion of *progressiveness* [12], is satisfied by most TM implementations today [6,7,14].

There are two design principles which state-of-the-art TM [6–8,11,14,21] implementations adhere to: *read invisibility* [4,9] and *disjoint-access parallelism* [5,16]. Both are assumed to decrease the chances of a transaction to encounter a data conflict and, thus, improve performance of progressive TMs. Intuitively, reads performed by a TM are invisible if they do not modify the shared memory used by the TM implementation and, thus, do not affect other transactions. A disjoint-access parallel (DAP) TM ensures that transaction accessing disjoint data sets do not contend on the shared memory and, thus, may proceed independently. As was earlier observed [13], the combination of these principles incurs some inherent costs, and the main motivation of this paper is to explore these costs.

Intuitively, the overhead invisible read may incur comes from the need of *validation*, *i.e.*, ensuring that read data items have not been updated when the transaction completes. Our first result (Sect. 4) is that a read-only transaction in an opaque TM featured with *weak* DAP and *weak* invisible reads must *incrementally* validate every next read operation. This results in a quadratic (in the size of the transaction's read set) step-complexity lower bound. Informally, weak DAP means that two transactions encounter a memory race only if their data sets are connected in the *conflict graph*, capturing data-set overlaps among all concurrent transactions. Weak read invisibility allows read operations of a transaction T to be "visible" only if T is concurrent with another transaction. The lower bound is derived for *minimal* progressiveness, where transactions are guaranteed to commit

only if they run sequentially. Our result improves the lower bound [12,13] derived for *strict-data partitioning* (a very strong version of DAP) and (strong) invisible reads.

Our second result is that, under weak DAP and weak read invisibility, a strictly serializable TM must have a read-only transaction that accesses a linear (in the size of the transaction's read set) number of distinct memory locations in the course of performing its last read operation. Naturally, this space lower bound also applies to opaque TMs.

We then turn our focus to *strongly progressive* TMs [13] that, in addition to progressiveness, ensures that *not all* concurrent transactions conflicting over a single data item abort. In Sect. 5, we prove that in any strongly progressive strictly serializable TM implementation that accesses the shared memory with *read*, *write* and *conditional* primitives, such as *compare-and-swap* and *load-linked/store-conditional*, the total number of *remote memory references* (RMRs) that take place in an execution of a progressive TM in which n concurrent processes perform transactions on a single data item might reach $\Omega(n \log n)$. The result is obtained via a reduction to an analogous lower bound for mutual exclusion [3]. In the reduction, we show that any TM with the above properties can be used to implement a *deadlock-free* mutual exclusion, employing transactional operations on only one data item and incurring a constant RMR overhead. The lower bound applies to RMRs in both the *cache-coherent (CC)* and *distributed shared memory (DSM)* models, and it appears to be the first RMR complexity lower bound for transactional memory.

2 Model

TM Interface. A *transactional memory* (in short, *TM*) supports *transactions* for reading and writing on a finite set of data items, referred to as *t-objects*. Every transaction T_k has a unique identifier k. We assume no bound on the size of a t-object, *i.e.*, the cardinality on the set V of possible different values a t-object can have. A transaction T_k may contain the following *t-operations*, each being a matching pair of an *invocation* and a *response*: $read_k(X)$ returns a value in some domain V (denoted $read_k(X) \rightarrow v$) or a special value $A_k \notin V$ (*abort*); $write_k(X, v)$, for a value $v \in V$, returns *ok* or A_k; $tryC_k$ returns $C_k \notin V$ (*commit*) or A_k.

Implementations. We assume an asynchronous shared-memory system in which a set of $n > 1$ processes $p_1, \ldots, p_n$ communicate by applying *operations* on shared *objects*. An object is an instance of an *abstract data type* which specifies a set of operations that provide the only means to manipulate the object. An *implementation* of an object type τ provides a specific data-representation of τ by applying *primitives* on shared *base objects*, each of which is assigned an initial value and a set of algorithms $I_1(\tau), \ldots, I_n(\tau)$, one for each process. We assume that these primitives are *deterministic*. Specifically, a TM *implementation* provides processes with algorithms for implementing $read_k$, $write_k$ and $tryC_k()$ of a transaction T_k by *applying primitives* from a set of shared *base objects*.

We assume that processes issue transactions sequentially, *i.e.*, a process starts a new transaction only after the previous transaction is committed or aborted. A primitive is a generic *read-modify-write* (*RMW*) procedure applied to a base object [10]. It is characterized by a pair of functions $\langle g, h \rangle$: given the current state of the base object, g is an *update function* that computes its state after the primitive is applied, while h is a *response function* that specifies the outcome of the primitive returned to the process. A RMW primitive is *trivial* if it never changes the value of the base object to which it is applied. Otherwise, it is *nontrivial.* An RMW primitive $\langle g, h \rangle$ is *conditional* if there exists v, w such that $g(v, w) = v$ and there exists v, w such that $g(v, w) \neq v$. For *e.g*, *compare-and-swap (CAS)* and *load-linked/store-conditional (LL/SC)* are nontrivial conditional RMW primitives while *fetch-and-add* is an example of a nontrivial RMW primitive that is not conditional.

Executions and Configurations. An *event* of a process p_i (sometimes we say *step* of p_i) is an invocation or response of an operation performed by p_i or a rmw primitive $\langle g, h \rangle$ applied by p_i to a base object b along with its response r (we call it a *rmw event* and write $(b, \langle g, h \rangle, r, i)$). A *configuration* specifies the value of each base object and the state of each process. The *initial configuration* is the configuration in which all base objects have their initial values and all processes are in their initial states.

An *execution fragment* is a (finite or infinite) sequence of events. An *execution* of an implementation I is an execution fragment where, starting from the initial configuration, each event is issued according to I and each response of a rmw event $(b, \langle g, h \rangle, r, i)$ matches the state of b resulting from all preceding events. An execution $E \cdot E'$, denoting the concatenation of E and E', is an *extension* of E and we say that E' *extends* E.

Let E be an execution fragment. For every transaction identifier k, $E|k$ denotes the subsequence of E restricted to events of transaction T_k. If $E|k$ is non-empty, we say that T_k *participates* in E, else we say E is T_k*-free.* Two executions E and E' are *indistinguishable* to a set $\mathscr{T}$ of transactions, if for each transaction $T_k \in \mathscr{T}$, $E|k = E'|k$. A TM *history* is the subsequence of an execution consisting of the invocation and response events of t-operations.

The *read set* (resp., the *write set*) of a transaction T_k in an execution E, denoted $\mathit{Rset}(T_k)$ (and resp. $\mathit{Wset}(T_k)$), is the set of t-objects on which T_k invokes reads (and resp. writes) in E. The *data set* of T_k is $\mathit{Dset}(T_k) = \mathit{Rset}(T_k) \cup \mathit{Wset}(T_k)$. A transaction is called *read-only* if $\mathit{Wset}(T_k) = \emptyset$; *write-only* if $\mathit{Rset}(T_k) = \emptyset$ and *updating* if $\mathit{Wset}(T_k) \neq \emptyset$. Note that, in our TM model, the data set of a transaction is not known apriori and it is identifiable only by the set of data items the transaction has invoked a read or write on in the given execution.

Transaction Orders. Let $\mathit{txns}(E)$ denote the set of transactions that participate in E. An execution E is *sequential* if every invocation of a t-operation is either the last event in the history H exported by E or is immediately followed by a matching response. We assume that executions are *well-formed*: no

process invokes a new operation before the previous operation returns. Specifically, we assume that for all T_k, $E|k$ begins with the invocation of a t-operation, is sequential and has no events after A_k or C_k. A transaction $T_k \in \mathit{txns}(E)$ is *complete in* E if $E|k$ ends with a response event. The execution E is *complete* if all transactions in $\mathit{txns}(E)$ are complete in E. A transaction $T_k \in \mathit{txns}(E)$ is *t-complete* if $E|k$ ends with A_k or C_k; otherwise, T_k is *t-incomplete*. T_k is *committed* (resp., *aborted*) in E if the last event of T_k is C_k (resp., A_k). The execution E is *t-complete* if all transactions in $\mathit{txns}(E)$ are t-complete.

For transactions $\{T_k, T_m\} \in \mathit{txns}(E)$, we say that T_k *precedes* T_m in the *real-time order* of E, denoted $T_k \prec_E^{RT} T_m$, if T_k is t-complete in E and the last event of T_k precedes the first event of T_m in E. If neither $T_k \prec_E^{RT} T_m$ nor $T_m \prec_E^{RT} T_k$, then T_k and T_m are *concurrent* in E. An execution E is *t-sequential* if there are no concurrent transactions in E.

Contention. We say that a configuration C after an execution E is *quiescent* (and resp. *t-quiescent*) if every transaction $T_k \in \mathit{txns}(E)$ is complete (and resp. t-complete) in C. If a transaction T is incomplete in an execution E, it has exactly one *enabled* event, which is the next event the transaction will perform according to the TM implementation. Events e and e' of an execution E *contend* on a base object b if they are both events on b in E and at least one of them is nontrivial (the event is trivial (and resp. nontrivial) if it is the application of a trivial (and resp. nontrivial) primitive). We say that a transaction T is *poised to apply an event* e *after* E if e is the next enabled event for T in E. We say that transactions T and T' *concurrently contend on* b *in* E if they are each poised to apply contending events on b after E.

We say that an execution fragment E is *step contention-free for t-operation* op_k if the events of $E|op_k$ are contiguous in E. We say that an execution fragment E is *step contention-free for* T_k if the events of $E|k$ are contiguous in E. We say that E is *step contention-free* if E is step contention-free for all transactions that participate in E.

3 TM Classes

TM-correctness. We say that $\mathit{read}_k(X)$ is *legal* in a t-sequential execution E if it returns the *latest written value* of X, and E is *legal* if every $\mathit{read}_k(X)$ in H that does not return A_k is legal in E.

A finite history H is *opaque* if there is a legal t-complete t-sequential history S, such that (1) for any two transactions $T_k, T_m \in \mathit{txns}(H)$, if $T_k \prec_H^{RT} T_m$, then T_k precedes T_m in S, and (2) S is equivalent to a *completion* of H.

A finite history H is *strictly serializable* if there is a legal t-complete t-sequential history S, such that (1) for any two transactions $T_k, T_m \in \mathit{txns}(H)$, if $T_k \prec_H^{RT} T_m$, then T_k precedes T_m in S, and (2) S is equivalent to $\mathit{cseq}(\bar{H})$, where $\bar{H}$ is some completion of H and $\mathit{cseq}(\bar{H})$ is the subsequence of $\bar{H}$ reduced to committed transactions in $\bar{H}$.

We refer to S as an opaque (and resp. strictly serializable) *serialization* of H.

TM-liveness. We say that a TM implementation M provides *interval-contention free (ICF)* TM-liveness if for every finite execution E of M such that the configuration after E is quiescent, and every transaction T_k that applies the invocation of a t-operation op_k immediately after E, the finite step contention-free extension for op_k contains a matching response.

A TM implementation M provides *wait-free TM-liveness* if in every execution of M, every t-operation returns a matching response in a finite number of its steps.

TM-progress. We say that a TM implementation provides *sequential TM-progress* (also called *minimal progressiveness* [13]) if every transaction running step contention-free from a t-quiescent configuration commits within a finite number of steps.

We say that transactions T_i, T_j *conflict* in an execution E on a t-object X if $X \in Dset(T_i) \cap Dset(T_j)$, and $X \in Wset(T_i) \cup Wset(T_j)$.

A TM implementation M provides *progressive* TM-progress (or *progressiveness*) if for every execution E of M and every transaction $T_i \in txns(E)$ that returns A_i in E, there exists a transaction $T_k \in txns(E)$ such that T_k and T_i are concurrent and conflict in E [13].

Let $CObj_H(T_i)$ denote the set of t-objects over which transaction $T_i \in txns(H)$ conflicts with any other transaction in history H, *i.e.*, $X \in CObj_H(T_i)$, *iff* there exist transactions T_i and T_k that conflict on X in H. Let $Q \subseteq txns(H)$ and $CObj_H(Q) = \bigcup_{T_i \in Q} CObj_H(T_i)$.

Let $CTrans(H)$ denote the set of non-empty subsets of $txns(H)$ such that a set Q is in $CTrans(H)$ if no transaction in Q conflicts with a transaction not in Q.

Definition 1. *A TM implementation M is* strongly progressive *if M is weakly progressive and for every history H of M and for every set $Q \in CTrans(H)$ such that $|CObj_H(Q)| \leq 1$, some transaction in Q is not aborted in H.*

Invisible Reads. A TM implementation M uses *invisible reads* if for every execution E of M and for every read-only transaction $T_k \in txns(E)$, $E|k$ does not contain any nontrivial events.

In this paper, we introduce a definition of *weak* invisible reads. For any execution E and any t-operation π_k invoked by some transaction $T_k \in txns(E)$, let $E|\pi_k$ denote the subsequence of E restricted to events of π_k in E.

We say that a TM implementation M satisfies *weak invisible reads* if for any execution E of M and every transaction $T_k \in txns(E)$; $Rset(T_k) \neq \emptyset$ that is not concurrent with any transaction $T_m \in txns(E)$, $E|\pi_k$ does not contain any nontrivial events, where π_k is any t-read operation invoked by T_k in E.

Disjoint-Access Parallelism (DAP). Let $\tau_E(T_i, T_j)$ be the set of transactions (T_i and T_j included) that are concurrent to at least one of T_i and T_j in E. Let $G(T_i, T_j, E)$ be an undirected graph whose vertex set is $\bigcup_{T \in \tau_E(T_i,T_j)} Dset(T)$ and there is an edge between t-objects X and Y *iff* there exists $T \in \tau_E(T_i, T_j)$ such that $\{X, Y\} \in Dset(T)$. We say that T_i and T_j are *disjoint-access* in E if there is no path between a t-object in $Dset(T_i)$ and a t-object in $Dset(T_j)$ in $G(T_i, T_j, E)$.

A TM implementation M is *weak disjoint-access parallel (weak DAP)* if, for all executions E of M, transactions T_i and T_j concurrently contend on the same base object in E only if T_i and T_j are not disjoint-access in E or there exists a t-object $X \in Dset(T_i) \cap Dset(T_j)$ [5,20].

Lemma 1. *([5,18]) Let M be any weak DAP TM implementation. Let $\alpha \cdot \rho_1 \cdot \rho_2$ be any execution of M where ρ_1 (and resp. ρ_2) is the step contention-free execution fragment of transaction $T_1 \notin txns(\alpha)$ (and resp. $T_2 \notin txns(\alpha)$) and transactions T_1, T_2 are disjoint-access in $\alpha \cdot \rho_1 \cdot \rho_2$. Then, T_1 and T_2 do not contend on any base object in $\alpha \cdot \rho_1 \cdot \rho_2$.*

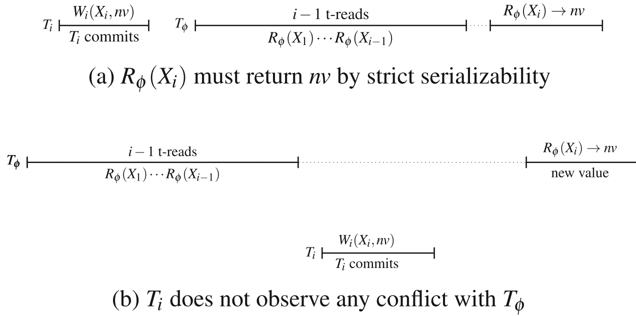

Fig. 1. Executions in the proof of Lemma 2; By weak DAP, T_ϕ cannot distinguish this from the execution in Fig. 1(a)

4 Time and Space Complexity of Sequential TMs

In this section, we prove that (1) that a read-only transaction in an opaque TM featured with *weak* DAP and *weak* invisible reads must *incrementally* validate every next read operation, and (2) a strictly serializable TM (under weak DAP and weak read invisibility), must have a read-only transaction that accesses a linear (in the size of the transaction's read set) number of distinct base objects in the course of performing its last t-read and tryCommit operations.

We first prove the following lemma concerning strictly serializable weak DAP TM implementations.

Lemma 2. *Let M be any strictly serializable, weak DAP TM implementation that provides sequential TM-progress. Then, for all $i \in \mathbb{N}$, M has an execution of the form $\pi^{i-1} \cdot \rho^i \cdot \alpha^i$ where,*

- *π^{i-1} is the complete step contention-free execution of read-only transaction T_ϕ that performs $(i-1)$ t-reads: $read_\phi(X_1) \cdots read_\phi(X_{i-1})$,*
- *ρ^i is the t-complete step contention-free execution of a transaction T_i that writes $nv_i \neq v_i$ to X_i and commits,*
- *α_i is the complete step contention-free execution fragment of T_ϕ that performs its ith t-read: $read_\phi(X_i) \rightarrow nv_i$.*

Proof. By sequential TM-progress, M has an execution of the form $\rho^i \cdot \pi^{i-1}$. Since $Dset(T_k) \cap Dset(T_i) = \emptyset$ in $\rho^i \cdot \pi^{i-1}$, by Lemma 1, transactions T_ϕ and T_i do not contend on any base object in execution $\rho^i \cdot \pi^{i-1}$. Thus, $\rho^i \cdot \pi^{i-1}$ is also an execution of M.

By assumption of strict serializability, $\rho^i \cdot \pi^{i-1} \cdot \alpha_i$ is an execution of M in which the t-read of X_i performed by T_ϕ must return nv_i. But $\rho^i \cdot \pi^{i-1} \cdot \alpha_i$ is indistinguishable to T_ϕ from $\pi^{i-1} \cdot \rho^i \cdot \alpha_i$. Thus, M has an execution of the form $\pi^{i-1} \cdot \rho^i \cdot \alpha_i$.

Theorem 1. *For every weak DAP TM implementation M that provides ICF TM-liveness, sequential TM-progress and uses weak invisible reads,*

(1) If M is opaque, for every $m \in \mathbb{N}$, there exists an execution E of M such that some transaction $T \in txns(E)$ performs $\Omega(m^2)$ steps, where $m = |Rset(T_k)|$.
(2) if M is strictly serializable, for every $m \in \mathbb{N}$, there exists an execution E of M such that some transaction $T_k \in txns(E)$ accesses at least $m-1$ distinct base objects during the executions of the mth t-read operation and $tryC_k()$, where $m = |Rset(T_k)|$.

Proof. For all $i \in \{1, \dots, m\}$, let v be the initial value of t-object X_i.

(1) Suppose that M is opaque. Let π^m denote the complete step contention-free execution of a transaction T_ϕ that performs m t-reads: $read_\phi(X_1) \cdots read_\phi(X_m)$ such that for all $i \in \{1, \dots, m\}$, $read_\phi(X_i) \rightarrow v$.

By Lemma 2, for all $i \in \{2, \dots, m\}$, M has an execution of the form $E^i = \pi^{i-1} \cdot \rho^i \cdot \alpha_i$.

For each $i \in \{2, \dots, m\}$, $j \in \{1, 2\}$ and $\ell \leq (i-1)$, we now define an execution of the form $\mathbb{E}^i_{j\ell} = \pi^{i-1} \cdot \beta^\ell \cdot \rho^i \cdot \alpha^i_j$ as follows:

- β^ℓ is the t-complete step contention-free execution fragment of a transaction T_ℓ that writes $nv_\ell \neq v$ to X_ℓ and commits
- α^i_1 (and resp. α^i_2) is the complete step contention-free execution fragment of $read_\phi(X_i) \rightarrow v$ (and resp. $read_\phi(X_i) \rightarrow A_\phi$).

Claim 1. *For all $i \in \{2, \dots, m\}$ and $\ell \leq (i-1)$, M has an execution of the form $\mathbb{E}^i_{1\ell}$ or $\mathbb{E}^i_{2\ell}$.*

Proof. For all $i \in \{2, \dots, m\}$, π^{i-1} is an execution of M. By assumption of weak invisible reads and sequential TM-progress, T_ℓ must be committed in $\pi^{i-1} \cdot \rho^\ell$ and M has an execution of the form $\pi^{i-1} \cdot \beta^\ell$. By the same reasoning, since T_i and T_ℓ have disjoint data sets, M has an execution of the form $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i$.

Since the configuration after $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i$ is quiescent, by ICF TM-liveness, $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i$ extended with $read_\phi(X_i)$ must return a matching response. If $read_\phi(X_i) \rightarrow v_i$, then clearly $\mathbb{E}^i_1$ is an execution of M with T_ϕ, T_{i-1}, T_i being a valid serialization of transactions. If $read_\phi(X_i) \rightarrow A_\phi$, the same serialization justifies an opaque execution.

Suppose by contradiction that there exists an execution of M such that $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i$ is extended with the complete execution of $read_\phi(X_i) \rightarrow r$; $r \notin \{A_\phi, v\}$.

The only plausible case to analyse is when $r = nv$. Since $read_\phi(X_i)$ returns the value of X_i updated by T_i, the only possible serialization for transactions is T_ℓ, T_i, T_ϕ; but $read_\phi(X_\ell)$ performed by T_k that returns the initial value v is not legal in this serialization—contradiction.

We now prove that, for all $i \in \{2, \ldots, m\}$, $j \in \{1, 2\}$ and $\ell \leq (i-1)$, transaction T_ϕ must access $(i-1)$ different base objects during the execution of $read_\phi(X_i)$ in the execution $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i \cdot \alpha^i_j$.

By the assumption of weak invisible reads, the execution $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i \cdot \alpha^i_j$ is indistinguishable to transactions T_ℓ and T_i from the execution $\tilde{\pi}^{i-1} \cdot \beta^\ell \cdot \rho^i \cdot \alpha^i_j$, where $Rset(T_\phi) = \emptyset$ in $\tilde{\pi}^{i-1}$. But transactions T_ℓ and T_i are disjoint-access in $\tilde{\pi}^{i-1} \cdot \beta^\ell \cdot \rho^i$ and by Lemma 1, they cannot contend on the same base object in this execution.

Consider the $(i-1)$ different executions: $\pi^{i-1} \cdot \beta^1 \cdot \rho^i$, ..., $\pi^{i-1} \cdot \beta^{i-1} \cdot \rho^i$. For all $\ell, \ell' \leq (i-1); \ell' \neq \ell$, M has an execution of the form $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i \cdot \beta^{\ell'}$ in which transactions T_ℓ and $T_{\ell'}$ access mutually disjoint data sets. By weak invisible reads and Lemma 1, the pairs of transactions $T_{\ell'}$, T_i and $T_{\ell'}$, T_ℓ do not contend on any base object in this execution. This implies that $\pi^{i-1} \cdot \beta^\ell \cdot \beta^{\ell'} \cdot \rho^i$ is an execution of M in which transactions T_ℓ and $T_{\ell'}$ each apply nontrivial primitives to mutually disjoint sets of base objects in the execution fragments β^ℓ and $\beta^{\ell'}$ respectively (by Lemma 1).

This implies that for any $j \in \{1, 2\}$, $\ell \leq (i-1)$, the configuration C^i after E^i differs from the configurations after $\mathbb{E}^i_{j\ell}$ only in the states of the base objects that are accessed in the fragment β^ℓ. Consequently, transaction T_ϕ must access at least $i-1$ different base objects in the execution fragment π^i_j to distinguish configuration C^i from the configurations that result after the $(i-1)$ different executions $\pi^{i-1} \cdot \beta^1 \cdot \rho^i$, ..., $\pi^{i-1} \cdot \beta^{i-1} \cdot \rho^i$ respectively.

Thus, for all $i \in \{2, \ldots, m\}$, transaction T_ϕ must perform at least $i-1$ steps while executing the ith t-read in π^i_j and T_ϕ itself must perform $\sum_{i=1}^{m-1} i = \frac{m(m-1)}{2}$ steps.

(2) Suppose that M is strictly serializable, but not opaque. Since M is strictly serializable, by Lemma 2, it has an execution of the form $E = \pi^{m-1} \cdot \rho^m \cdot \alpha_m$.

For each $\ell \leq (i-1)$, we prove that M has an execution of the form $E_\ell = \pi^{m-1} \cdot \beta^\ell \cdot \rho^m \cdot \bar{\alpha}^m$ where $\bar{\alpha}^m$ is the complete step contention-free execution fragment of $read_\phi(X_m)$ followed by the complete execution of $tryC_\phi$. Indeed, by weak invisible reads, π^{m-1} does not contain any nontrivial events and the execution $\pi^{m-1} \cdot \beta^\ell \cdot \rho^m$ is indistinguishable to transactions T_ℓ and T_m from the executions $\tilde{\pi}^{m-1} \cdot \beta^\ell$ and $\tilde{\pi}^{m-1} \cdot \beta^\ell \cdot \rho^m$ respectively, where $Rset(T_\phi) = \emptyset$ in $\tilde{\pi}^{m-1}$. Thus, applying Lemma 1, transactions $\beta^\ell \cdot \rho^m$ do not contend on any base object in the execution $\pi^{m-1} \cdot \beta^\ell \cdot \rho^m$. By ICF TM-liveness, $read_\phi(X_m)$ and $tryC_\phi$ must return matching responses in the execution fragment $\bar{\alpha}^m$ that extends $\pi^{m-1} \cdot \beta^\ell \cdot \rho^m$. Consequently, for each $\ell \leq (i-1)$, M has an execution of the form $E_\ell = \pi^{m-1} \cdot \beta^\ell \cdot \rho^m \cdot \bar{\alpha}^m$ such that transactions T_ℓ and T_m do not contend on any base object.

Strict serializability of M means that if $read_\phi(X_m) \rightarrow nv$ in the execution fragment $\bar{\alpha}^m$, then $tryC_\phi$ must return A_ϕ. Otherwise if $read_\phi(X_m) \rightarrow v$ (i.e. the initial value of X_m), then $tryC_\phi$ may return A_ϕ or C_ϕ.

Thus, as with (1), in the worst case, T_ϕ must access at least $m-1$ distinct base objects during the executions of $read_\phi(X_m)$ and $tryC_\phi$ to distinguish the configuration C^i from the configurations after the $m-1$ different executions $\pi^{m-1} \cdot \beta^1 \cdot \rho^m, \ldots, \pi^{m-1} \cdot \beta^{m-1} \cdot \rho^m$ respectively.

5 RMR Complexity of Strongly Progressive TMs

In this section, we prove every strongly progressive strictly serializable TM providing wait-free TM-liveness that uses only read, write and *conditional* RMW primitives has an execution in which in which n concurrent processes perform transactions on a single data item and incur $\Omega(\log n)$ *remote memory references* [2].

Remote Memory References(RMR) [3]. In the *cache-coherent (CC) shared memory*, each process maintains *local* copies of shared objects inside its cache, whose consistency is ensured by a coherence protocol. Informally, we say that an access to a base object b is *remote* to a process p and causes a *remote memory reference (RMR)* if p's cache contains a cached copy of the object that is out of date or *invalidated*; otherwise the access is *local*.

Algorithm 1. Mutual-exclusion object L from a strongly progressive, strict serializable TM M; code for process p_i; $1 \leq i \leq n$

```
1:  Local variables:
2:     bit face_i, for each process p_i

3:  Shared objects:
4:     strongly progressive, strictly
5:       serializable TM M
6:       t-object X, initially ⊥
7:       storing value v ∈ {[p_i, face_i]} ∪ {⊥}
8:     for each tuple [p_i, face_i]
9:       Done[p_i, face_i] ∈ {true, false}
10:      Succ[p_i, face_i] ∈ {p_1, ..., p_n} ∪ {⊥}
11:    for each p_i and j ∈ {1, ..., n} \ {i}
12:      Lock[p_i][p_j] ∈ {locked, unlocked}

13: Function: func():
14:    atomic using M
15:       value := tx-read(X)
16:       tx-write(X, [p_i, face_i])
17:    on abort Return false
18:    Return value

19: Entry:
20:    face_i := 1 − face_i
21:    Done[p_i, face_i].write(false)
22:    Succ[p_i, face_i].write(⊥)
23:    while (prev ← func) = false do
24:       no op
25:    end while
26:    if prev ≠ ⊥ then
27:       Lock[p_i][prev.pid].write(locked)
28:       Succ[prev].write(p_i)
29:       if Done[prev] = false then
30:          while Lock[p_i][prev.pid] = unlocked do
31:             no op
32:          end while
33:    Return ok
34:    // Critical section

35: Exit:
36:    Done[p_i, face_i].write(true)
37:    Lock[Succ[p_i, face_i]][p_i].write(unlocked)
38:    Return ok
```

In the *write-through (CC) protocol*, to read a base object b, process p must have a cached copy of b that has not been invalidated since its previous read. Otherwise, p incurs a RMR. To write to b, p causes a RMR that invalidates all cached copies of b and writes to the main memory.

In the *write-back (CC) protocol*, p reads a base object b without causing a RMR if it holds a cached copy of b in shared or exclusive mode; otherwise the access of b causes a RMR that (1) invalidates all copies of b held in exclusive mode, and writing b back to the main memory, (2) creates a cached copy of b in shared mode. Process p can write to b without causing a RMR if it holds a copy of b in exclusive mode; otherwise p causes a RMR that invalidates all cached copies of b and creates a cached copy of b in exclusive mode.

In the *distributed shared memory (DSM)*, each register is forever assigned to a single process and it *remote* to the others. Any access of a remote register causes a RMR.

Mutual Exclusion. The *mutex object* supports two operations: *Enter* and *Exit*, both of which return the response *ok*. We say that a process p_i *is in the critical section after an execution* π if π contains the invocation of `Enter` by p_i that returns *ok*, but does not contain a subsequent invocation of `Exit` by p_i in π.

A mutual exclusion implementation satisfies the following properties:

(*Mutual-exclusion*) After any execution π, there exists at most one process that is in the critical section.

(*Deadlock-freedom*) Let π be any execution that contains the invocation of `Enter` by process p_i. Then, in every extension of π in which every process takes infinitely many steps, some process is in the critical section.

(*Finite-exit)* Every process completes the `Exit` operation within a finite number of steps.

5.1 Mutual Exclusion from a Strongly Progressive TM

We describe an implementation of a mutex object $L(M)$ from a strictly serializable, strongly progressive TM implementation M providing wait-free TM-liveness (Algorithm 1). The algorithm is based on the mutex implementation in [15].

Given a sequential implementation, we use a TM to execute the sequential code in a concurrent environment by encapsulating each sequential operation within an *atomic* transaction that replaces each read and write of a t-object with the transactional read and write implementations, respectively. If the transaction commits, then the result of the operation is returned; otherwise if one of the transactional operations aborts. For instance, in Algorithm 1, we wish to atomically read a t-object X, write a new value to it and return the old value of X prior to this write. To achieve this, we employ a strictly serializable TM implementation M. Moreover, we assume that M is strongly progressive, *i.e.*, in every execution, at least one transaction successfully commits and the value of X is returned.

Shared Objects. We associate each process p_i with two alternating identities $[p_i, \mathit{face}_i]$; $\mathit{face}_i \in \{0, 1\}$. The strongly progressive TM implementation M is used to enqueue processes that attempt to enter the critical section within a single

t-object X (initially $\bot$). For each $[p_i, face_i]$, $L(M)$ uses a register bit $Done[p_i, face_i]$ that indicates if this face of the process has left the critical section or is executing the `Entry` operation. Additionally, we use a register $Succ[p_i, face_i]$ that stores the process expected to succeed p_i in the critical section. If $Succ[p_i, face_i] = p_j$, we say that p_j is the *successor of* p_i (and p_i is the *predecessor* of p_j). Intuitively, this means that p_j is expected to enter the critical section immediately after p_i. Finally, $L(M)$ uses a 2-dimensional bit array *Lock*: for each process p_i, there are $n-1$ registers associated with the other processes. For all $j \in \{0, \ldots, n-1\} \setminus \{i\}$, the registers $Lock[p_i][p_j]$ are local to p_i and registers $Lock[p_j][p_i]$ are remote to p_i. Process p_i can only access registers in the *Lock* array that are local or remote to it.

Entry Operation. A process p_i adopts a new identity $face_i$ and writes *false* to $Done(p_i, face_i)$ to indicate that p_i has started the `Entry` operation. Process p_i now initializes the successor of $[p_i, face_i]$ by writing $\bot$ to $Succ[p_i, face_i]$. Now, p_i uses a strongly progressive TM implementation M to atomically store its *pid* and identity i.e., $face_i$ to t-object X and returns the *pid* and identity of its *predecessor*, say $[p_j, face_j]$. Intuitively, this suggests that $[p_i, face_i]$ is scheduled to enter the critical section immediately after $[p_j, face_j]$ exits the critical section. Note that if p_i reads the initial value of t-object X, then it immediately enters the critical section. Otherwise it writes *locked* to the register $Lock[p_i, p_j]$ and sets itself to be the successor of $[p_j, face_j]$ by writing p_i to $Succ[p_j, face_j]$. Process p_i now checks if p_j has started the `Exit` operation by checking if $Done[p_j, face_j]$ is set. If it is, p_i enters the critical section; otherwise p_i spins on the register $Lock[p_i][p_j]$ until it is *unlocked*.

Exit Operation. Process p_i first indicates that it has exited the critical section by setting $Done[p_i, face_i]$, following which it *unlocks* the register $Lock[Succ[p_i, face_i]][p_i]$ to allow p_i's successor to enter the critical section.

5.2 Proof of Correctness

Lemma 3. *The implementation $L(M)$ (Algorithm 1) satisfies mutual exclusion.*

Proof. Let E be any execution of $L(M)$. We say that $[p_i, face_i]$ is the *successor* of $[p_j, face_j]$ if p_i reads the value of *prev* in Line 25 to be $[p_j, face_j]$ (and $[p_j, face_j]$ is the *predecessor* of $[p_i, face_i]$); otherwise if p_i reads the value to be $\bot$, we say that p_i has no predecessor.

Suppose by contradiction that there exist processes p_i and p_j that are both inside the critical section after E. Since p_i is inside the critical section, either (1) p_i read $prev = \bot$ in Line 23, or (2) p_i read that $Done[prev]$ is *true* (Line 29) or p_i reads that $Done[prev]$ is *false* and $Lock[p_i][prev.pid]$ is *unlocked* (Line 30).

(Case 1) Suppose that p_i read $prev = \bot$ and entered the critical section. Since in this case, p_i does not have any predecessor, some other process that returns successfully from the *while* loop in Line 25 must be successor of p_i in E. Since there exists $[p_j, face_j]$ also inside the critical section after E, p_j reads that either $[p_i, face_i]$ or some other process to be its predecessor. Observe that there must

exist some such process $[p_k, face_k]$ whose predecessor is $[p_i, face_i]$. Hence, without loss of generality, we can assume that $[p_j, face_j]$ is the successor of $[p_i, face_i]$. By our assumption, $[p_j, face_j]$ is also inside the critical section. Thus, p_j *locked* the register $Lock[p_j, p_i]$ in Line 27 and set itself to be p_i's successor in Line 28. Then, p_j read that $Done[p_i, face_i]$ is *true* or read that $Done[p_i, face_i]$ is *false* and waited until $Lock[p_j, p_i]$ is *unlocked* and then entered the critical section. But this is possible only if p_i has left the critical section and updated the registers $Done[p_i, face_i]$ and $Lock[p_j, p_i]$ in Lines 36 and 37 respectively—contradiction to the assumption that $[p_i, face_i]$ is also inside the critical section after E.

(Case 2) Suppose that p_i did not read $prev = \perp$ and entered the critical section. Thus, p_i read that $Done[prev]$ is *false* in Line 29 and $Lock[p_i][prev.pid]$ is *unlocked* in Line 30, where *prev* is the predecessor of $[p_i, face_i]$. As with case 1, without loss of generality, we can assume that $[p_j, face_j]$ is the successor of $[p_i, face_i]$ or $[p_j, face_j]$ is the predecessor of $[p_i, face_i]$.

Suppose that $[p_j, face_j]$ is the predecessor of $[p_i, face_i]$, *i.e.*, p_i writes the value $[p_i, face_i]$ to the register $Succ[p_j, face_j]$ in Line 28. Since $[p_j, face_j]$ is also inside the critical section after E, process p_i must read that $Done[p_j, face_j]$ is *true* in Line 29 and $Lock[p_i, p_j]$ is *locked* in Line 30. But then p_i could not have entered the critical section after E—contradiction.

Suppose that $[p_j, face_j]$ is the successor of $[p_i, face_i]$, *i.e.*, p_j writes the value $[p_j, face_j]$ to the register $Succ[p_i, face_i]$. Since both p_i and p_j are inside the critical section after E, process p_j must read that $Done[p_i, face_i]$ is *true* in Line 29 and $Lock[p_j, p_i]$ is *locked* in Line 30. Thus, p_j must spin on the register $Lock[p_j, p_i]$, waiting for it to be *unlocked* by p_i before entering the critical section—contradiction to the assumption that both p_i and p_j are inside the critical section.

Thus, $L(M)$ satisfies mutual-exclusion.

Lemma 4. *The implementation $L(M)$ (Algorithm 1) provides deadlock-freedom.*

Proof. Let E be any execution of $L(M)$. Observe that a process may be stuck indefinitely only in Lines 23 and 30 as it performs the *while* loop.

Since M is strongly progressive and provides wait-free TM-liveness, in every execution E that contains an invocation of `Enter` by process p_i, some process returns *true* from the invocation of *func*() in Line 23.

Now consider a process p_i that returns successfuly from the *while* loop in Line 23. Suppose that p_i is stuck indefinitely as it performs the *while* loop in Line 30. Thus, no process has *unlocked* the register $Lock[p_i][prev.pid]$ by writing to it in the `Exit` section. Recall that since $[p_i, face_i]$ has reached the *while* loop in Line 30, $[p_i, face_i]$ necessarily has a predecessor, say $[p_j, face_j]$, and has set itself to be p_j's successor by writing p_i to register $Succ[p_j, face_j]$ in Line 28. Consider the possible two cases: the predecessor of $[p_j, face_j$ is some process $p_k; k \neq i$ or the predecessor of $[p_j, face_j$ is the process p_i itself.

(Case 1) Since by assumption, process p_j takes infinitely many steps in E, the only reason that p_j is stuck without entering the critical section is that

$[p_k, \mathit{face}_k]$ is also stuck in the *while* loop in Line 30. Note that it is possible for us to iteratively extend this execution in which p_k's predecessor is a process that is not p_i or p_j that is also stuck in the *while* loop in Line 30. But then the last such process must eventually read the corresponding *Lock* to be *unlocked* and enter the critical section. Thus, in every extension of E in which every process takes infinitely many steps, some process will enter the critical section.

(Case 2) Suppose that the predecessor of $[p_j, \mathit{face}_j$ is the process p_i itself. Thus, as $[p_i, \mathit{face}]$ is stuck in the *while* loop waiting for $\mathit{Lock}[p_i, p_j]$ to be *unlocked* by process p_j, p_j leaves the critical section, *unlocks* $\mathit{Lock}[p_i, p_j]$ in Line 37 and prior to the read of $\mathit{Lock}[p_i, p_j]$, p_j re-starts the `Entry` operation, writes *false* to $\mathit{Done}[p_j, 1 - \mathit{face}_j]$ and sets itself to be the successor of $[p_i, \mathit{face}_i]$ and spins on the register $\mathit{Lock}[p_j, p_i]$. However, observe that process p_i, which takes infinitely many steps by our assumption must eventually read that $\mathit{Lock}[p_i, p_j]$ is *unlocked* and enter the critical section, thus establishing deadlock-freedom.

We say that a TM implementation M *accesses a single t-object* if in every execution E of M and every transaction $T \in \mathit{txns}(E)$, $|\mathit{Dset}(T)| \leq 1$. We can now prove the following theorem:

Theorem 2. *Any strictly serializable, strongly progressive TM implementation M providing wait-free TM-liveness that accesses a single t-object implies a* deadlock-free, finite exit *mutual exclusion implementation $L(M)$ such that the RMR complexity of M is within a constant factor of the RMR complexity of $L(M)$.*

Proof. (Mutual-exclusion) Follows from Lemma 3.

(Finite-exit) The proof is immediate since the `Exit` operation contains no unbounded loops or waiting statements.

(Deadlock-freedom) Follows from Lemma 4.

(RMR complexity) First, let us consider the CC model. Observe that every event not on M performed by a process p_i as it performs the `Entry` or `Exit` operations incurs $O(1)$ RMR cost clearly, possibly barring the *while* loop executed in Line 30. During the execution of this *while* loop, process p_i spins on the register $\mathit{Lock}[p_i][p_j]$, where p_j is the predecessor of p_i. Observe that p_i's cached copy of $\mathit{Lock}[p_i][p_j]$ may be invalidated only by process p_j as it *unlocks* the register in Line 37. Since no other process may write to this register and p_i terminates the *while* loop immediately after the write to $\mathit{Lock}[p_i][p_j]$ by p_j, p_i incurs $O(1)$ RMR's. Thus, the overall RMR cost incurred by M is within a constant factor of the RMR cost of $L(M)$.

Now we consider the DSM model. As with the reasoning for the CC model, every event not on M performed by a process p_i as it performs the `Entry` or `Exit` operations incurs $O(1)$ RMR cost clearly, possibly barring the *while* loop executed in Line 30. During the execution of this *while* loop, process p_i spins on the register $\mathit{Lock}[p_i][p_j]$, where p_j is the predecessor of p_i. Recall that $\mathit{Lock}[p_i][p_j]$ is a register that is local to p_i and thus, p_i does not incur any RMR cost on account of executing this loop. It follows that p_i incurs $O(1)$ RMR cost in the DSM model. Thus, the overall RMR cost of M is within a constant factor of the RMR cost of $L(M)$ in the DSM model.

Theorem 3. *([3]) Any deadlock-free, finite-exit mutual exclusion implementation from read, write and conditional primitives has an execution whose RMR complexity is* $\Omega(n \log n)$.

Theorems 2 and 3 imply:

Theorem 4. *Any strictly serializable, strongly progressive TM implementation providing wait-free TM-liveness from read, write and conditional primitives that accesses a single t-object has an execution whose RMR complexity is* $\Omega(n \log n)$.

6 Related Work and Concluding Remarks

Theorem 1 improves the read-validation step-complexity lower bound [12,13] derived for *strict-data partitioning* (a very strong version of DAP) and (strong) invisible reads. In a *strict data partitioned* TM, the set of base objects used by the TM is split into disjoint sets, each storing information only about a single data item. Indeed, every TM implementation that is strict data-partitioned satisfies weak DAP, but not vice-versa. The definition of invisible reads assumed in [12,13] requires that a t-read operation does not apply nontrivial events in any execution. Theorem 1 however, assumes *weak* invisible reads, stipulating that t-read operations of a transaction T do not apply nontrivial events only when T is not concurrent with any other transaction.

The notion of weak DAP used in this paper was introduced by Attiya *et al.* [5].

Proving a lower bound for a concurrent object by reduction to a form of mutual exclusion has previously been used in [1,13]. Guerraoui and Kapalka [13] proved that it is impossible to implement strictly serializable strongly progressive TMs that provide *wait-free* TM-liveness (every t-operation returns a matching response within a finite number of steps) using only read and write primitives. Alistarh *et al.* proved a lower bound on RMR complexity of *renaming* problem [1]. Our reduction algorithm (Sect. 5) is inspired by the $O(1)$ RMR mutual exclusion algorithm by Hyonho [15].

To the best of our knowledge, the TM properties assumed for Theorem 1 cover all of the TM implementations that are subject to the validation step-complexity [6,7,14]. It is easy to see that the lower bound of Theorem 1 is tight for both strict serializability and opacity. We refer to the TM implementation in [17] or *DSTM* [14] for the matching upper bound.

Finally, we conjecture that the lower bound of Theorem 4 is tight. Proving this remains an interesting open question.

References

1. Alistarh, D., Aspnes, J., Gilbert, S., Guerraoui, R.: The complexity of renaming. In: IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, 22–25 October, 2011, pp. 718–727, Palm Springs, CA, USA (2011)

2. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. **1**(1), 6–16 (1990)
3. Attiya, H., Hendler, D., Woelfel, P.: Tight RMR lower bounds for mutual exclusion and other problems. In: Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing, PODC 2008, pp. 447–447, New York, NY, USA. ACM (2008)
4. Attiya, H., Hillel, E.: The cost of privatization in software transactional memory. IEEE Trans. Comput. **62**(12), 2531–2543 (2013)
5. Attiya, H., Hillel, E., Milani, A.: Inherent limitations on disjoint-access parallel implementations of transactional memory. Theory Comput. Syst. **49**(4), 698–719 (2011)
6. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining STM by abolishing ownership records. SIGPLAN Not. **45**(5), 67–78 (2010)
7. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
8. Dice, D., Shavit, N.: What really makes transactions fast? In: Transact (2006)
9. Dice, D., Shavit, N.: TLRW: return of the read-write lock. In: SPAA, pp. 284–293 (2010)
10. Ellen, F., Hendler, D., Shavit, N.: On the inherent sequentiality of concurrent objects. SIAM J. Comput. **41**(3), 519–536 (2012)
11. Fraser, K.: Practical lock-freedom. Technical report, Cambridge University Computer Laborotory (2003)
12. Guerraoui, R., Kapalka, M.: The semantics of progress in lock-based transactional memory. SIGPLAN Not. **44**(1), 404–415 (2009)
13. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool, San Rafael (2010)
14. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, PODC 2003, pp. 92–101, New York, NY, USA. ACM (2003)
15. Hyonho, L.: Local-spin mutual exclusion algorithms on the DSM model using fetch-and-store objects (2003). http://www.cs.toronto.edu/pub/hlee/thesis.ps
16. Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In: PODC, pp. 151–160 (1994)
17. Kuznetsov, P., Ravi, S.: On the cost of concurrency in transactional memory. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 112–127. Springer, Heidelberg (2011)
18. Kuznetsov, P., Ravi, S.: On partial wait-freedom in transactional memory. In: Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, p. 10, 4–7 Jan 2015
19. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**, 631–653 (1979)
20. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in STM. In: PODC, pp. 16–25 (2010)
21. Tabba, F., Moir, M., Goodman, J.R., Hay, A.W., Wang, C.: Nztm: nonblocking zero-indirection transactional memory. In: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2009, pp. 204–213, New York, NY, USA. ACM (2009)