Grasping the Gap between Blocking and Non-Blocking Transactional Memories

Petr Kuznetsov, Srivatsan Ravi

Grasping the Gap between Blocking and Non-Blocking Transactional Memories

Petr Kuznetsov[1] * and Srivatsan Ravi[2]

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

Abstract. Transactional memory (TM) is an inherently optimistic synchronization abstraction: it allows concurrent processes to execute sequences of shared-data accesses (transactions) speculatively, with an option of aborting them in the future. Early TM designs avoided using locks and relied on non-blocking synchronization to ensure *obstruction-freedom*: a transaction that encounters no step contention is not allowed to abort. However, it was later observed that obstruction-free TMs perform poorly and, as a result, state-of-the-art TM implementations are nowadays *blocking*, allowing aborts because of data *conflicts* rather than step contention.
In this paper, we explain this shift in the TM practice theoretically, via complexity bounds. We prove a few important lower bounds on obstruction-free TMs. Then we present a *lock-based* TM implementation that beats all of these lower bounds. In sum, our results exhibit a considerable complexity gap between non-blocking and blocking TM implementations.

1 Introduction

Transactional memory (TM) allows concurrent processes to organize sequences of operations on shared *data items* into *transactions*. A transaction may commit, in which case its updates of data items "take effect" or it may *abort*, in which case no data item is modified. Typically, it is required that all committed transactions appear to execute sequentially, respecting the timing of non-overlapping transactions (*strict serializability*).

As a *synchronization abstraction*, TM came as an alternative to conventional lock-based synchronization, and it therefore appears natural that early TM *implementations* [12,18,23,26], *i.e.*, algorithms for implementing operations on data items using shared *base objects*, avoided using locks. Instead, early TM designs relied on *non-blocking* (sometimes also called *lock-free*) synchronization, where a prematurely halted transaction cannot prevent all other transactions from committing. Possibly the weakest non-blocking progress condition is *obstruction-freedom* [17,19] stipulating that every transaction running in the absence of *step contention*, *i.e.*, not encountering steps of concurrent transactions, must commit.

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

In 2005, Ennals [11] argued that obstruction-free TMs inherently yield poor performance, because they require transactions to forcefully abort each other. Ennals further described a *lock-based* TM implementation [10] that he claimed to outperform *DSTM* [18], the most referenced obstruction-free TM implementation at the time. Inspired by [11], more recent lock-based TMs, such as *TL* [7], *TL2* [6] and *NOrec* [5], demonstrate better performance than obstruction-free TMs on most workloads. These TMs typically ensure *progressiveness*: a transaction may be aborted only if it encounters a read-write or a write-write conflict on a data item with a concurrent transaction [14].

There is a considerable amount of empirical evidence on the performance gap between non-blocking (obstruction-free) and blocking (progressive) TM implementations but, to the best of our knowledge, no analytical result explains it. Complexity lower and upper bounds presented in this paper provide such an explanation.

Lower bounds for non-blocking TMs. Our first result focuses on strictly serializable TM implementations that satisfy two important properties: *weak disjoint-access-parallelism* (weak DAP) and *read invisibility*. Informally, weak DAP [4] is believed to improve TM performance by ensuring that two transactions concurrently contend on the same base object only if their data sets are connected in the *conflict graph*, capturing data-set overlaps among all concurrent transactions [4]. The requirement of invisible reads [3, 8], believed to be important for most commonly observed read-dominated workloads, ensures that a transaction cannot reveal any information about its read set to other transactions.

There exist weak DAP lock-based TM implementations that use invisible reads [7, 10]. In contrast, we establish that it is impossible to implement an obstruction-free TM that provides both weak DAP and read invisibility. Indeed, DSTM [18] and FSTM [12] are weak DAP, but use *visible* reads for aborting pending writing transactions.

We then derive lower bounds on the *stall complexity* [9] of obstruction-free TM implementations. Intuitively, the metric captures the fact that the time a process might have to spend before it applies a primitive on a base object can be proportional to the number of processes that try to update the object concurrently. We show that a read operation in an n-process obstruction-free TM implementation may incur $\Omega(n)$ stalls.

Finally, we prove that any *read-write (RW) DAP opaque* obstruction-free TM implementation has an execution in which a read-only transaction incurs $\Omega(n)$ non-overlapping *RAWs* or *AWARs*. Intuitively, RAW (read-after-write) or AWAR (atomic-write-after-read) patterns [2] capture the amount of "expensive synchronization", *i.e.*, the number of costly conditional primitives or memory barriers [24] incurred by the implementation. The metric appears to be more practically relevant than simple step complexity, as it accounts for expensive cache-coherence operations or conditional instructions. RW DAP, a restriction of weak DAP, defines the conflict graph based on the *write-set overlaps* among concurrent transactions and is satisfied by several popular obstruction-free im-

plementations [12, 18, 26]. For this lower bound, probably the most interesting and technically challenging, we assume opacity [15], a restriction of strict serializability that ensures safety of incomplete and aborted transactions.

	Obstruction-free	Progressive LP (Sec. 4)
strict DAP	No [13]	Yes
invisible reads+weak DAP	No (Sec. 3.1)	Yes
stall complexity of reads	$\Omega(n)$ (Sec. 3.2)	$O(1)$
RAW/AWAR complexity	$\Omega(n)$ (Sec. 3.3)	$O(1)$
read-write base objects, wait-free termination	No [15]	Yes

Fig. 1: Complexity gap between blocking and non-blocking TMs; n is the number of processes

An upper bound for blocking TMs. We describe a progressive opaque TM implementation that uses invisible reads and beats *all* the lower bounds we established for obstruction-free TMs.

Our implementation, denoted LP, (1) uses only read-write base objects and ensures that every transactional operation terminates in a wait-free manner, (2) ensures *strict DAP* [15] (a restriction of RW DAP), (3) has invisible reads, (4) performs $O(1)$ non-overlapping RAWs/AWARs per transaction, and (5) incurs $O(1)$ memory stalls per read operation. In contrast, from prior work and our lower bounds we know that (i) no OF TM that provides wait-free transactional operations can be implemented using only read-write base objects [15]; (ii) no OF TM can provide strict DAP [13]; (iii) no weak DAP OF TM has invisible reads (Section 3.1) and (iv) no OF TM ensures a constant number of stalls incurred by a read operation (Section 3.2). Finally, (v) no RW DAP *opaque* OF TM has constant RAW/AWAR complexity (Section 3.3). Thus, (iv) and (v) exhibit a linear separation between blocking and non-blocking TMs w.r.t expensive synchronization and memory stall complexity, respectively.

Our results are summarized and put in perspective in Figure 1. Altogether, we grasp a considerable complexity gap between blocking and non-blocking TM implementations, justifying theoretically the shift in TM practice we observed during the past decade.

Overcoming our lower bounds for obstruction-free TMs individually is comparatively easy. Say, TL [7] combines strict DAP with invisible reads, but it is not read-write, and it does not provide constant RAW/AWAR and stall complexities.

Coming out with a single algorithm that beats all these lower bounds is quite nontrivial. Our algorithm LP incurs the cost of *incremental validation*, *i.e.*, checking that the current read set has not changed per every new read operation. This is, however, unavoidable for invisible read algorithms [15, 21], and is, in fact, believed to yield better performance in practice than "visible" reads [5, 7, 10], and we show that it enables constant stall and RAW/AWAR complexity.

Roadmap. Sections 2 defines our model and the classes of TMs considered in this paper. Section 3 contains lower bounds for obstruction-free TMs. Section 4 describes our progressive TM implementation LP. Sections 5 and 6 present related work and concluding remarks respectively. Due to space constraints, formal proofs are delegated to the technical report [22].

2 TM Model and Properties

TM interface. *Transactional memory* (in short, TM) allows a set of data items (called *t-objects*) to be accessed via atomic *transactions*. A transaction T_k may contain the following *t-operations*: $read_k(X)$ returns a value in some domain V (denoted $read_k(X) \to v$) 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.

TM implementations. We consider an asynchronous shared-memory system in which a set of n processes, communicate by applying *primitives* on shared *base objects*. We assume that processes issue transactions sequentially, *i.e.*, a process starts a new transaction only after its previous transaction has *completed* (committed or aborted). 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*, each of which is assigned an *initial value*. A primitive is a generic *read-modify-write* (*rmw*) procedure applied to a base object [9,16]. 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*.

Executions and configurations. An *event* of a transaction T_k (sometimes we say a *step* of T_k) is a rmw primitive $\langle g, h\rangle$ applied by T_k to a base object b along with its response r (we call it a *rmw event* and write $(b, \langle g, h\rangle, r, k)$), or the invocation or the response of a t-operation performed by T_k.

A *configuration* (of a TM implementation) 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 a TM implementation M is an execution fragment where, starting from the initial configuration, each event is issued according to M and each response of a RMW event $(b, \langle g, h\rangle, r, k)$ matches the state of b resulting from the preceding events. If an execution can be represented as $E \cdot E'$ (concatenation of execution fragments E and E'), then we say that $E \cdot E'$ is an *extension* of E or E' *extends* E.

Let E be an execution fragment. For a transaction T_k (and resp. process p_k), $E|k$ denotes the subsequence of E restricted to events of T_k (and resp. p_k). If

$E|k$ is non-empty, we say that T_k (resp. p_k) *participates* in E, else we say E is T_k*-free* (resp. p_k*-free*). Two executions E and E' are *indistinguishable* to a set $\mathcal{T}$ of transactions, if for each transaction $T_k \in \mathcal{T}$, $E|k = E'|k$. A TM *history* is the subsequence of an execution consisting of the invocation and response events of t-operations. Two histories H and H' are *equivalent* if $txns(H) = txns(H')$ and for every transaction $T_k \in txns(H)$, $H|k = H'|k$.

The *read set* (resp., the *write set*) of a transaction T_k in an execution E, denoted $Rset_E(T_k)$ (and resp. $Wset_E(T_k)$), is the set of t-objects that T_k attempts to read (and resp. write) by issuing a t-read (and resp. t-write) invocation in E (for brevity, we sometimes omit the subscript E from the notation). The *data set* of T_k is $Dset(T_k) = Rset(T_k) \cup Wset(T_k)$. T_k is called *read-only* if $Wset(T_k) = \emptyset$; *write-only* if $Rset(T_k) = \emptyset$ and *updating* if $Wset(T_k) \neq \emptyset$. Note that we consider the conventional dynamic TM model: the data set of a transaction is identifiable only by the set of t-objects the transaction has invoked a read or write in the given execution.

Orders on transactions. Let $txns(E)$ 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*, *i.e.*, 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 $T_k \in txns(E)$ is *complete in* E if $E|k$ ends with a response event. The execution E is *complete* if all transactions in $txns(E)$ are complete in E. A transaction $T_k \in txns(E)$ 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 $txns(E)$ are t-complete.

For transactions $\{T_k, T_m\} \in txns(E)$, 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. 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 (resp., nontrivial) if it is the application of a trivial (resp., nontrivial) primitive).

We say that 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 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.

TM-correctness. Informally, a t-sequential history S is *legal* if every t-read of a t-object returns the *latest written value* of this t-object in S. A history H is *opaque* if there exists a legal t-sequential history S equivalent to H such that S respects the real-time order of transactions in H [15]. A weaker condition called *strict serializability* ensures opacity only with respect to committed transactions.

TM-liveness. We say that a TM implementation M provides *obstruction-free (OF) TM-liveness* if for every finite execution E of M, 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. Progress for TMs specifies the conditions under which a transaction is allowed to abort. We say that a TM implementation M provides *obstruction-free (OF) TM-progress* if for every execution E of M, if any transaction $T_k \in \mathit{txns}(E)$ returns A_k in E, then E is not step contention-free for T_k.

We say that transactions T_i, T_j *conflict* in an execution E on a t-object X if T_i and T_j are concurrent in E and $X \in \mathit{Dset}(T_i) \cap \mathit{Dset}(T_j)$, and $X \in \mathit{Wset}(T_i) \cup \mathit{Wset}(T_j)$. A TM implementation M provides *progressive* TM-progress (or *progressiveness*) if for every execution E of M and every transaction $T_i \in \mathit{txns}(E)$ that returns A_i in E, there exists prefix E' of E and a transaction $T_k \in \mathit{txns}(E')$ such that T_k and T_i conflict in E.

Read invisibility. Informally, in a TM using invisible reads, a transaction cannot reveal any information about its read set to other transactions. Thus, given an execution E and some transaction T_k with a non-empty read set, transactions other than T_k cannot distinguish E from an execution in which T_k's read set is empty. This prevents TMs from applying nontrivial primitives during t-read operations and from announcing read sets of transactions during tryCommit. Most popular TM implementations like *TL2* [6] and *NOrec* [5] satisfy this property (the formal definition can be found in the technical report [22]).

Disjoint-access parallelism (DAP). A TM implementation M is *strictly disjoint-access parallel (strict DAP)* if, for all executions E of M, and for all transactions T_i and T_j that participate in E, T_i and T_j contend on a base object in E only if $\mathit{Dset}(T_i) \cap \mathit{Dset}(T_j) \neq \emptyset$ [15].

We now describe two relaxations of strict DAP. For the definitions, we introduce the notion of a *conflict graph* which captures the dependency relation among t-objects accessed by transactions.

We denote by $\tau_E(T_i, T_j)$, the set of transactions (T_i and T_j included) that are concurrent to at least one of T_i and T_j in an execution E.

Let $G(T_i, T_j, E)$ be an undirected graph whose vertex set is $\bigcup_{T \in \tau_E(T_i, T_j)} \mathit{Dset}(T)$ and there is an edge between t-objects X and Y *iff* there exists $T \in \tau_E(T_i, T_j)$ such that $\{X, Y\} \in \mathit{Dset}(T)$. We say that T_i and T_j are *disjoint-access* in E if there is no path between a t-object in $\mathit{Dset}(T_i)$ and a t-object in $\mathit{Dset}(T_j)$ in $G(T_i, T_j, E)$ [4, 25].

Let $\tilde{G}(T_i, T_j, E)$ be a subgraph of $G(T_i, T_j, E)$ where t-objects X and Y are connected with an edge *iff* there exists $T \in \tau_E(T_i, T_j)$ such that $\{X, Y\} \in Wset(T)$. Respectively, T_i and T_j are *read-write 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 $\tilde{G}(T_i, T_j, E)$.

A TM implementation M is *read-write disjoint-access parallel (RW DAP)* (and resp. weak DAP) if, for all executions E of M, transactions T_i and T_j contend (and resp. concurrently contend) on the same base object in E only if T_i and T_j are not read-write disjoint-access (and resp. disjoint-access) in E or there exists a t-object $X \in Dset(T_i) \cap Dset(T_j)$. The technical report [22] provides further details and examples on the DAP definitions.

3 Lower bounds for obstruction-free TMs

Let $\mathcal{OF}$ denote the class of TMs that provide OF TM-progress and OF TM-liveness. In Section 3.1, we show that no strict serializable TM in $\mathcal{OF}$ can be weak DAP and have invisible reads. In Section 3.2, we determine stall complexity bounds for strict serializable TMs in $\mathcal{OF}$, and in Section 3.3, we present a linear (in n) lower bound on the RAW/AWAR complexity for RW DAP opaque TMs in $\mathcal{OF}$.

3.1 Impossibility of invisible reads

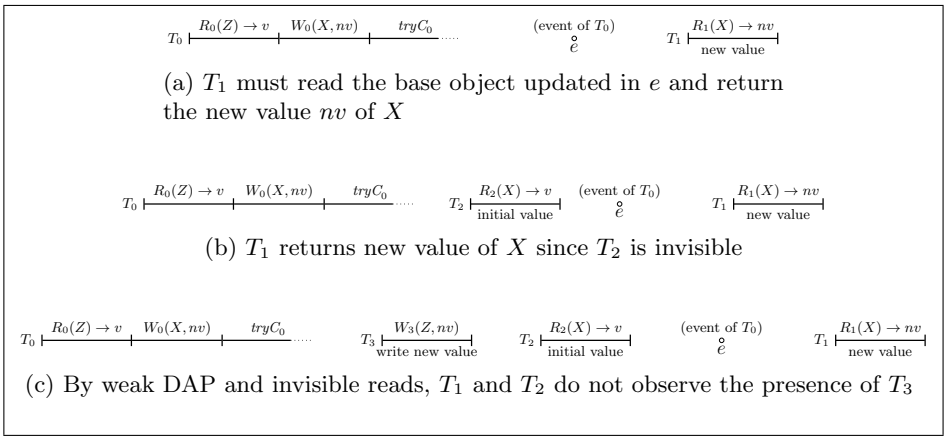
In this section, we prove that it is impossible to derive TM implementations in $\mathcal{OF}$ that combine weak DAP and invisible reads. The formal proof is given in the technical report [22], we present an intuition below.

Theorem 1. *There does not exist a weak DAP strictly serializable TM implementation in $\mathcal{OF}$ that uses invisible reads.*

Proof (Outline). Suppose, by contradiction, that such a TM implementation M exists. Consider an execution E of M in which a transaction T_0 performs a t-read of t-object Z (returning the initial value v), writes nv (new value) to t-object X, and commits. Let E' denote the longest prefix of E that cannot be extended with the t-complete step contention-free execution of any transaction that reads nv in X and commits.

Thus if T_0 takes one more step after E', then the resulting execution $E' \cdot e$ can be extended with the t-complete step contention-free execution of a transaction T_1 that reads nv in X and commits (Figure 2a).

Since M uses invisible reads, the following execution exists: E' can be extended with the t-complete step contention-free execution of a transaction T_2 that reads the initial value v in X and commits, followed by the step e of T_0 after which transaction T_1 running step contention-free reads nv in X and commits (Figure 2b). Moreover, this execution is indistinguishable to T_1 and T_2 from an execution in which the read set of T_0 is empty. Thus, we can modify this execution by inserting the step contention-free execution of a committed



(a) T_1 must read the base object updated in e and return the new value nv of X

(b) T_1 returns new value of X since T_2 is invisible

(c) By weak DAP and invisible reads, T_1 and T_2 do not observe the presence of T_3

Fig. 2: Executions describing the proof sketch of Theorem 1; execution in 2c is not strictly serializable

transaction T_3 that writes a new value to Z after E', but preceding T_2 in real-time order. Intuitively, by weak DAP, transactions T_1 and T_2 cannot distinguish this execution from the original one in which T_3 does not participate.

Thus, we can show that the following execution exists: E' is extended with the t-complete step contention-free execution of T_3 that writes nv to Z and commits, followed by the t-complete step contention-free execution of T_2 that reads the initial value v in X and commits, followed by the step e of T_0, after which T_1 reads nv in X and commits (Figure 2c).

This execution is, however, not strictly serializable: T_0 must appear in any serialization (T_1 reads a value written by T_0). Transaction T_2 must precede T_0, since the t-read of X by T_2 returns the initial value of X. To respect real-time order, T_3 must precede T_2. Finally, T_0 must precede T_3 since the t-read of Z returns the initial value of Z. The cycle $T_0 \to T_3 \to T_2 \to T_0$ implies a contradiction.

3.2 Stall complexity

We prove a linear (in n) lower bound for strictly serializable TM implementations in $\mathcal{OF}$ on the total number of *memory stalls* incurred by a single t-read operation.

Let $E = \alpha \cdot e_1 \cdots e_m \cdot e \cdot \beta$ be an execution of M, where α and β are execution fragments, e is a primitive applied by a process p on a base object b within a t-operation *op*, and $e_1 \cdots e_m$ is a maximal sequence of $m \geq 1$ consecutive nontrivial events by distinct processes other than p that access b. Then, we say that *op* incurs *m memory stalls in E on account of e*. The *number of memory stalls incurred by op in E* is the sum of memory stalls incurred by all events of *op* in E [1, 9].

Theorem 2. *Every strictly serializable TM implementation $M \in \mathcal{OF}$ has an execution in which some t-read operation incurs $\Omega(n)$ stalls.*

We give an intuitive sketch below, but the full proof can be found in [22]. Inductively, for each $k \leq n-1$, we construct a specific *k-stall execution* [9] in which some t-read operation by a process p incurs k stalls. In the k-stall execution, k processes are partitioned into disjoint subsets $S_1, \ldots, S_i$. The execution can be represented as $\alpha \cdot \sigma_1 \cdots \sigma_i$; α is p-free, where in each σ_j, $j = 1, \ldots, i$, p first runs by itself, then each process in S_j applies a *nontrivial* event on a base object b_j, and then p applies an event on b_j. Moreover, p does not detect step contention in this execution and, thus, must return a non-abort value in its t-read and commit in the solo extension of it. Additionally, it is guaranteed that in any extension of α by the processes other than $\{p\} \cup S_1 \cup S_2 \cup \ldots \cup S_i$, no nontrivial primitive is applied on a base object accessed in $\sigma_1 \cdots \sigma_i$.

Assuming a k-stall execution $\alpha \cdot \sigma_1 \cdots \sigma_i$ for process p executing a t-read operation where $k \leq n-2$, we introduce a not previously used process executing an updating transaction immediately after α, so that the subsequent t-read operation executed by p is "perturbed" (must return another value). This will help us to construct a $(k+k')$-stall execution $\alpha \cdot \alpha' \cdot \sigma_1 \cdots \sigma_i \cdot \sigma_{i+1}$, where $k' > 0$. Thus, the TM has a $(n-1)$-stall execution for some t-read operation.

3.3 RAW/AWAR complexity

In this section, we characterize the complexity of implementations in $\mathcal{OF}$ by measuring the amount of expensive synchronization patterns like *RAW* (read-after-write) or *AWAR* (atomic-write-after-read) that read-only transactions may need to perform.

A RAW pattern performed by a transaction T_k in an execution π is a pair of its events e and e', such that: (1) e is a write to a base object b by T_k, (2) e' is a subsequent read of a base object $b' \neq b$ by T_k, and (3) no event on b by T_k takes place between e and e'. In this paper, we are concerned only with *non-overlapping* RAWs, *i.e.*, the read performed by one RAW precedes the write performed by the other RAW. An AWAR pattern e in an execution $\pi \cdot e$ is a nontrivial rmw event on an object b which atomically returns the value of b (resulting after π) and updates b with a new value, *e.g.*, a successful *compare-and-swap*.

We prove that opaque, RW DAP TM implementations in $\mathcal{OF}$ have executions in which some read-only transaction performs a linear (in n) number of non-overlapping RAWs or AWARs. Our result illustrates why individual t-read operations of RW DAP obstruction-free TMs like DSTM [18] must forcefully abort pending conflicting transactions using compare-and-swap in some executions.

Theorem 3. *Every RW DAP opaque TM implementation $M \in \mathcal{OF}$ has an execution E in which some read-only transaction $T \in txns(E)$ performs $\Omega(n)$ non-overlapping RAW/AWARs.*

Proof (Outline). We first construct an execution of the form $\bar{\rho}_1 \cdots \bar{\rho}_m$, where for all $j \in \{1, \ldots, m\}$; $m = n-3$, $\bar{\rho}_j$ denotes the t-complete step contention-free execution of transaction T_j that reads the initial value v in a distinct t-object

Z_j, writes a new value nv to a distinct t-object X_j and commits. Observe that since any two transactions that participate in this execution are mutually read-write disjoint-access, they cannot contend on the same base object and, thus, the execution appears solo to each of them.

Let each of two new transactions T_{n-1} and T_n perform m t-reads on objects $X_1, \ldots, X_m$. For $j \in \{1, \ldots, m\}$, we now define ρ_j to be the longest prefix of $\bar{\rho}_j$ such that $\rho_1 \cdots \rho_j$ cannot be extended the complete step contention-free execution fragment of T_{n-1} or T_n where the t-read of X_j returns nv. Let e_j be the event by T_j enabled after $\rho_1 \cdots \rho_j$. Let us count the number of indices $j \in \{1, \ldots, m\}$ such that T_{n-1} (resp., T_n) reads the new value nv in X_j when it runs after $\rho_1 \cdots \rho_j \cdot e_j$. Without loss of generality, assume that T_{n-1} has more such indices j than T_n. We are going to show that, in the worst-case, T_n must perform $\lceil \frac{m}{2} \rceil$ non-overlapping RAW/AWARs in the course of performing m t-reads of $X_1, \ldots, X_m$ immediately after $\rho_1 \cdots \rho_m$.

Consider any $j \in \{1, \ldots, m\}$ such that T_{n-1}, when it runs step contention-free after $\rho_1 \cdots \rho_j \cdot e_j$, reads nv in X_j. We claim that, in $\rho_1 \cdots \rho_m$ extended with the step contention-free execution of T_n performing j t-reads $\mathit{read}_n(X_1) \cdots \mathit{read}_n(X_j)$, the t-read of X_j must contain a RAW or an AWAR.

Suppose not. Then we are going to schedule a specific execution of T_j and T_{n-1} concurrently with $\mathit{read}_n(X_j)$ so that T_n cannot detect the concurrency. By the definition of ρ_j and the fact that the TM is RW DAP, T_n, when it runs step contention-free after $\rho_1 \cdots \rho_m$, must read v (the initial value) in X_j. Then the following execution exists: $\rho_1 \cdots \rho_m$ is extended with the t-complete step contention-free execution of T_{n-2} writing nv to Z_j and committing, after which T_n runs step contention-free and reads v in X_j. Since, by the assumption, $\mathit{read}_n(X_j)$ contains no RAWs or AWARs, we show that we can run T_{n-1} performing j t-reads concurrently with the execution of $\mathit{read}_n(X_j)$ so that T_n and T_{n-1} are unaware of step contention and $\mathit{read}_{n-1}(X_j)$ still reads the value nv in X_j.

To understand why this is possible, consider the following: we take the execution constructed above, but without the execution of $\mathit{read}_n(X_j)$, *i.e*, $\rho_1 \cdots \rho_m$ is extended with the step contention-free execution of committed transaction T_{n-2} writing nv to Z_j, after which T_n runs step contention-free performing $j-1$ t-reads. This execution can be extended with the step e_j by T_j, followed by the step contention-free execution of transaction T_{n-1} in which it reads nv in X_j. Indeed, by RW DAP and the definition of $\rho_j \cdot e_j$, there exists such an execution.

Since $\mathit{read}_n(X_j)$ contains no RAWs or AWARs, we can reschedule the execution fragment e_j followed by the execution of T_{n-1} so that it is concurrent with the execution of $\mathit{read}_n(X_j)$ and neither T_n nor T_{n-1} see a difference. Therefore, in this execution, $\mathit{read}_n(X_j)$ still returns v, while $\mathit{read}_{n-1}(X_j)$ returns nv.

However, the resulting execution is not opaque. In any serialization the following must hold. Since T_{n-1} reads the value written by T_j in X_j, T_j must be committed. Since $\mathit{read}_n(X_j)$ returns the initial value v, T_n must precede T_j. The committed transaction T_{n-2}, which writes a new value to Z_j, must precede T_n to respect the real-time order on transactions. However, T_j must precede T_{n-2}

since $read_j(Z_j)$ returns the initial value and the implementation is opaque. The cycle $T_j \to T_{n-2} \to T_n \to T_j$ implies a contradiction.

Thus, we can show that transaction T_n must perform $\Omega(n)$ RAW/AWARs during the execution of m t-reads immediately after $\rho_1 \cdots \rho_m$.

4 Upper bound for opaque progressive TMs

In this section, we describe a *progressive*, opaque TM implementation LP (Algorithm 1) that is not subject to any of the lower bounds we derived so far for $\mathcal{OF}$ (cf. Figure 1). In our TM LP, every transaction performs at most a single RAW, every t-read operation incurs $O(1)$ memory stalls and maintains exactly one version of every t-object in every execution. Moreover, the implementation is strict DAP and uses only read-write base objects.

Base objects. For every t-object X_j, LP maintains a base object v_j that stores the *value* of X_j. Additionally, for each X_j, we maintain a bit L_j, which if set, indicates the presence of an updating transaction writing to X_j. Also, for every process p_i and t-object X_j, LP maintains a *single-writer bit* r_{ij} to which only p_i is allowed to write. Each of these base objects may be accessed only via read and write primitives.

Read operations. The implementation first reads the value of t-object X_j from base object v_j and then reads the bit L_j to detect contention with an updating transaction. If L_j is set, the transaction is aborted; if not, read validation is performed on the entire read set. If the validation fails, the transaction is aborted. Otherwise, the implementation returns the value of X_j. For a read-only transaction T_k, $tryC_k$ simply returns the commit response.

Updating transactions. The $write_k(X, v)$ implementation by process p_i simply stores the value v locally, deferring the actual updates to $tryC_k$. During $tryC_k$, process p_i attempts to obtain exclusive write access to every $X_j \in Wset(T_k)$. This is realized through the single-writer bits, which ensure that no other transaction may write to base objects v_j and L_j until T_k relinquishes its exclusive write access to $Wset(T_k)$. Specifically, process p_i writes 1 to each r_{ij}, then checks that no other process p_t has written 1 to any r_{tj} by executing a series of reads (incurring a single RAW). If there exists such a process that concurrently contends on write set of T_k, for each $X_j \in Wset(T_k)$, p_i writes 0 to r_{ij} and aborts T_k. If successful in obtaining exclusive write access to $Wset(T_k)$, p_i sets the bit L_j for each X_j in its write set. Implementation of $tryC_k$ now checks if any t-object in its read set is concurrently contended by another transaction and then validates its read set. If there is contention on the read set or validation fails (indicating the presence of a conflicting transaction), the transaction is aborted. If not, p_i writes the values of the t-objects to shared memory and relinquishes exclusive write access to each $X_j \in Wset(T_k)$ by writing 0 to each of the base objects L_j and r_{ij}.

Complexity. Read-only transactions do not apply any nontrivial primitives. Any updating transaction performs at most a single RAW in the course of acquiring exclusive write access to the transaction's write set. Thus, every transaction

Algorithm 1 Strict DAP progressive opaque TM implementation LP; code for T_k executed by process p_i

1: **Shared base objects:**
2: v_j, for each t-object X_j
allows reads and writes
3: r_{ij}, for every p_i and t-object X_j
single-writer bit
allows reads and writes
4: L_j, for every t-object X_j
allows reads and writes

5: **Local variables:**
6: $Rset_k$, $Wset_k$ for every T_k;
dictionaries storing $\{X_m, v_m\}$

7: $read_k(X_j)$:
8: **if** $X_j \notin Rset(T_k)$ **then**
9: $[ov_j, k_j] := read(v_j)$
10: $Rset(T_k)$.add($\{X_j, [ov_j, k_j]\}$)
11: **if** $read(L_j) \neq 0$ **then**
12: **Return** A_k
13: **if** validate() **then**
14: **Return** A_k
15: **Return** ov_j
16: **else**
17: $[ov_j, \perp] := Rset(T_k)$.locate($X_j$)
18: **Return** ov_j

19: $write_k(X_j, v)$:
20: $nv_j := v$
21: $Wset(T_k)$.add($\{X_j\}$)
22: **Return** ok

23: $tryC_k()$:
24: **if** $|Wset(T_k)| = \emptyset$ **then**
25: **Return** C_k
26: locked := acquire($Wset(T_k)$)
27: **if** $\neg$ locked **then**
28: **Return** A_k
29: **if** isAbortable() **then**
30: release($Wset(T_k)$)
31: **Return** A_k
// Exclusive write access to each v_j
32: **for all** $X_j \in Wset(T_k)$ **do**
33: $write(v_j, [nv_j, k])$
34: release($Wset(T_k)$)
35: **Return** C_k

36: **Function:** release(Q):
37: **for all** $X_j \in Q$ **do**
38: $write(L_j, 0)$
39: **for all** $X_j \in Q$ **do**
40: $write(r_{ij}, 0)$
41: **Return** ok

42: **Function:** acquire(Q):
43: **for all** $X_j \in Q$ **do**
44: $write(r_{ij}, 1)$
45: **if** $\exists X_j \in Q; t \neq i : read(r_{tj}) = 1$ **then**
46: **for all** $X_j \in Q$ **do**
47: $write(r_{ij}, 0)$
48: **Return** *false*
// Exclusive write access to each L_j
49: **for all** $X_j \in Q$ **do**
50: $write(L_j, 1)$
51: **Return** *true*

52: **Function:** isAbortable() :
53: **if** $\exists X_j \in Rset(T_k) : X_j \notin Wset(T_k) \wedge read(L_j) \neq 0$ **then**
54: **Return** *true*
55: **if** validate() **then**
56: **Return** *true*
57: **Return** *false*

58: **Function:** validate() :
// Read validation
59: **if** $\exists X_j \in Rset(T_k)$:$[ov_j, k_j] \neq read(v_j)$ **then**
60: **Return** *true*
61: **Return** *false*

performs $O(1)$ non-overlapping RAWs in any execution. However, just as state-of-the-art progressive opaque TM implementations like TL [7] and NOrec [5] that use invisible reads, LP must incur the inherent incremental validation cost that is linear in the size of the read set [15, 21].

Recall that a transaction may write to base objects v_j and L_j only after obtaining exclusive write access to t-object X_j, which in turn is realized via single-writer base objects. Thus, no transaction performs a write to any base object b immediately after a write to b by another transaction, *i.e.*, every transaction incurs only $O(1)$ memory stalls on account of any event it performs. The $read_k(X_j)$ implementation reads base objects v_j and L_j, followed by the validation phase in which it reads v_k for each X_k in its current read set. Note that if the first read in the validation phase incurs a stall, then $read_k(X_j)$ aborts. It follows that each t-read incurs $O(1)$ stalls in every execution.

Thus, we can prove the following theorem:

Theorem 4. *Algorithm 1 describes a progressive, opaque and strict DAP TM implementation LP that provides wait-free TM-liveness, uses invisible reads, uses only read-write base objects, and for every execution E and transaction $T_k \in txns(E)$: (i) T_k performs at most a single RAW, and (ii) every t-read operation performed by T_k incurs $O(1)$ memory stalls in E.*

5 Related work

Attiya *et al.* [4] were the first to formally define DAP for TMs. They proved the impossibility of implementing weak DAP strictly serializable TMs that use invisible reads and guarantee that read-only transactions eventually commit, while updating transactions are guaranteed to commit only when they run sequentially [4]. This class is orthogonal to the class of obstruction-free TMs, as is the proof technique used to establish the impossibility arguments (Section 3.1).

Perelman *et al.* [25] showed that *mv-permissive* weak DAP TMs cannot be implemented. In mv-permissive TMs, only updating transactions may be aborted, and only when they conflict with other updating transactions. In particular, read-only transactions cannot be aborted and updating transactions may sometimes be aborted even in the absence of step contention, which makes the impossibility result in [25] unrelated to ours (Section 3.1).

Guerraoui and Kapalka [15] proved that it is impossible to implement strict DAP obstruction-free TMs. They also proved that a strict serializable TM that provides OF TM-progress and wait-free TM-liveness cannot be implemented using only read and write primitives. We show in Section 4 that progressive TMs are not subject to either of these lower bounds.

Attiya *et al.* [2] proved that it is impossible to derive RAW/AWAR-free implementations of data types like *stacks*, *queues* and *deadlock-free mutual exclusion*. The metric was previously used in [20] to measure the complexity of read-only transactions in a strictly stronger (than $\mathcal{OF}$) class of *permissive* TMs (assuming wait-free TM-liveness) which ensure that a transaction may be aborted only if committing it would violate opacity. This lower bound in [20] is unrelated to

Theorem 3 on RW DAP obstruction-free TMs. Detailed coverage on memory fences and the RAW/AWAR metric can be found in [24].

To derive the linear lower bound on the memory stall complexity of obstruction-free TMs (Section 3.2), we adopted the definition of a *k-stall execution* and certain proof steps from [1, 9].

Our upper bound LP that theoretically demonstrates the advantages of adapting TMs to data conflicts rather than step contention is inspired by the progressive TM of [20]. Complexity optimizations for progressive TMs like reducing the cost of read-validation by slightly relaxing strict DAP, as achieved in TL2 [6], can also be applied to LP.

The technical report [22] provides details on the DAP definitions as well as opaque implementations in $\mathcal{OF}$ that satisfy weak and RW DAP. The definition of invisible reads used in this paper is adopted from [3].

6 Concluding remarks

As highlighted in [7, 11], obstruction-free TMs require an *indirection* from the t-object *metadata* in order to find the current version of the t-object. This suggests that obstruction-free TMs must forcefully abort pending conflicting transactions in order to return the correct t-object version. This observation inspires the impossibility of invisible reads (Theorem 1). Typically, to detect the presence of a conflicting transaction and abort it, the reading transaction must employ a RAW or read-modify-write primitives like compare-and-swap, motivating the linear lower bound on expensive synchronization (Theorem 3). Also, in obstruction-free TMs, a transaction may not wait for a concurrent inactive transaction to complete and, as a result, we may have an execution in which a transaction incurs a distinct stall due to a transaction run by each other process (Theorem 2). Intuitively, since transactions in progressive TMs may abort themselves in case of conflicts, they can employ invisible reads and maintain constant stall and RAW/AWAR complexities.

Some benefits of obstruction-free TMs, namely their ability to make progress even if some transactions prematurely fail, are not provided by progressive TMs. However, several papers [6, 7, 11] argued that lock-based TMs tend to outperform obstruction-free ones by allowing for simpler algorithms with lower overhead, and their inherent progress issues may be resolved using timeouts and contention-managers. This paper explains the empirically observed performance gap between blocking and non-blocking TMs via a series of lower bounds on obstruction-free TMs and a progressive TM algorithm that beats all of them.

References

1. H. Attiya, R. Guerraoui, D. Hendler, and P. Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4), 2009.
2. H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

3. H. Attiya and E. Hillel. The cost of privatization in software transactional memory. *IEEE Trans. Computers*, 62(12):2531–2543, 2013.
4. H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.
5. L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPOPP*, pages 67–78, 2010.
6. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.
7. D. Dice and N. Shavit. What really makes transactions fast? In *Transact*, 2006.
8. D. Dice and N. Shavit. TLRW: return of the read-write lock. In *SPAA*, pages 284–293, 2010.
9. F. Ellen, D. Hendler, and N. Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.
10. R. Ennals. The lightweight transaction library. http://sourceforge.net/projects/libltx/files/.
11. R. Ennals. Software transactional memory should not be obstruction-free. 2005.
12. K. Fraser. Practical lock-freedom. Technical report, Cambridge University Computer Laborotory, 2003.
13. R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 304–313, New York, NY, USA, 2008. ACM.
14. R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *POPL*, pages 404–415, 2009.
15. R. Guerraoui and M. Kapalka. *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.
16. M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.
17. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.
18. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.
19. M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.
20. P. Kuznetsov and S. Ravi. On the cost of concurrency in transactional memory. In *OPODIS*, pages 112–127, 2011. full version: http://arxiv.org/abs/1103.1302.
21. P. Kuznetsov and S. Ravi. Progressive transactional memory in time and space. *CoRR*, abs/1502.04908, 2015. To appear in 13th International Conference on Parallel Computing Technologies August 31 - September 4, 2015, Petrozavodsk, Russia.
22. P. Kuznetsov and S. Ravi. Why transactional memory should not be obstruction-free. *CoRR*, abs/1502.02725, 2015. http://arxiv.org/abs/1502.02725.
23. V. J. Marathe, W. N. S. Iii, and M. L. Scott. Adaptive software transactional memory. In *Proc. of the 19th International Symposium on Distributed Computing*, pages 354–368, 2005.
24. P. E. McKenney. Memory barriers: a hardware view for software hackers. Linux Technology Center, IBM Beaverton, June 2010.
25. D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC*, pages 16–25, 2010.
26. F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. Nztm: Nonblocking zero-indirection transactional memory. SPAA '09, pages 204–213, New York, NY, USA, 2009. ACM.