Why Transactional Memory Should Not Be Obstruction-Free

Petr Kuznetsov\textsuperscript{1} Srivatsan Ravi\textsuperscript{2}
\textsuperscript{1}Télécom ParisTech
\textsuperscript{2}TU Berlin

Abstract

Transactional memory (TM) is an inherently optimistic 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 atomic transactions. A transaction may commit, in which case its updates of data items “take effect” or it may abort, in which case no data items are updated. A TM implementation provides processes with algorithms for implementing transactional operations on data items (such as read, write and tryCommit) by applying primitives on shared base objects. Intuitively, the idea behind the TM abstraction is optimism: before a transaction commits, all its operations are speculative, and it is expected that, in the absence of concurrency, a transaction commits.

It therefore appears natural that early TMs implementations \cite{LockFreeTMs} adopted optimistic concurrency control and guaranteed that a prematurely halted transaction cannot not prevent other transactions from committing. These implementations avoided using locks and relied on non-blocking (sometimes also called lock-free) synchronization. Possibly the weakest non-blocking progress condition is obstruction-freedom \cite{obstruction-freedom} stipulating that every transaction running in the absence of step contention, i.e., not encountering steps of concurrent transactions, must commit.

In 2005, Ennals \cite{Ennals2005} argued that that obstruction-free TMs inherently yield poor performance, because they require transactions to forcefully abort each other. Ennals further describes a lock-based TM implementation \cite{Ennals2005} that he claimed to outperform DSTM \cite{DSTM}, the most referenced obstruction-free TM implementation at the time. Inspired by \cite{Ennals2005}, more recent TM implementations like TL \cite{TL}, TL2 \cite{TL2} and NOrec \cite{NOrec} employ locking and showed that Ennal’s claims about performance of lock-based TMs hold true on most workloads. The progress guarantee provided by these TMs is typically progressiveness: a transaction may be aborted only if it encounters a read-write or a write-write conflicts with a concurrent transaction \cite{progressiveness}.

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 two important TM properties: weak disjoint-access-parallelism (weak DAP) and read invisibility. Weak DAP \cite{5} is believed to improve TM performance by ensuring that transactions concurrently contend on the same base object (both access the base object and at least one updates it) only if their data sets are connected in the conflict graph constructed on the data sets of concurrent transactions \cite{5}. Many popular obstruction-free TM implementations satisfy weak DAP \cite{13,20,30}, but not the stronger property of strict DAP \cite{14,17} that disallows any two transactions to contend on a base object unless they access a common data item.

A TM implementation uses invisible reads if, informally, a reading transaction cannot cause a concurrent transaction to abort (we give a more precise definition later in this paper), which is believed to be important for (most commonly observed) read-dominated workloads. Interestingly, lock-based TM implementations like TL \cite{8} are weak DAP and use invisible reads. In contrast, we establish that it is impossible to implement a strictly serializable (all committed transactions appear to execute sequentially in some total-order respecting the timing of non-overlapping transactions) obstruction-free TM that provides both weak DAP and read invisibility. Indeed, obstructions TMs like DSTM \cite{20} and FSTM \cite{13} satisfy weak DAP, but not read invisibility since read operations must write to the shared memory.

We then derive lower bounds on obstruction-free TM implementations with respect to the number of stalls \cite{10}. The stall complexity 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 concurrently update the object \cite{10}. Our second result shows that a single read operation in a \( n \)-process strictly serializable obstruction-free TM implementation may incur \( \Omega(n) \) stalls.

Finally, we prove that any read-write (RW) DAP opaque (all transactions appear to execute sequentially in some total-order respecting the timing of non-overlapping transactions) 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 \cite{3} capture the amount of “expensive synchronization”, i.e., the number of costly conditional primitives or memory barriers \cite{1} 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, satisfied by most obstruction-free implementations \cite{13,20}, requires that read-only transactions do not contend on the same base object with transactions having disjoint write sets. It is stronger than weak DAP \cite{5}, but weaker than strict DAP \cite{15}.

| strict DAP | Obstruction-free TMs | Our progressive TM LP |
|------------|----------------------|-----------------------|
| invisible reads+weak DAP | No \cite{15} | Yes |
| stall complexity of reads | \( \Omega(n) \) | \( O(1) \) |
| RAW/AWAR complexity | \( \Omega(n) \) | \( O(1) \) |
| read-write primitives, wait-free termination | No \cite{17} | Yes |

Figure 1: Complexity gap between blocking and non-blocking strictly serializable TM implementations; \( n \) is the number of processes

An upper bound for blocking TMs. To exhibit a complexity gap between blocking and non-blocking TMs, we describe a progressive opaque TM implementation that beats the impossibility result and the lower bounds we established for obstruction-free TMs.

Our implementation, denoted \( LP \), (1) uses only read and write primitives on base objects and ensures that every transactional operation terminates in a wait-free manner, (2) ensures strict DAP, (3) has invisible reads, (4) performs \( O(1) \) non-overlapping RAWs/AWARs per transaction, and (5) incurs \( O(1) \) memory stalls for read operations. In contrast, the following claims hold for any implementation in the class of obstruction-free (OF) strict serializable TMs: No OF TM can
be implemented (i) using only read and write primitives and provide wait-free termination [17], or (ii) provide strict DAP [15]. Furthermore, (iii) no weak DAP OF TM has invisible reads (Theorem 2) and (iv) no OP TM ensures a constant number of stalls incurred by a read operation (Theorem 5). Finally, (v) no RW DAP opaque OFTM has constant RAW/AWAR complexity (Theorem 6). In fact, (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 in Figure 4. 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.

**Roadmap.** Sections 2 and 3 define our model and the classes of TMs considered in this paper. Section 4 contains lower bounds for obstruction-free TMs. Section 5 describes our lock-based TM implementation LP. In Section 6 we discuss the related work and in Section 7 concluding remarks. Some proofs are delegated to the optional appendix.

## 2 Model

**TM interface.** *Transactional memory* (in short, *TM*) allows a set of data items (called *t-objects*) to be accessed via atomic *transactions*. Every transaction $T_k$ has a unique identifier $k$. We make no assumptions on the size of a t-object, i.e., the cardinality on the set $V$ of possible values a t-object can store. A transaction $T_k$ may contain the following t-operations, each being a matching pair of an *invocation* and a *response*: $\text{read}_k(X)$ returns a value in $V$ or a special value $A_k \notin V$ (abort); $\text{write}_k(X,v)$, for a value $v \in V$, returns ok or $A_k$; $\text{tryC}_k$ 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 the previous transaction has committed or aborted. A TM *implementation* provides processes with algorithms for implementing $\text{read}_k$, $\text{write}_k$ and $\text{tryC}_k$ of a transaction $T_k$ by applying primitives from a set of shared base objects, each of which is assigned an *initial value*. We assume that these primitives are *deterministic*. A primitive is a generic *read-modify-write (RMW)* procedure applied to a base object $b$ [10],[15]. It is characterized by a pair of functions $(g,h)$: 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 *step* of $T_k$) is an invocation or response of a t-operation performed by $T_k$ or a RMW primitive $(g,h)$ applied by $T_k$ to a base object $b$ along with its response $r$ (we call it a *RMW event* and write $(b,(g,h),r,k)$).

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 $(b,(g,h),r,k)$ matches the state of $b$ resulting from all preceding events. An execution $E \cdot E'$ denotes the concatenation of $E$ and execution fragment $E'$, and we say that $E'$ is an *extension* of $E$ or $E'$ *extends* $E$.

Let $E$ be an execution fragment. For every transaction (resp., process) identifier $k$, $E|k$ denotes the subsequence of $E$ restricted to events of transaction $T_k$ (resp., process $p_k$). 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 sequence of an execution consisting of the invocation
and response events of t-operations. Two histories \( H, H' \) are equivalent if \( \text{txns}(H) = \text{txns}(H') \) and for every transaction \( T_k \in \text{txns}(H), H|k = H'|k. \)

The read set (resp., the write set) of a transaction \( T_k \) in an execution \( E \), denoted \( \text{Rset}(T_k) \) (resp., \( \text{Wset}(T_k) \)), is the set of t-objects that \( T_k \) reads (resp., writes to) in \( E \). More specifically, if \( E \) contains an invocation of \( \text{read}_k(X) \) (resp., \( \text{write}_k(X, v) \)), we say that \( X \in \text{Rset}(T_k) \) (resp., \( \text{Wset}(T_k) \)). The data set of \( T_k \) is \( \text{Dset}(T_k) = \text{Rset}(T_k) \cup \text{Wset}(T_k) \). A transaction is called read-only if \( \text{Wset}(T_k) = \emptyset \); write-only if \( \text{Rset}(T_k) = \emptyset \) and updating if \( \text{Wset}(T_k) \neq \emptyset \). Note that we consider the conventional dynamic TM programming model: the data set of a transaction is not known apriori (i.e., at the start of the transaction) and it is identifiable only by the set of t-objects the transaction has invoked a read or write in the given execution.

**Transaction orders.** Let \( \text{txns}(E) \) 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 \( T_k \in \text{txns}(E) \) is complete in \( E \) if \( E|k \) ends with a response event. The execution \( E \) is complete if all transactions in \( \text{txns}(E) \) are complete in \( E \). A transaction \( T_k \in \text{txns}(E) \) 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 \( \text{txns}(E) \) are t-complete.

For transactions \( \{T_k, T_m\} \in \text{txns}(E) \), we say that \( T_k \) precedes \( T_m \) in the real-time order of \( E \), denoted \( T_k \prec^RT T_m \), 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 \( T_k \prec^RT T_m \) nor \( T_m \prec^RT T_k \), then \( T_k \) and \( T_m \) are concurrent in \( E \). An execution \( E \) is t-sequential if there are no concurrent transactions in \( E \). We say that \( \text{read}_k(X) \) is legal in a t-sequential execution \( E \) if it returns the latest written value of \( X \) in \( E \), and \( E \) is legal if every \( \text{read}_k(X) \) in \( E \) that does not return \( A_k \) is legal in \( E \).

**Contestion.** We say that a configuration \( C \) after an execution \( E \) is quiescent (resp., t-quiescent) if every transaction \( T_k \in \text{txns}(E) \) is complete (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 (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 \( \text{op}_k \) if the events of \( E|\text{op}_k \) 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

In this section, we define the properties of TM implementations considered in this paper.

**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 \( H \) [17]. A weaker condition called strict serializability ensures opacity only with respect to committed transactions. Precise definitions can be found in Appendix A.

**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 \( \text{op}_k \) immediately after \( E \), the finite step contention-free extension for \( \text{op}_k \) 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 abstraction-free (OF) TM-progress if for every execution \( E \) of \( M \), if any transaction \( T_k \in txns(E) \) 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 \( X \in Dset(T_i) \cap Dset(T_j) \), 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 prefix \( E' \) of \( E \) and a transaction \( T_k \in txns(E') \) such that \( T_k \) and \( T_i \) conflict in \( E \).

**Read invisibility.** Informally, the invisible reads assumption prevents TM implementations from applying nontrivial primitives during t-read operations and from announcing read sets of transactions during tryCommit.

We say that a TM implementation \( M \) uses invisible reads if for every execution \( E \) of \( M \),

- for every read-only transaction \( T_k \in txns(E) \), no event of \( E \setminus k \) is nontrivial in \( E \),
- for every updating transaction \( T_k \in txns(E) \), \( Rset(T_k) \neq \emptyset \), there exists an execution \( E' \) of \( M \) such that
  - \( Rset(T_k) = \emptyset \) in \( E' \)
  - \( txns(E) = txns(E') \) and \( \forall T_m \in txns(E) \setminus \{T_k\} \colon E|m = E'|m \)
  - for any two step contention-free transactions \( T_i, T_j \in txns(E) \), if the last event of \( T_i \) precedes the first event of \( T_j \) in \( E \), then the last event of \( T_i \) precedes the first event of \( T_j \) in \( E' \).

Most popular TM implementations like TL2 [7] and NOrec [6] satisfy this definition of invisible reads.

**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 \( Dset(T_i) \cap Dset(T_j) \neq \emptyset \) [17].

We now describe two relaxations of strict DAP. For the formal 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 \( \bigcup_{T \in \tau_E(T_i, T_j)} Dset(T) \) 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 \( X \in Dset(T_i) \cap Dset(T_j) \) [27].

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 \( \{X, Y\} \in Wset(T) \). We say that \( T_i \) 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 \( G(T_i, T_j, E) \). A TM implementation \( M \) is read-write disjoint-access parallel (RW DAP) if, for all executions \( E \) of \( M \), transactions \( T_i \) and \( T_j \) contend on the same base object in \( E \) only if \( T_i \) and \( T_j \) are not read-write disjoint-access in \( E \) or there exists a t-object \( X \in Dset(T_i) \cap Dset(T_j) \).

We make the following observations about the DAP definitions presented in this paper.

- From the definitions, it is immediate that every RW DAP TM implementation satisfies weak DAP. But the converse is not true. Consider the following execution \( E \) of a weak DAP TM implementaton \( M \) that begins with the t-incomplete execution of a transaction \( T_0 \) that reads \( X \) and writes to \( Y \), followed by the step contention-free executions of two transactions \( T_i \)
Figure 2: Executions in the proof of Theorem 2; execution in 2d is not strictly serializable

and $T_2$ which write to $X$ and read $Y$ respectively. Transactions $T_1$ and $T_2$ may contend on a base object since there is a path between $X$ and $Y$ in $G(T_1, T_2, E)$. However, a RW DAP TM implementation would preclude transactions $T_1$ and $T_2$ from contending on the same base object: there is no edge between t-objects $X$ and $Y$ in the corresponding conflict graph $\tilde{G}(T_1, T_2, E)$ because $X$ and $Y$ are not contained in the write set of $T_0$. Algorithm 3 in Appendix B.2 describes a TM implementation that satisfies weak DAP, but not RW DAP.

• From the definitions, it is immediate that every strict DAP TM implementation satisfies RW DAP. But the converse is not true. To understand why, consider the following execution $E$ of a RW DAP TM implementaton that begins with the t-incomplete execution of a transaction $T_0$ that accesses t-objects $X$ and $Y$, followed by the step contention-free executions of two transactions $T_1$ and $T_2$ which access $X$ and $Y$ respectively. Transactions $T_1$ and $T_2$ may contend on a base object since there is a path between $X$ and $Y$ in $\tilde{G}(T_1, T_2, E)$. However, a strict DAP TM implementation would preclude transactions $T_1$ and $T_2$ from contending on the same base object since $Dset(T_1) \cap Dset(T_2) = \emptyset$ in $E$. Algorithm 2 in Appendix B.1 describes a TM implementation that satisfies RW DAP, but not strict DAP.

4 Lower bounds for obstruction-free TMs

Let $OF$ denote the class of TMs that provide OF TM-progress and OF TM-liveness. In Section 4.1 we show that no strict serializable TM in $OF$ can be weak DAP and have invisible reads. In Section 4.2 we determine stall complexity bounds for strict serializable TMs in $OF$, and in Section 4.3 we present a linear (in $n$) lower bound on RAW/AWARs for RW DAP opaque TMs in $OF$.

4.1 Impossibility of invisible reads

In this section, we prove that it is impossible to derive TM implementations in $OF$ that combine weak DAP and invisible reads. The following lemma will be useful in proving our result.

Lemma 1. ([5], [24]) Let $M$ be any weak DAP TM implementation. Let $\alpha \cdot \rho_1 \cdot \rho_2$ be any execution of $M$ where $\rho_1$ (resp., $\rho_2$) is the step contention-free execution fragment of transaction $T_1 \not\in \text{txns}(\alpha)$ (resp., $T_2 \not\in \text{txns}(\alpha)$) 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 $\alpha \cdot \rho_1 \cdot \rho_2$. 


Theorem 2. There does not exist a weak DAP strictly serializable TM implementation in OF that uses invisible reads.

Proof. By contradiction, assume that such an implementation $M \in OF$ exists. Let $v$ be the initial value of t-objects $X$ and $Z$. Consider an execution $E$ of $M$ in which a transaction $T_0$ performs $read_0(Z) \rightarrow v$ (returning $v$), writes $nv \neq v$ to $X$, and commits. Let $E'$ denote the longest prefix of $E$ that cannot be extended with the t-complete step contention-free execution of transaction $T_1$ that performs a t-read $X$ and returns $nv$ nor with the t-complete step contention-free execution of transaction $T_2$ that performs a t-read of $X$ and returns $nv$.

Let $e$ be the enabled event of transaction $T_0$ in the configuration after $E'$. Without loss of generality, assume that $E' \cdot e$ can be extended with the t-complete step contention-free execution of committed transaction $T_2$ that reads $X$ and returns $nv$. Let $E' \cdot e \cdot E_2$ be such an execution, where $E_2$ is the t-complete step contention-free execution fragment of transaction $T_2$ that performs $read_2(X) \rightarrow nv$ and commits.

We now prove that $M$ has an execution of the form $E' \cdot E_1 \cdot e \cdot E_2$, where $E_1$ is the t-complete step contention-free execution fragment of transaction $T_1$ that performs $read_1(X) \rightarrow v$ and commits.

We observe that $E' \cdot E_1$ is an execution of $M$. Indeed, by OF TM-progress and OF TM-liveness, $T_1$ must return a matching response that is not $A_1$ in $E' \cdot E_1$, and by the definition of $E'$, this response must be the initial value $v$ of $X$.

By the assumption of invisible reads, $E_1$ does not contain any nontrivial events. Consequently, $E' \cdot E_1 \cdot e \cdot E_2$ is indistinguishable to transaction $T_2$ from the execution $E' \cdot e \cdot E_2$. Thus, $E' \cdot E_1 \cdot e \cdot E_2$ is also an execution of $M$ (Figure 2a).

Claim 3. $M$ has an execution of the form $E' \cdot E_1 \cdot E_3 \cdot e \cdot E_2$ where $E_3$ is the t-complete step contention-free execution fragment of transaction $T_3$ that writes $nv \neq v$ to $Z$ and commits.

Proof. The proof is through a sequence of indistinguishability arguments to construct the execution.

We first claim that $M$ has an execution of the form $E' \cdot E_1 \cdot e \cdot E_2 \cdot E_3$. Indeed, by OF TM-progress and OF TM-liveness, $T_3$ must be committed in $E' \cdot E_1 \cdot e \cdot E_2 \cdot E_3$.

Since $M$ uses invisible reads, the execution $E' \cdot E_1 \cdot e \cdot E_2 \cdot E_3$ is indistinguishable to transactions $T_2$ and $T_3$ from the execution $E' \cdot E_2 \cdot E_3$, where $E'$ is the t-incomplete step contention-free execution of transaction $T_0$ with $Wset_{E'}(T_0) = \{X\}$; $Rset_{E'}(T_0) = \emptyset$ that writes $nv$ to $X$.

Observe that the execution $E' \cdot E_1 \cdot e \cdot E_2 \cdot E_3$ is indistinguishable to transactions $T_2$ and $T_3$ from the execution $E' \cdot E_2 \cdot E_3$, in which transactions $T_2$ and $T_3$ are disjoint-access. Consequently, by Lemma 1, $T_2$ and $T_3$ do not contend on any base object in $E' \cdot E_2 \cdot E_3$. Thus, $M$ has an execution of the form $E' \cdot E_1 \cdot e \cdot E_3 \cdot E_2$ (Figure 2b).

By definition of $E'$, $T_0$ applies a nontrivial primitive to some base object, say $b$, in event $e$ that $T_2$ must access in $E_2$. Thus, the execution fragment $E_3$ does not contain any nontrivial event on $b$ in the execution $E' \cdot E_1 \cdot e \cdot E_2 \cdot E_3$. Indeed, since $T_3$ is disjoint-access with $T_0$ in the execution $E' \cdot E_3 \cdot E_2$, by Lemma 1, it cannot access the base object $b$ to which $T_0$ applies a nontrivial primitive in the event $e$. Thus, transaction $T_3$ must perform the same sequence of events $E_3$ immediately after $E'$, implying that $M$ has an execution of the form $E' \cdot E_1 \cdot E_3 \cdot e \cdot E_2$ (Figure 2c).

Finally, we observe that the execution $E' \cdot E_1 \cdot E_3 \cdot e \cdot E_2$ established in Claim 3 is indistinguishable to transactions $T_1$ and $T_3$ from an execution $E' \cdot E_1 \cdot E_3 \cdot e \cdot E_2$, where $Wset(T_0) = \{X\}$ and $Rset(T_0) = \emptyset$ in $E'$. But transactions $T_2$ and $T_3$ are disjoint-access in $E' \cdot E_1 \cdot E_3 \cdot e \cdot E_2$, and by Lemma 1, $T_1$ and $T_3$ do not contend on any base object in this execution. Thus, $M$ has an execution of the form $E' \cdot E_3 \cdot E_1 \cdot e \cdot E_2$ (Figure 2d) in which $T_3$ precedes $T_1$ in real-time order.

However, the execution $E' \cdot E_3 \cdot E_1 \cdot e \cdot E_2$ is not strictly serializable: $T_0$ must be committed in any serialization and transaction $T_1$ must precede $T_0$ since $read_1(X)$ returns the initial value of $X$. To respect real-time order, $T_3$ must precede $T_1$, while $T_0$ must precede $T_2$ since $read_2(X)$ returns $nv$, the value of $X$ updated by $T_0$. Finally, $T_0$ must precede $T_3$ since $read_0(Z)$ returns the initial value of $Z$. But there exists no such serialization—contradiction.

\[ \square \]
4.2 Stall complexity

Let $M$ be any TM implementation. Let $e$ be an event applied by process $p$ to a base object $b$ as it performs a transaction $T$ during an execution $E$ of $M$. Let $E = \alpha \cdot e_1 \cdots e_m \cdot e \cdot \beta$ be an execution of $M$, where $\alpha$ and $\beta$ are execution fragments 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 $T$ incurs $m$ memory stalls in $E$ on account of $e$. The number of memory stalls incurred by $T$ in $E$ is the sum of memory stalls incurred by all events of $T$ in $E$ \cite{2,10}.

In this section, we prove a lower bound of $n - 1$ on the worst case number of stalls incurred by a transaction as it performs a single t-read operation. We adopt the following definition of a $k$-stall execution from \cite{2,10}.

**Definition 1.** An execution $\alpha \cdot \sigma_1 \cdots \sigma_i$ is a $k$-stall execution for t-operation $op$ executed by process $p$ if

- $\alpha$ is $p$-free,
- there are distinct base objects $b_1, \ldots, b_i$ and disjoint sets of processes $S_1, \ldots, S_i$ whose union does not include $p$ and has cardinality $k$ such that, for $j = 1, \ldots, i$,
  - each process in $S_j$ has an enabled nontrivial event about to access base object $b_j$ after $\alpha$, and
  - in $\sigma_j$, $p$ applies events by itself until it is the first about to apply an event to $b_j$, then each of the processes in $S_j$ applies an event that accesses $b_j$, and finally, $p$ applies an event that accesses $b_j$.
- $p$ invokes exactly one t-operation $op$ in the execution fragment $\sigma_1 \cdots \sigma_i$.
- $\sigma_1 \cdots \sigma_i$ contains no events of processes not in $\{p\} \cup S_1 \cup \cdots \cup S_i$.
- in every $(\{p\} \cup S_1 \cup \cdots \cup S_i)$-free execution fragment that extends $\alpha$, no process applies a nontrivial event to any base object accessed in $\sigma_1 \cdots \sigma_i$.

Observe that in a $k$-stall execution $E$ for t-operation $op$, the number of memory stalls incurred by $op$ in $E$ is $k$.

**Lemma 4.** Let $\alpha \cdot \sigma_1 \cdots \sigma_i$ be a $k$-stall execution for t-operation $op$ executed by process $p$. Then, $\alpha \cdot \sigma_1 \cdots \sigma_i$ is indistinguishable to $p$ from a step contention-free execution \cite{2}.

**Theorem 5.** Every strictly serializable TM implementation $M \in OF$ has a $(n-1)$-stall execution $E$ for a t-read operation performed in $E$.

**Proof.** We proceed by induction. Observe that the empty execution is a 0-stall execution since it vacuously satisfies the invariants of Definition 1.

Let $v$ be the initial value of t-objects $X$ and $Z$. Let $\alpha = \alpha_1 \cdots \alpha_{n-2}$ be a step contention-free execution of a strictly serializable TM implementation $M \in OF$, where for all $j \in \{1, \ldots, n-2\}$, $\alpha_j$ is the longest prefix of the execution fragment $\bar{\alpha}_j$ that denotes the t-complete step-contention free execution of committed transaction $T_j$ (invoked by process $p_j$) that performs $read_j(Z) \rightarrow v$, writes value $nv \neq v$ to $X$ in the execution $\alpha_1 \cdots \alpha_{j-1} \cdot \bar{\alpha}_j$ such that

- $tryC_j()$ is incomplete in $\alpha_j$,
- $\alpha_1 \cdots \alpha_j$ cannot be extended with the t-complete step contention-free execution fragment of any transaction $T_{n-1}$ or $T_n$ that performs exactly one t-read of $X$ that returns $nv$ and commits.

Assume, inductively, that $\alpha \cdot \sigma_1 \cdots \sigma_i$ is a $k$-stall execution for $read_n(X)$ executed by process $p_n$, where $0 \leq k \leq n-2$. By Definition 1 there are distinct base objects $b_1, \ldots, b_i$ accessed by disjoint sets of processes $S_1 \cdots S_i$ in the execution fragment $\sigma_1 \cdots \sigma_i$, where $|S_1 \cup \cdots \cup S_i| = k$ and $\sigma_1 \cdots \sigma_i$ contains no events of processes not in $S_1 \cup \cdots \cup S_i \cup \{p_n\}$. We will prove that there exists a $(k+k')$-stall execution for $read_n(X)$, for some $k' \geq 1$.

By Lemma 4 $\alpha \cdot \sigma_1 \cdots \sigma_i$ is indistinguishable to $T_n$ from a step contention-free execution. Let $\sigma$ be the finite step contention-free execution fragment that extends $\alpha \cdot \sigma_1 \cdots \sigma_i$ in which $T_n$ performs
events by itself: completes \( \text{read}_n(X) \) and returns a response. By OF TM-progress and OF TM-

liveness, \( \text{read}_n(X) \) and the subsequent tryC_\alpha \) must each return non-\( A_\alpha \) responses in \( \alpha \cdot \sigma_1 \cdots \sigma_i \cdot \sigma \).

By construction of \( \alpha \) and strict serializability of \( M \), \( \text{read}_n(X) \) must return the response \( v \) or \( n\!v \) in this execution. We prove that there exists an execution fragment \( \gamma \) performed by some process \( p_{n-1} \notin \{p_n\} \cup S_1 \cup \cdots \cup S_i \) extending \( \alpha \) that contains a nontrivial event on some base object that must be accessed by \( \text{read}_n(X) \) in \( \sigma_1 \cdots \sigma_i \cdot \sigma \).

Consider the case that \( \text{read}_n(X) \) returns the response \( n\!v \) in \( \alpha \cdot \sigma_1 \cdots \sigma_i \cdot \sigma \). We define a step contention-free fragment \( \gamma \) extending \( \alpha \) that is the t-complete step contention-free execution of transaction \( T_{n-1} \) executed by some process \( p_{n-1} \notin \{p_n\} \cup S_1 \cup \cdots \cup S_i \) that performs \( \text{read}_{n-1}(X) \to v \), writes \( n\!v \neq v \) to \( Z \) and commits. By definition of \( \alpha \), OF TM-progress and OF TM-liveness, \( M \) has an execution of the form \( \alpha \cdot \gamma \). We claim that the execution fragment \( \gamma \) must contain a nontrivial event on some base object that must be accessed by \( \text{read}_n(X) \) in \( \sigma_1 \cdots \sigma_i \cdot \sigma \). Suppose otherwise. Then, \( \text{read}_n(X) \) must return the response \( n\!v \) in \( \sigma_1 \cdots \sigma_i \cdot \sigma \). But the execution \( \alpha \cdot \sigma_1 \cdots \sigma_i \cdot \sigma \) is not strictly serializable. Since \( \text{read}_n(X) \to n\!v \), there exists a transaction \( T_q \in \text{txns}(\alpha) \) that must be committed and must precede \( T_n \) in any serialization. Transaction \( T_{n-1} \) must precede \( T_n \) in any serialization to respect the real-time order and \( T_{n-1} \) must precede \( T_q \) in any serialization. Also, \( T_q \) must precede \( T_{n-1} \) in any serialization. But there exists no such serialization.

Consider the case that \( \text{read}_n(X) \) returns the response \( v \) in \( \alpha \cdot \sigma_1 \cdots \sigma_i \cdot \sigma \). In this case, we define the step contention-free fragment \( \gamma \) extending \( \alpha \) as the t-complete step contention-free execution of transaction \( T_{n-1} \) executed by some process \( p_{n-1} \notin \{p_n\} \cup S_1 \cup \cdots \cup S_i \) that writes \( n\!v \neq v \) to \( X \) and commits. By definition of \( \alpha \), OF TM-progress and OF TM-liveness, \( M \) has an execution of the form \( \alpha \cdot \gamma \). By strict serializability of \( M \), the execution fragment \( \gamma \) must contain a nontrivial event on some base object that must be accessed by \( \text{read}_n(X) \) in \( \sigma_1 \cdots \sigma_i \cdot \sigma \). Suppose otherwise. Then, \( \sigma_1 \cdots \sigma_i \cdot \gamma \cdot \sigma \) is an execution of \( M \) in which \( \text{read}_n(X) \to v \). But this execution is not strictly serializable: every transaction \( T_q \in \text{txns}(\alpha) \) must be aborted or must be preceded by \( T_n \) in any serialization, but committed transaction \( T_{n-1} \) must precede \( T_n \) in any serialization to respect the real-time ordering of transactions. But then \( \text{read}_n(X) \) must return the new value \( n\!v \) of \( X \) that is updated by \( T_{n-1} \)—contradiction.

Since, by Definition[I] the execution fragment \( \gamma \) executed by some process \( p_{n-1} \notin \{p_n\} \cup S_1 \cup \cdots \cup S_i \) contains no nontrivial events to any base object accessed in \( \sigma_1 \cdots \sigma_i \), it must contain a nontrivial event to some base object \( b_{i+1} \notin \{b_1, \ldots, b_i\} \) that is accessed by \( T_n \) in the execution fragment \( \sigma \).

Let \( \mathcal{A} \) denote the set of all finite \( \{p_n\} \cup S_1 \cup \cdots \cup S_i \)-free execution fragments that extend \( \alpha \). Let \( b_{i+1} \notin \{b_1, \ldots, b_i\} \) be the first base object accessed by \( T_n \) in the execution fragment \( \sigma \) to which some transaction applies a nontrivial event in the execution fragment \( \alpha' \in \mathcal{A} \). Clearly, some such execution \( \alpha \cdot \alpha' \) exists that contains a nontrivial event in \( \alpha' \) to some distinct base object \( b_{i+1} \) not accessed in the execution fragment \( \sigma \). We choose the execution \( \alpha \cdot \alpha' \in \mathcal{A} \) that maximizes the number of transactions that are poised to apply nontrivial events on \( b_{i+1} \) in the configuration after \( \alpha \cdot \alpha' \). Let \( S_{i+1} \) denote the set of processes executing these transactions and \( k' = |S_{i+1}| \) (\( k' > 0 \) as already proved).

We now construct a \( (k + k') \)-stall execution \( \alpha \cdot \alpha' \cdot \sigma_1 \cdots \sigma_i \cdot \sigma_{i+1} \) for \( \text{read}_n(X) \), where in \( \sigma_{i+1} \),\( p_n \) applies events by itself, then each of the processes in \( S_{i+1} \) applies a nontrivial event on \( b_{i+1} \), and finally, \( p_n \) accesses \( b_{i+1} \).

By construction, \( \alpha \cdot \alpha' \) is \( p_n \)-free. Let \( \sigma_{i+1} \) be the prefix of \( \sigma \) not including \( T_n \)'s first access to \( b_{i+1} \), concatenated with the nontrivial events on \( b_{i+1} \) by each of the \( k' \) transactions executed by processes in \( S_{i+1} \) followed by the access of \( b_{i+1} \) by \( T_n \). Observe that \( T_n \) performs exactly one t-operation \( \text{read}_n(X) \) in the execution fragment \( \sigma_1 \cdots \sigma_{i+1} \) and \( \sigma_1 \cdots \sigma_{i+1} \) contains no events of processes not in \( \{p_n\} \cup S_1 \cup \cdots \cup S_i \cup S_{i+1} \).

To complete the induction, we need to show that in every \( \{p_n\} \cup S_1 \cup \cdots \cup S_i \cup S_{i+1} \)-free extension of \( \alpha \cdot \alpha' \), no transaction applies a nontrivial event to any base object accessed in the execution fragment \( \sigma_1 \cdots \sigma_i \cdot \sigma_{i+1} \). Let \( \beta \) be any such execution fragment that extends \( \alpha \cdot \alpha' \). By our construction, \( \sigma_{i+1} \) is the execution fragment that consists of events by \( p_n \) on base objects accessed
Throughout this proof, we assume that, for all \( \pi \), every RW DAP opaque TM implementation exists some transaction \( \alpha \) that has an enabled nontrivial event to \( b \), where \( \alpha \) does not contain nontrivial events on any base object accessed in \( \sigma_1 \cdots \sigma_i \). We now claim that \( \beta \) does not contain nontrivial events to \( b_{i+1} \). Suppose otherwise. Thus, there exists some transaction \( T' \) that has an enabled nontrivial event to \( b_{i+1} \) in the configuration after \( \alpha \cdot \alpha' \cdot \beta' \), where \( \beta' \) is some prefix of \( \beta \). But this contradicts the choice of \( \alpha \cdot \alpha' \) as the extension of \( \alpha \) that maximizes \( k' \).

Thus, \( \alpha \cdot \alpha' \cdot \sigma_1 \cdots \sigma_i \cdot \sigma_{i+1} \) is indeed a \((k+k')\)-stall execution for \( T_n \) where \( 1 < k < (k+k') \leq (n-1) \).

### 4.3 RAW/AWAR complexity

Attia et al. \cite{3} identified two common expensive synchronization patterns that frequently arise in the design of concurrent algorithms: read-after-write (RAW) and atomic write-after-read (AWAR). In this section, we prove that opaque, RW DAP TM implementations in \( \mathcal{OF} \) have executions in which some read-only transaction performs a linear \((n)\) number of RAWs or AWARs.

We recall the formal definitions of RAW and AWAR from \cite{3}. Let \( \pi^i \) denote the \( i \)-th event in an execution \( \pi \) \((i = 0, \ldots, |\pi| - 1)\).

We say that a transaction \( T \) performs a RAW \((\text{read-after-write})\) in \( \pi \) if \( \exists i, j; 0 \leq i < j < |\pi| \) such that (1) \( \pi^i \) is a write to a base object \( b \) by \( T \), (2) \( \pi^j \) is a read of a base object \( b' \neq b \) by \( T \), and (3) there is no \( \pi^k \) such that \( i < k < j \) and \( \pi^k \) is a write to \( b' \) by \( T \). In this paper, we are concerned only with non-overlapping RAWs, i.e., the read performed by one precedes the write performed by the other.

We say a transaction \( T \) performs an AWAR \((\text{atomic-write-after-read})\) in \( \pi \) if \( \exists i, 0 \leq i < |\pi| \) such that the event \( \pi^i \) is the application of a nontrivial primitive that atomically reads a base object \( b \) and writes to \( b \).

**Theorem 6.** Every RW DAP opaque TM implementation \( M \in \mathcal{OF} \) has an execution \( E \) in which some read-only transaction \( T \in \text{txns}(E) \) performs \( \Omega(n) \) non-overlapping RAW/AWARS.

**Proof.** For all \( j \in \{1, \ldots, m\}; m = n - 3 \), let \( v \) be the initial value of t-objects \( X_j \) and \( Z_j \). Throughout this proof, we assume that, for all \( i \in \{1, \ldots, n\} \), transaction \( T_i \) is invoked by process \( p_i \).
By OF TM-progress and OF TM-liveness, any opaque and RW DAP TM implementation \( M \in \mathcal{OF} \) has an execution of the form \( \tilde{\rho}_1 \cdots \tilde{\rho}_m \), where for all \( j \in \{1, \ldots, m\} \), \( \tilde{\rho}_j \) denotes the t-complete step contention-free execution of transaction \( T_j \) that performs \( \text{read}_j(Z_j) \rightarrow v \), writes \( v \) to \( X_j \) and commits.

By construction, any two transactions that participate in \( \tilde{\rho}_1 \cdots \tilde{\rho}_n \) are mutually read-write-disjoint-access and cannot contend on the same base object. It follows that for all \( 1 \leq j \leq m, \tilde{\rho}_j \) is an execution of \( M \).

For all \( j \in \{1, \ldots, m\} \), we iteratively define an execution \( \rho_j \) of \( M \) as follows: it is the longest prefix of \( \tilde{\rho}_j \) such that \( \rho_1 \cdots \rho_j \) cannot be extended with the complete step contention-free execution fragment of transaction \( T_n \) that performs \( j \) t-reads: \( \text{read}_n(X_1) \cdots \text{read}_n(X_j) \) in which \( \text{read}_n(X_j) \rightarrow \nu v \) nor with the complete step contention-free execution fragment of transaction \( T_{n-1} \) that performs \( j \) t-reads: \( \text{read}_{n-1}(X_1) \cdots \text{read}_{n-1}(X_j) \) in which \( \text{read}_{n-1}(X_j) \rightarrow \nu v \) (Figure 3a).

For any \( j \in \{1, \ldots, m\} \), let \( e_j \) be the event transaction \( T_j \) is poised to apply in the configuration after \( \rho_1 \cdots \rho_j \). Thus, the execution \( \rho_1 \cdots \rho_j \cdot e_j \) can be extended with the complete step contention-free executions of at least one of transaction \( T_n \) or \( T_{n-1} \) that performs \( j \) t-reads of \( X_1, \ldots, X_j \) in which the t-read of \( X_j \) returns the new value \( \nu v \). Let \( T_{n-1} \) be the transaction that must return the new value for the maximum number of \( X_j \)’s when \( \rho_1 \cdots \rho_j \cdot e_j \) is extended with the t-reads of \( X_1, \ldots, X_j \). We show that, in the worst-case, transaction \( T_n \) 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 \( \rho_1 \cdots \rho_m \).

Symmetric arguments apply for the case when \( T_n \) must return the new value for the maximum number of \( X_j \)’s when \( \rho_1 \cdots \rho_j \cdot e_j \) is extended with the t-reads of \( X_1, \ldots, X_j \).

**Proving the RAW/AWAR lower bound.** We prove that transaction \( T_n \) must perform \( \lceil \frac{m}{2} \rceil \) non-overlapping RAWs or AWARs in the course of performing \( m \) t-reads of \( X_1, \ldots, X_m \) immediately after the execution \( \rho_1 \cdots \rho_m \). Specifically, we prove that \( T_n \) must perform a RAW or an AWAR during the execution of the t-read of each \( X_j \) such that \( \rho_1 \cdots \rho_j \cdot e_j \) can be extended with the complete step contention-free execution of \( T_{n-1} \) as it performs \( j \) t-reads of \( X_1, \ldots, X_j \) in which the t-read of \( X_j \) returns the new value \( \nu v \). Let \( J \) denote the \( \{ \} \) of all \( j \in \{1, \ldots, m\} \) such that \( \rho_1 \cdots \rho_j \cdot e_j \) is extended with the complete step contention-free execution of \( T_{n-1} \) performing \( j \) t-reads of \( X_1, \ldots, X_j \) must return the new value \( \nu v \) during the t-read of \( X_j \).

We first prove that, for all \( j \in J, M \) has an execution of the form \( \rho_1 \cdots \rho_m \cdot \delta_j \) (Figures 3a and 3b), where \( \delta_j \) is the complete step contention-free execution fragment of \( T_n \) that performs \( j \) t-reads: \( \text{read}_n(X_1) \cdots \text{read}_n(X_j) \), each of which return the initial value \( v \).

By definition of \( \rho_j \), OF TM-progress and OF TM-liveness, \( M \) has an execution of the form \( \rho_1 \cdots \rho_j \cdot \delta_j \). By construction, transaction \( T_n \) is read-write disjoint-access with each transaction \( T \in \{T_{j+1}, \ldots, T_m\} \) in \( \rho_1 \cdots \rho_j \cdot \rho_m \cdot \delta_j \). Thus, \( T_n \) cannot contend with any of the transactions in \( \{T_{j+1}, \ldots, T_m\} \), implying that, for all \( j \in \{1, \ldots, m\} \), \( M \) has an execution of the form \( \rho_1 \cdots \rho_m \cdot \delta_j \) (Figure 3b).

We claim that, for each \( j \in J \), the t-read of \( X_j \) performed by \( T_n \) must perform a RAW or an AWAR in the course of performing \( j \) t-reads of \( X_1, \ldots, X_j \) immediately after \( \rho_1 \cdots \rho_m \). Suppose by contradiction that \( \text{read}_n(X_j) \) does not perform a RAW or an AWAR in \( \rho_1 \cdots \rho_m \cdot \delta_n \).

**Claim 7.** For all \( j \in J, M \) has an execution of the form \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \delta_{j-1} \cdot e_j \cdot \beta \) where, \( \beta \) is the complete step contention-free execution fragment of transaction \( T_{n-1} \) that performs \( j \) t-reads: \( \text{read}_{n-1}(X_1) \cdots \text{read}_{n-1}(X_{j-1}) \cdot \text{read}_{n-1}(X_j) \) in which \( \text{read}_{n-1}(X_j) \rightarrow \nu v \).

**Proof.** We observe that transaction \( T_n \) is read-write disjoint-access with every transaction \( T \in \{T_j, T_{j+1}, \ldots, T_m\} \) in \( \rho_1 \cdots \rho_j \cdot \rho_m \cdot \delta_{j-1} \). By RW DAP, it follows that \( M \) has an execution of the form \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \delta_{j-1} \cdot e_j \) since \( T_n \) cannot perform a nontrivial event on the base object accessed by \( T_j \) in the event \( e_j \).

By the definition of \( \rho_j \), transaction \( T_{n-1} \) must access the base object to which \( T_j \) applies a nontrivial primitive in \( e_j \) to return the value \( \nu v \) of \( X_j \) as it performs \( j \) t-reads of \( X_1, \ldots, X_j \) immediately after the execution \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \delta_{j-1} \cdot e_j \). Thus, \( M \) has an execution of the form \( \rho_1 \cdots \rho_j \cdot \delta_{j-1} \cdot e_j \cdot \beta \).
By construction, transactions \( T_{n-1} \) is read-write disjoint-access with every transaction \( T \in \{ T_{j+1}, \ldots, T_n \} \) in \( \rho_1 \cdot \rho_2 \cdots \rho_m \cdot \delta_{j-1} \cdot e_j \cdot \beta \). It follows that \( M \) has an execution of the form \( \rho_1 \cdot \rho_2 \cdots \rho_m \cdot \delta_{j-1} \cdot e_j \cdot \beta \).

Claim 8. For all \( j \in \{1, \ldots, m\} \), \( M \) has an execution of the form \( \rho_1 \cdot \rho_2 \cdots \rho_m \cdot \gamma \cdot \delta_{j-1} \cdot e_j \cdot \beta \), where \( \gamma \) is the t-complete step contention-free execution fragment of transaction \( T_{n-2} \) that writes \( \nu \neq v \) to \( Z_j \) and commits.

**Proof.** Observe that \( T_{n-2} \) precedes transactions \( T_n \) and \( T_{n-1} \) in real-time order in the above execution.

By OF TM-progress and OF TM-liveness, transaction \( T_{n-2} \) must be committed in \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \gamma \).

Since transaction \( T_{n-1} \) is read-write disjoint-access with \( T_{n-2} \) in \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \gamma \cdot \delta_{j-1} \cdot e_j \cdot \beta \), \( T_{n-1} \) does not contend with \( T_{n-2} \) on any base object (recall that we associate an edge with t-objects in the conflict graph only if they are both contained in the write set of some transaction). Since the execution fragment \( \beta \) contains an access to the base object to which \( T_j \) performs a nontrivial primitive in the event \( e_j \), \( T_{n-2} \) cannot perform a nontrivial event on this base object in \( \gamma \). It follows that \( M \) has an execution of the form \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \gamma \cdot \delta_{j-1} \cdot e_j \cdot \beta \) since, it is indistinguishable to \( T_{n-1} \) from the execution \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \delta_{j-1} \cdot e_j \cdot \beta \) (the existence of which is already established in Claim 7).

Recall that transaction \( T_n \) is read-write disjoint-access with \( T_{n-2} \) in \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \gamma \cdot \delta_j \). Thus, \( M \) has an execution of the form \( \rho_1 \cdots \rho_j \cdots \rho_m \cdot \gamma \cdot \delta_j \) (Figure 3d).

**Deriving a contradiction.** For all \( j \in \{1, \ldots, m\} \), we represent the execution fragment \( \delta_j \) as \( \delta_j \cdot \pi^j \), where \( \pi^j \) is the complete execution fragment of the \( j \)th t-read \( \text{read}_t(X_j) \rightarrow v \). By our assumption, \( \pi^j \) does not contain a RAW or an AWAR.

For succinctness, let \( \alpha = \rho_1 \cdots \rho_m \cdot \gamma \cdot \delta_{j-1} \). We now prove that if \( \pi^j \) does not contain a RAW or an AWAR, we can define \( \pi^1 \cdot \pi^2 = \pi^j \) to construct an execution of the form \( \alpha \cdot \pi^1 \cdot \pi^2 \cdot \pi^2 \) (Figure 3d) such that

- no event in \( \pi^1 \) is the application of a nontrivial primitive
- \( \alpha \cdot \pi^1 \cdot e_j \cdot \beta \cdot \pi^2 \) is indistinguishable to \( T_n \) from the step contention-free execution \( \alpha \cdot \pi^1 \cdot \pi^2 \)
- \( \alpha \cdot \pi^1 \cdot e_j \cdot \beta \cdot \pi^2 \) is indistinguishable to \( T_{n-1} \) from the step contention-free execution \( \alpha \cdot e_j \cdot \beta \).

The following claim defines \( \pi^1 \) and \( \pi^2 \) to construct this execution.

Claim 9. For all \( j \in \{1, \ldots, m\} \), \( M \) has an execution of the form \( \alpha \cdot \pi^1 \cdot \pi^2 \).

**Proof.** Let \( t \) be the first event containing a write to a base object in the execution fragment \( \pi^j \). We represent \( \pi^j \) as the execution fragment \( \pi^1 \cdot t \cdot \pi^f \). Since \( \pi^1 \) does not contain nontrivial events that write to a base object, \( \alpha \cdot \pi^1 \cdot e_j \cdot \beta \) is indistinguishable to transaction \( T_{n-1} \) from the step contention-free execution \( \alpha \cdot e_j \cdot \beta \) (as already proven in Claim 8). Consequently, \( \alpha \cdot \pi^1 \cdot e_j \cdot \beta \) is an execution of \( M \).

Since \( t \) is not an atomic-write-after-read, \( M \) has an execution of the form \( \alpha \cdot \gamma \cdot \pi^1 \cdot e_j \cdot \beta \cdot t \). Secondly, since \( \pi^j \) does not contain a read-after-write, any read of a base object performed in \( \pi^j \) may only be performed to base objects previously written in \( t \cdot \pi^f \). Thus, \( \alpha \cdot \pi^1 \cdot e_j \cdot \beta \cdot t \cdot \pi^f \) is indistinguishable to \( T_n \) from the step contention-free execution \( \alpha \cdot \pi^1 \cdot t \cdot \pi^f \). But, as already proved, \( \alpha \cdot \pi^j \) is an execution of \( M \).

Choosing \( \pi^2 = t \cdot \pi^f \), it follows that \( M \) has an execution of the form \( \alpha \cdot \pi^1 \cdot e_j \cdot \beta \cdot \pi^2 \).

We have now proved that, for all \( j \in \{1, \ldots, m\} \), \( M \) has an execution of the form \( \rho_1 \cdots \rho_m \cdot \gamma \cdot \delta_{j-1} \cdot \pi^1 \cdot e_j \cdot \beta \cdot \pi^2 \) (Figure 3d).

The execution in Figure 3d is not opaque. Indeed, 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 \( \text{read}_t(X_j) \) 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 \( \text{read}_j(Z_j) \) returns the initial value of \( Z_j \). The cycle \( T_j \rightarrow T_{n-2} \rightarrow T_n \rightarrow T_j \) implies that there exists no such a serialization.

Thus, for each \( j \in J \), transaction \( T_n \) must perform a RAW or an AWAR during the t-read of \( X_j \) in the course of performing \( m \) t-reads of \( X_1, \ldots, X_m \) immediately after \( \rho_1 \cdots \rho_m \). Since \( |J| \geq \lceil \frac{n-2}{2} \rceil \), in the worst-case, \( T_n \) must perform \( \Omega(n) \) RAW/AWARs during the execution of \( m \) t-reads immediately after \( \rho_1 \cdots \rho_m \).

\[ \square \]

5 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 inherent to implementations in \( OF \) (cf. Figure 1). Our implementation satisfies strict DAP, every transaction performs at most a single RAW and every t-read operation incurs \( O(1) \) memory stalls in any execution.

**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 \( X_j \), there is a bit \( L_j \), which if set, indicates the presence of an updating transaction writing to \( X_j \). For every process \( p_i \) and t-object \( X_j \), \( LP \) maintains a single-writer bit \( r_{ij} \) (only \( p_i \) is allowed to write to \( r_{ij} \)). Each of these base objects may be accessed only via read and write primitives.

**Updating transactions.** The \( \text{write}_k(X, v) \) implementation by process \( p_i \) simply stores the value \( v \) locally, deferring the actual updates to \( \text{try}_C_k \). During \( \text{try}_C_k \), 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 \( p_i \) has written 1 to any \( r_{ij} \) 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 returns \( A_k \). 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 \( \text{try}_C_k \) 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 concurrent 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 \( r_{ij} \).

**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 \( T_k \), \( \text{try}_C_k \) simply returns the commit response.

**Complexity.** Observe that our implementation uses invisible reads since 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. Consequently, every transaction performs \( O(1) \) non-overlapping RAWs in any execution.

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. Since the \( \text{read}_k(X_j) \) implementation only accesses base objects \( v_j \) and \( L_j \), and the validating \( T_k \)’s read set does not cause any stalls, it follows that each t-operation performs \( O(1) \) stalls in every execution.

Moreover, \( LP \) ensures that any two transactions \( T_i \) and \( T_j \) access the same base object iff there exists \( X \in Dset(T_i) \cap Dset(T_j) \) (strict DAP) and maintains exactly one version for every
t-object at any prefix of the execution.

**Theorem 10.** Algorithm 1 describes a progressive, opaque and strict DAP TM implementation LP that provides wait-free TM-liveness, uses invisible reads and in every execution E of LP,

- every transaction $T \in \text{txns}(E)$ applies only read and write primitives in $E$,
- every transaction $T \in \text{txns}(E)$ performs at most a single RAW,
- for every transaction $T \in \text{txns}(E)$, every t-read operation performed by $T$ incurs $O(1)$ memory stalls in $E$.

6 Related work

The lower bounds and impossibility results presented in this paper apply to obstruction-free TMs, such as DSTM [20], FSTM [13], and others [13, 25, 30]. Our upper bound is inspired by the progressive TM of [23].

Attiya et al. [5] 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 [5]. This class is orthogonal to the class of obstruction-free TMs, as is the proof technique used to establish the impossibility.

Perelman et al. [27] 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 [27] unrelated to ours.

Guerraoui and Kapalka [17] 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 that progressive TMs are not subject to either of these lower bounds.

Attiya et al. introduced the RAW/AWAR metric and proved that it is impossible to derive RAW/AWAR-free implementations of a wide class of data types that include sets, queues and deadlock-free mutual exclusion. The metric was previously used in [23] to measure the complexity of read-only transactions in a strictly stronger (than OF) class of permissive TMs. Detailed coverage on memory fences and the RAW/AWAR metric can be found in [26].

To derive the linear lower bound on the memory stall complexity of obstruction-free TMs, we adopted the definition of a $k$-stall execution and certain proof steps from [2][10].

7 Discussion

**Lower bounds for obstruction-free TMs.** We chose obstruction-freedom to elucidate non-blocking TM-progress since it is a very weak non-blocking progress condition [21]. As highlighted in the paper by Ennals [12], (1) obstruction-freedom increases the number of concurrently executing transactions since transactions cannot wait for inactive transactions to complete, and (2) while performing a t-read, obstruction-free TMs like [13, 20] must forcefully abort pending conflicting transactions. Intuitively, (1) allows us to construct executions in which some pending transaction is stalled while accessing a base object by all other concurrent transactions waiting to apply nontrivial primitives on the base object. Observation (2) inspires the proof of the impossibility of invisible reads in Theorem 2. Typically, the reading transaction must acquire exclusive ownership of the object via mutual exclusion or employing a read-modify-write primitive like compare-and-swap, motivating the linear lower bound on expensive synchronization in Theorem 6. In practice though, obstruction-free TMs may possibly circumvent these lower bounds in models that allow the use of contention managers [28].
Observe that Theorems 2 and 5 assume strict serializability and thus, also hold under the assumption of stronger TM-correctness conditions like opacity, virtual-world consistency [22] and TMS [9].

Since there are at most $n$ concurrent transactions, we cannot do better than $(n - 1)$ stalls (cf. Definition 1). Thus, the lower bound of Theorem 5 is tight. Moreover, we conjecture that the linear (in $n$) lower bound of Theorem 6 for RW DAP opaque obstruction-free TMs can be strengthened to be linear in the size of the transaction’s read set. Then, Algorithm 2 in Appendix B would allow us to establish a linear tight bound (in the size of the transaction’s read set) for RW DAP opaque obstruction-free TMs.

Progressive vs. obstruction-free TMs. Progressiveness is a blocking TM-progress condition that is satisfied by several popular TM implementations like TL2 [7] and NOrec [6]. In general, progressiveness and obstruction-freedom are incomparable. On the one hand, a t-read $X$ by a transaction $T$ that runs step contention-free from a configuration that contains an incomplete t-write to $X$ is typically blocked or aborted in lock-based TMs; obstruction-free TMs however, must ensure that $T$ must complete its t-read of $X$ without blocking or aborting. On the other hand, progressiveness requires two non-conflicting transactions to commit even in executions that are not step contention-free; but this is not guaranteed by obstruction-freedom.

Intuitively, progressive implementations are not forced to abort conflicting transactions, which allows us to employ invisible reads, derive constant stall and RAW/AWAR implementations. While it is relatively easy to derive standalone progressive TM implementations that are not individually subject to the lower bounds of obstruction-free TMs (cf. Figure 1), our progressive opaque TM implementation $LP$ is not subject to any of the lower bounds we prove for implementations in $OF$.

Circa. 2005, several papers presented the case for a shift from TMs that provide obstruction-free TM-progress to lock-based progressive TMs [7,8,12]. They argued that lock-based TMs tend to outperform obstruction-free ones by allowing for simpler algorithms with lower overheads and their inherent progress issues may be resolved using timeouts and contention-managers. The lower bounds for non-blocking TMs and the complexity gap with our progressive TM implementation established in this paper suggest that this course correction was indeed justified.
References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] H. Attiya, R. Guerraoui, D. Hendler, and P. Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4), 2009.

[3] 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.

[4] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. *2013 IEEE 33rd International Conference on Distributed Computing Systems*, 0:601–610, 2013.

[5] 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.

[6] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPOPP*, pages 67–78, 2010.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[8] D. Dice and N. Shavit. What really makes transactions fast? In *Transact*, 2006.

[9] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5):769–799, 2013.

[10] F. Ellen, D. Hendler, and N. Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[11] R. Ennals. The lightweight transaction library. http://sourceforge.net/projects/libltx/files/.

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

[13] K. Fraser. Practical lock-freedom. Technical report, Cambridge University Computer Laboratory, 2003.

[14] 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.

[15] 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.

[16] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *POPL*, pages 404–415, 2009.

[17] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*, Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool, 2010.

[18] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.

[19] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[20] 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.
In this section, we describe our blocking TM implementation $LP$ that satisfies progressiveness and opacity [17]. We begin with the formal definition of opacity.

For simplicity of presentation, we assume that each execution $E$ begins with an “imaginary” transaction $T_0$ that writes initial values to all t-objects and commits before any other transaction begins in $E$. Let $E$ be a t-sequential execution. For every operation $\text{read}_k(X)$ in $E$, we define the latest written value of $X$ as follows: (1) If $T_k$ contains a $\text{write}_k(X,v)$ preceding $\text{read}_k(X)$, then the latest written value of $X$ is the value of the latest such write to $X$. (2) Otherwise, if $E$ contains a $\text{write}_m(X,v)$, $T_m$ precedes $T_k$, and $T_m$ commits in $E$, then the latest written value of $X$ is the value of the latest such write to $X$ in $E$. (This write is well-defined since $E$ starts with $T_0$ writing to all t-objects.) We say that $\text{read}_k(X)$ is legal in a t-sequential execution $E$ if it returns the latest written value of $X$, and $E$ is legal if every $\text{read}_k(X)$ in $H$ that does not return $A_k$ is legal in $E$.

For a history $H$, a completion of $H$, denoted $\overline{H}$, is a history derived from $H$ through the following procedure: (1) for every incomplete t-operation $\text{op}_k$ of $T_k \in \mathsf{txns}(H)$ in $H$, if $\text{op}_k = \text{read}_k \lor \text{write}_k$, insert $A_k$ somewhere after the invocation of $\text{op}_k$; otherwise, if $\text{op}_k = \text{tryC}_k()$, insert $C_k$ or $A_k$ somewhere after the last event of $T_k$. (2) for every complete transaction $T_k$ that is not t-complete, insert $\text{tryC}_k \cdot A_k$ somewhere after the last event of transaction $T_k$.

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

A Opaque progressive TM implementation $LP$
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 each process \(p_i\) and t-object \(X_j\)
4: single-writer bit
5: allows reads and writes
6: \(L_j\), for each t-object \(X_j\)
7: allows reads and writes
8: Local variables:  
9: \(Rset_k\), \(Wset_k\) for every transaction \(T_k\)
10: dictionaries storing \(\{X_m, v_m\}\)

11: \(\text{read}_k(X_j)\):  
12: if \(X_j \not\in \text{Rset}(T_k)\) then
13: \([ov_j, k_j] := \text{read}(v_j)\)
14: \(\text{Rset}(T_k) := \text{Rset}(T_k) \cup \{X_j, [ov_j, k_j]\}\)
15: if \(\text{read}(L_j) \neq 0\) then
16: \(A_k\)
17: if \(\text{validate()}\) then
18: \(A_k\)
19: \(\text{Return } ov_j\)
20: else
21: \([ov_j, 1] := \text{Rset}(T_k)\).\(\text{locate}(X_j)\)
22: \(\text{Return } ov_j\)

23: \(\text{write}_k(X_j, v)\):  
24: \(m_j := v\)
25: \(\text{Wset}(T_k) := \text{Wset}(T_k) \cup \{X_j\}\)
26: \(\text{Return } ok\)

27: \(\text{tryC}_k()\):  
28: if \(|\text{Wset}(T_k)| = 0\) then
29: \(\text{Return } C_k\)
30: locked := \(\text{acquire}(\text{Wset}(T_k))\)
31: if \(\neg\text{locked}\) then
32: \(\text{Return } A_k\)
33: if \(\text{isAbortable()}\) then
34: \(\text{release}(\text{Wset}(T_k))\)
35: \(\text{Return } A_k\)
36: \(\text{// Exclusive write access to each } v_j\)
37: for all \(X_j \in \text{Wset}(T_k)\) do
38: \(\text{write}(v_j, [m_j, k])\)
39: \(\text{release}(\text{Wset}(T_k))\)
40: \(\text{Return } C_k\)
41: \(\text{Function: } \text{release}(Q)\):  
42: for all \(X_j \in Q\) do
43: \(\text{write}(L_j, 0)\)
44: for all \(X_j \in Q\) do
45: \(\text{write}(r_{ij}, 0)\)
46: \(\text{Return } ok\)
47: \(\text{Function: } \text{acquire}(Q)\):  
48: for all \(X_j \in Q\) do
49: \(\text{write}(r_{ij}, 1)\)
50: if \(\exists X_j \in Q; t \neq k \ ; \text{read}(r_{ij}) = 1\) then
51: for all \(X_j \in Q\) do
52: \(\text{write}(r_{ij}, 0)\)
53: \(\text{Return } false\)
54: \(\text{// Exclusive write access to each } X_j\)
55: for all \(X_j \in Q\) do
56: \(\text{write}(L_j, 1)\)
57: \(\text{Return } true\)
58: \(\text{Return } true\)
59: \(\text{Return } false\)
60: \(\text{Return } false\)
61: \(\text{Return } false\)
62: \(\text{Function: } \text{validate()}\):  
63: if \(\exists X_j \in \text{Rset}(T_k); [ov_j, k_j] \neq \text{read}(v_j)\) then
64: \(\text{Return } true\)
65: \(\text{Return } false\)

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 \(T_k, T_m \in \text{txns}(H)\), if \(T_k \sqsubseteq^R_H T_m\), then \(T_k\) precedes \(T_m\) in \(S\), and \((2)\) \(S\) is equivalent to \(\text{cseq}(H)\), where \(\bar{H}\) is some completion of \(H\) and \(\text{cseq}(\bar{H})\) is the subsequence of \(H\) reduced to committed transactions in \(H\).

We refer to \(S\) as a serialization of \(H\).

We now prove that \(LP\) implements an opaque TM.

We introduce the following technical definition: process \(p_i\) \emph{holds a lock on } \(X_j\) after an execution \(\pi\) of \(\text{Algorithm}\) \(\宣\) if \(\pi\) contains the invocation of \(\text{acquire}(Q)\), \(X_j \in Q\) by \(p_i\) that returned \(true\), but does not contain a subsequent invocation of \(\text{release}(Q')\), \(X_j \in Q'\), by \(p_i\) in \(\pi\).
Lemma 11. For any object $X_j$, and any execution $\pi$ of Algorithm $\pi$, there exists at most one process that holds a lock on $X_j$ after $\pi$.

Proof. Assume, by contradiction, that there exists an execution $\pi$ after which processes $p_i$ and $p_k$ hold a lock on the same object, say $X_j$. In order to hold the lock on $X_j$, process $p_i$ writes 1 to register $r_{ij}$ and then checks if any other process $p_k$ has written 1 to $r_{kj}$. Since the corresponding operation acquire($Q$), $X_j \in Q$ invoked by $p_i$ returns true, $p_i$ read 0 in $r_{kj}$ in Line 49. But then $p_k$ also writes 1 to $r_{kj}$ and later reads that $r_{ij}$ is 1. This is because $p_k$ can write 1 to $r_{kj}$ only after the read of $r_{kj}$ returned 0 to $p_i$ which is preceded by the write of 1 to $r_{ij}$. Hence, there exists an object $X_j$ such that $r_{ij} = 1; i \neq k$, but the conditional in Line 49 returns true to process $p_k$—a contradiction. □

Observation 12. Let $\pi$ be any execution of Algorithm $\pi$. Then, for any updating transaction $T_k \in \text{txns}(\pi)$ executed by process $p_i$ writes to $L_j$ (in Line 54) or $v_j$ (in Line 37) for some $X_j \in Wset(T_k)$ immediately after $\pi$ iff $p_i$ holds the lock on $X_j$ after $\pi$.

Lemma 13. Algorithm $\pi$ implements an opaque TM.

Proof. Let $E$ by any finite execution of Algorithm $\pi$. Let $\prec_E$ denote a total-order on events in $E$.

Let $H$ denote a subsequence of $E$ constructed by selecting linearization points of t-operations performed in $E$. The linearization point of a t-operation $op$, denoted as $\ell_{op}$ is associated with a base object event or an event performed between the invocation and response of $op$ using the following procedure.

Completions. First, we obtain a completion of $E$ by removing some pending invocations and adding responses to the remaining pending invocations involving a transaction $T_k$ as follows: every incomplete read$_k$, write$_k$ operation is removed from $E$; an incomplete tryC$_k$ is removed from $E$ if $T_k$ has not performed any write to a base object during the release function in Line 38 otherwise it is completed by including C$_k$ after $E$.

Linearization points. Now a linearization $H$ of $E$ is obtained by associating linearization points to t-operations in the obtained completion of $E$ as follows:

- For every t-read $op_k$ that returns a non-A$_k$ value, $\ell_{op_k}$ is chosen as the event in Line 13 of Algorithm $\pi$ else, $\ell_{op_k}$ is chosen as invocation event of $op_k$.
- For every $op_k = \text{write}_k$ that returns, $\ell_{op_k}$ is chosen as the invocation event of $op_k$.
- For every $op_k = \text{tryC}_k$ that returns $C_k$ such that $Wset(T_k) \neq \emptyset$, $\ell_{op_k}$ is associated with the response of acquire in Line 30 else if $op_k$ returns $A_k$, $\ell_{op_k}$ is associated with the invocation event of $op_k$.
- For every $op_k = \text{tryC}_k$ that returns $C_k$ such that $Wset(T_k) = \emptyset$, $\ell_{op_k}$ is associated with Line 20 $\prec_H$ denotes a total-order on t-operations in the complete sequential history $H$.

Serialization points. The serialization of a transaction $T_j$, denoted as $\delta_{T_j}$ is associated with the linearization point of a t-operation performed within the execution of $T_j$.

We obtain a t-complete history $H$ from $H$ as follows: for every transaction $T_k$ in $H$ that is complete, but not t-complete, we insert $\text{tryC}_k \cdot A_k$ after $H$.

A t-complete t-sequential history $S$ is obtained by associating serialization points to transactions in $H$ as follows:

- If $T_k$ is an update transaction that commits, then $\delta_{T_k}$ is $\ell_{\text{tryC}_k}$.
- If $T_k$ is a read-only or aborted transaction in $H$, $\delta_{T_k}$ is assigned to the linearization point of the last t-read that returned a non-A$_k$ value in $T_k$.

$\prec_S$ denotes a total-order on transactions in the t-sequential history $S$.

Claim 14. If $T_i \prec_H T_j$, then $T_i \prec_S T_j$.

Proof. This follows from the fact that for a given transaction, its serialization point is chosen between the first and last event of the transaction implying if $T_i \prec_H T_j$, then $\delta_{T_i} \prec_E \delta_{T_j}$ implies $T_i \prec_S T_j$. □
Claim 15. Let \( T_k \) be any updating transaction that returns false from the invocation of isAbortable in Line 33. Then, \( T_k \) returns \( C_k \) within a finite number of its own steps in any extension of \( E \).

Proof. Observe that \( T_k \) performs the write to base objects \( v_j \) for every \( X_j \in Wset(T_k) \) and then invokes release in Lines 37 and 38 respectively. Since neither of these involve aborting the transaction or contain unbounded loops or waiting statements, it follows that \( T_k \) will return \( C_k \) within a finite number of its steps.

Claim 16. \( S \) is legal.

Proof. Observe that for every \( \text{read}_j(X_m) \rightarrow v \), there exists some transaction \( T_i \) that performs \( \text{write}_j(X_m, v) \) and completes the event in Line 37 such that \( \text{read}_j(X_m) \neq \text{HRT} \text{ write}_i(X_m, v) \). More specifically, \( \text{read}_j(X_m) \) returns as a non-abort response, the value of the base object \( v_m \) and \( v_m \) can be updated only by a transaction \( T_i \) such that \( X_m \in Wset(T_i) \). Since \( \text{read}_j(X_m) \) returns the response \( v \), the event in Line 13 succeeds the event in Line 37 performed by \( \text{tryC}_i \). Consequently, by Claim 15 and the assignment of linearization points, \( \ell_{\text{tryC}_i} < E \ell_{\text{read}_j(X_m)} \). Since, for any updating committing transaction \( T_i \), \( \delta_{T_i} = \ell_{\text{tryC}_i} \), by the assignment of serialization points, it follows that \( \delta_{T_i} < E \delta_{T_k} \).

Thus, to prove that \( S \) is legal, it suffices to show that there does not exist a transaction \( T_k \) that returns \( C_k \) in \( S \) and performs \( \text{write}_k(X_m, v') \); \( v' \neq v \) such that \( T_i < S T_k < S T_j \). Suppose that there exists a committed transaction \( T_k, X_m \in Wset(T_k) \) such that \( T_i < S T_k < S T_j \).

\( T_i \) and \( T_k \) are both updating transactions that commit. Thus, 

\[
(T_i < S T_k) \iff (\delta_{T_i} < E \delta_{T_k})
\]

\[
(\delta_{T_k} < E \delta_{T_i}) \iff (\ell_{\text{tryC}_i} < E \ell_{\text{tryC}_k})
\]

Since, \( T_j \) reads the value of \( X \) written by \( T_i \), one of the following is true: \( \ell_{\text{tryC}_i} < E \ell_{\text{tryC}_k} < E \ell_{\text{read}_j(X_m)} \) or \( \ell_{\text{tryC}_i} < E \ell_{\text{read}_j(X_m)} < E \ell_{\text{tryC}_k} \). Let \( T_i \) and \( T_k \) be executed by processes \( p_i \) and \( p_k \) respectively.

Consider the case that \( \ell_{\text{tryC}_i} < E \ell_{\text{tryC}_k} < E \ell_{\text{read}_j(X_m)} \).

By the assignment of linearization points, \( T_k \) returns a response from the event in Line 30 before the read of \( v_m \) by \( T_j \) in Line 13. Since \( T_i \) and \( T_k \) are both committed in \( E \), \( p_k \) returns true from the event in Line 30 only after \( T_i \) writes 0 to \( r_{km} \) in Line 44 (Lemma 11).

Recall that \( \text{read}_j(X_m) \) checks if \( X_m \) is locked by a concurrent transaction (i.e \( L_j \neq 0 \)), then performs read-validation (Line 13) before returning a matching response. Consider the following possible sequence of events: \( T_k \) returns true from the acquire function invocation, sets \( L_j \) to 1 for every \( X_j \in Wset(T_k) \) (Line 54) and updates the value of \( X_m \) to shared-memory (Line 37). The implementation of \( \text{read}_j(X_m) \) then reads the base object \( v_m \) associated with \( X_m \) after which \( T_k \) releases \( X_m \) by writing 0 to \( r_{km} \) and finally \( T_j \) performs the check in Line 15. However, \( \text{read}_j(X_m) \) is forced to return \( A_j \) because \( X_m \in Rset(T_j) \) (Line 14) and has been invalidated since last reading its value. Otherwise suppose that \( T_k \) acquires exclusive access to \( X_m \) by writing 1 to \( r_{km} \) and returns true from the invocation of acquire, updates \( v_m \) in Line 37, \( T_j \) reads \( v_m \), \( T_j \) performs the check in Line 15 and finally \( T_k \) releases \( X_m \) by writing 0 to \( r_{km} \). Again, \( \text{read}_j(X_m) \) returns \( A_j \) since \( T_j \) reads that \( r_{km} \) is 1—contradiction.

Thus, \( \ell_{\text{tryC}_i} < E \ell_{\text{read}_j(X_m)} < E \ell_{\text{tryC}_k} \).

We now need to prove that \( \delta_{T_j} \) indeed precedes \( \ell_{\text{tryC}_k} \) in \( E \).

Consider the two possible cases:

- Suppose that \( T_j \) is a read-only or aborted transaction in \( H \). Then, \( \delta_{T_j} \) is assigned to the last t-read performed by \( T_j \) that returns a non-\( A_j \) value. If \( \text{read}_j(X_m) \) is not the last t-read performed by \( T_j \) that returned a non-\( A_j \) value, then there exists a \( \text{read}_j(X_z) \) performed by \( T_j \) such that \( \ell_{\text{read}_j(X_m)} < E \ell_{\text{tryC}_k} < E \ell_{\text{read}_j(X_z)} \). Now assume that \( \ell_{\text{tryC}_k} \) must precede \( \ell_{\text{read}_j(X_z)} \) to obtain a legal \( S \). Since \( T_k \) and \( T_j \) are concurrent in \( E \), we are restricted to the case that \( T_k \) performs a \( \text{write}_k(X_z, v) \) and \( \text{read}_j(X_z) \) returns \( v \). However, we claim that this t-read of \( X_z \) must abort by performing the checks in Line 15. Observe that \( T_k \) writes 1 to \( L_m, L_z \)
Theorem 9. Algorithm 1 describes a progressive, opaque and strict DAP TM implementation LP that provides wait-free TM-liveness, uses invisible reads and in every execution E of LP,

- every transaction $T \in \text{txns}(E)$ applies only read and write primitives in $E$,
- every transaction $T \in \text{txns}(E)$ performs at most a single RAW,
- for every transaction $T \in \text{txns}(E)$, every t-read operation performed by $T$ incurs $O(1)$ memory stalls in $E$.

Proof. (TM-liveness and TM-progress) Since none of the implementations of the t-operations in Algorithm 1 contain unbounded loops or waiting statements, every t-operation $op_k$ returns a matching response after taking a finite number of steps in every execution. Thus, Algorithm 1 provides wait-free TM-liveness.

To prove progressiveness, we proceed by enumerating the cases under which a transaction $T_k$ may be aborted.

- Suppose that there exists a read$_k(X_j)$ performed by $T_k$ that returns $A_k$ from Line 15. Thus, there exists a process $p_t$ executing a transaction that has written 1 to $r_{ij}$ in Line 48 but has not yet written 0 to $r_{ij}$ in Line 14 or some t-object in Rset($T_k$) has been updated since its t-read by $T_k$. In both cases, there exists a concurrent transaction performing a t-write to some t-object in Rset($T_k$).
- Suppose that tryC$_k$ performed by $T_k$ that returns $A_k$ from Line 31. Thus, there exists a process $p_t$ executing a transaction that has written 1 to $r_{ij}$ in Line 48 but has not yet written 0 to $r_{ij}$ in Line 44. Thus, $T_k$ encounters step-contention with another transaction that concurrently attempts to update a t-object in Wset($T_k$).
- Suppose that tryC$_k$ performed by $T_k$ that returns $A_k$ from Line 33. Since $T_k$ returns $A_k$ from Line 33 for the same reason it returns $A_k$ after Line 15, the proof follows.

(Strict disjoint-access parallelism) Consider any execution $E$ of Algorithm 1 and let $T_i$ and $T_j$ be any two transactions that participate in $E$ and access the same base object $b$ in $E$.

- Suppose that $T_i$ and $T_j$ contend on base object $v_j$ or $L_j$. Since for every t-object $X_j$, there exists distinct base objects $v_j$ and $L_j$, $T_j$ and $T_j$ contend on $v_j$ only if $X_j \in Dset(T_i) \cap Dset(T_j)$.
- Suppose that $T_i$ and $T_j$ contend on base object $r_{ij}$. Without loss of generality, let $p_i$ be the process executing transaction $T_i$; $X_j \in Wset(T_i)$ that writes 1 to $r_{ij}$ in Line 48. Indeed, no other process executing a transaction that writes to $X_j$ can write to $r_{ij}$. Transaction $T_j$ reads $r_{ij}$ only if $X_j \in Dset(T_j)$ as evident from the accesses performed in Lines 48, 49, 44.

Thus, $T_i$ and $T_j$ access the same base object only if they access a common t-object.

(Opacity) Follows from Lemma 13.

(Invisible reads) Observe that read-only transactions do not perform any nontrivial events. Secondly, in any execution $E$ of Algorithm 1 and any transaction $T_k \in \text{txns}(E)$, if $X_j \in Rset(T_k)$, $T_k$ does not write to any of the base objects associated with $X_j$ nor write any information that reveals its read set to other transactions.
Proof. Since opacity is a safety property, we only consider finite executions \([4]\). Let \(\mathrm{finite~execution~of~Algorithm~2}\).

- Let \(\ell \in \text{txns}(E)\) be a write event performed by some transaction \(T_k\) executed by process \(p_i\) in \(E\) on base objects \(v_j\) and \(L_j\) (Lines \([37]\) and \([54]\)). Any transaction \(T_k\) performs a write to \(v_j\) or \(L_j\) only after \(T_k\) writes to \(r_{ij}\), for every \(X_j \in Wset(T_k)\). Thus, by Lemmata \([11]\) and \([13]\), it follows that events that involve an access to either of these base objects incurs \(O(1)\) stalls.
- Let \(\delta\) be a write event on base object \(r_{ij}\) (Line \([48]\)) while writing to t-object \(X_j\). By Algorithm \([1]\) no other process can write to \(r_{ij}\). It follows that any transaction \(T_k \in \text{txns}(E)\) incurs \(O(1)\) memory stalls on account of any event it performs in \(E\). Observe that any t-read \(\text{read}_k(X_j)\) only accesses base objects \(v_j\), \(L_j\) and other value base objects in \(Rset(T_k)\). But as already established above, these are \(O(1)\) stall events. Hence, every t-read operation incurs \(O(1)\)-stalls in \(E\).

\[\square\]

B Obstruction-free TMs

B.1 An opaque RW DAP TM implementation \(M \in \mathcal{OF}\)

Lemma 10. Algorithm \([2]\) implements an opaque TM.

Proof. Since opacity is a safety property, we only consider finite executions \([4]\). Let \(E\) be any finite execution of Algorithm \([2]\). Let \(<_E\) denote a total-order on events in \(E\).

Let \(H\) denote a subsequence of \(E\) constructed by selecting linearization points of t-operations performed in \(E\). The linearization point of a t-operation \(op\), denoted as \(\ell_{op}\) is associated with a base object event or an event performed during the execution of \(op\) using the following procedure.

**Completions.** First, we obtain a completion of \(E\) by removing some pending invocations and adding responses to the remaining pending invocations involving a transaction \(T_k\) as follows: every incomplete \(\text{read}_k\), \(\text{write}_k\), \(\text{try}_C_k\) operation is removed from \(E\); an incomplete \(\text{write}_k\) is removed from \(E\).

**Linearization points.** We now associate linearization points to t-operations in the obtained completion of \(E\) as follows:

- For every t-read \(\text{op}_k\) that returns a non-A\(_k\) value, \(\ell_{op_k}\) is chosen as the event in Line \([13]\) of Algorithm \([2]\) else, \(\ell_{op_k}\) is chosen as invocation event of \(\text{op}_k\)
- For every t-write \(\text{op}_k\) that returns a non-A\(_k\) value, \(\ell_{op_k}\) is chosen as the event in Line \([37]\) of Algorithm \([2]\) else, \(\ell_{op_k}\) is chosen as invocation event of \(\text{op}_k\)
- For every \(\text{op}_k = \text{try}_C_k\) that returns \(C_k\), \(\ell_{op_k}\) is associated with Line \([60]\)

\(<_H\) denotes a total-order on t-operations in the complete sequential history \(H\).

**Serialization points.** The serialization of a transaction \(T_j\), denoted as \(\delta_T\), is associated with the linearization point of a t-operation performed during the execution of the transaction.

We obtain a t-complete history \(\bar{H}\) from \(H\) as follows: for every transaction \(T_k\) in \(H\) that is complete, but not t-complete, we insert \(\text{try}_C_k \cdot A_k\) after \(H\).

\(\bar{H}\) is thus a t-complete sequential history. A t-complete t-sequential history \(S\) equivalent to \(\bar{H}\) is obtained by associating serialization points to transactions in \(\bar{H}\) as follows:

- If \(T_k\) is an update transaction that commits, then \(\delta_{T_k}\) is \(\ell_{\text{try}_C_k}\)
- If \(T_k\) is an aborted or read-only transaction in \(\bar{H}\), then \(\delta_{T_k}\) is assigned to the linearization point of the last t-read that returned a non-A\(_k\) value in \(T_k\)

\(<_S\) denotes a total-order on transactions in the t-sequential history \(S\).
Claim 11. If \( T_i \prec_{RT} T_j \), then \( T_i \prec_{S} T_j \).

Proof. This follows from the fact that for a given transaction, its serialization point is chosen between the first and last event of the transaction implying if \( T_i \prec_{RT} T_j \), then \( \delta_{T_i} \prec_{E} \delta_{T_j} \) implies \( T_i \prec_{S} T_j \).

Claim 12. If transaction \( T_i \) returns \( C_i \) in \( E \), then \( \text{status}[i] = \text{committed} \) in \( E \).

Proof. Transaction \( T_i \) must perform the event in Line 66 before returning \( T_i \) i.e. the \text{cas} on its own \text{status} to change the value to \text{committed}. The proof now follows from the fact that any other transaction may change the \text{status} of \( T_i \) only if it is \text{live} (Lines 45 and 21).

Claim 13. \( S \) is legal.

Proof. Observe that for every \( \text{read}_j(X) \rightarrow v \), there exists some transaction \( T_i \) that performs \( \text{write}_j(X, v) \) and completes the event in Line 19 to write \( v \) as the new value of \( X \) such that \( \text{read}_j(X) \not\prec_{RT} \text{write}_j(X, v) \). For any updating committing transaction \( T_i \), \( \delta_{T_i} = \ell_{\text{tryC}} \). Since \( \text{read}_j(X) \) returns a response \( v \), the event in Line 13 must succeed the event in Line 66 when \( T_i \) changes \text{status}[i] to \text{committed}. Suppose otherwise, then \( \text{read}_j(X) \) subsequently forces \( T_i \) to abort
by writing aborted to status/[i] and must return the old value of $X$ that is updated by the previous owner of $X$, which must be committed in $E$ (Line 40). Since $\delta_{T_i} = \ell_{\text{try}C_i}$ precedes the event in Line 66, it follows that $\delta_{T_i} < E \ell_{\text{read}_j}(X)$.

We now need to prove that $\delta_{T_j} < E \delta_{T_i}$. Consider the following cases:

- if $T_j$ is an updating committed transaction, then $\delta_{T_j}$ is assigned to $\ell_{\text{try}C_j}$. But since $\ell_{\text{read}_j}(X) < E \ell_{\text{try}C_j}$, it follows that $\delta_{T_i} < E \delta_{T_j}$.
- if $T_j$ is a read-only or aborted transaction, then $\delta_{T_j}$ is assigned to the last t-read that did not abort. Again, it follows that $\delta_{T_i} < E \delta_{T_j}$.

To prove that $S$ is legal, we need to show that, there does not exist any transaction $T_k$ that returns $C_k$ in $S$ and performs $\text{write}_k(X, v')$; $v' \neq v$ such that $T_i < S T_k < S T_j$. Now, suppose by contradiction that there exists a committed transaction $T_k$, $X \in \text{Wset}(T_k)$ that writes $v' \neq v$ to $X$ such that $T_i < S T_k < S T_j$. Since $T_i$ and $T_k$ are both updating transactions that commit, $(T_i < S T_k) \iff (\delta_{T_i} < E \delta_{T_k})$

$(\delta_{T_i} < E \delta_{T_k}) \iff (\ell_{\text{try}C_i} < E \ell_{\text{try}C_k})$

Since, $T_j$ reads the value of $X$ written by $T_i$, one of the following is true: $\ell_{\text{try}C_i} < E \ell_{\text{try}C_k} < E \ell_{\text{read}_j}(X)$ or $\ell_{\text{try}C_i} < E \ell_{\text{read}_j}(X) < E \ell_{\text{try}C_k}$.

If $\ell_{\text{try}C_i} < E \ell_{\text{try}C_k} < E \ell_{\text{read}_j}(X)$, then the event in Line 66 performed by $T_k$ when it changes the status field to committed precedes the event in Line 13 performed by $T_j$. Since $\ell_{\text{try}C_i} < E \ell_{\text{try}C_k}$ and both $T_i$ and $T_k$ are committed in $E$, $T_k$ must perform the event in Line 37 after $T_i$ changes status/[i] to committed since otherwise, $T_k$ would perform the event in Line 45 and change status/[i] to aborted, thereby forcing $T_i$ to return $A_i$. However, $\text{read}_j(X)$ observes that the owner of $X$ is $T_k$ and since the status of $T_k$ is committed at this point in the execution, $\text{read}_j(X)$ must return $v'$ and not $v$—contradiction.

Thus, $\ell_{\text{try}C_i} < E \ell_{\text{read}_j}(X) < E \ell_{\text{try}C_k}$. We now need to prove that $\delta_{T_j}$ indeed precedes $\delta_{T_k} = \ell_{\text{try}C_k}$ in $E$.

Now consider two cases:

- Suppose that $T_j$ is a read-only transaction. Then, $\delta_{T_j}$ is assigned to the last t-read performed by $T_j$ that returns a non-$A_j$ value. If $\text{read}_j(X)$ is not the last t-read that returned a non-$A_j$ value, then there exists a $\text{read}_j(X'')$ such that $\ell_{\text{read}_j}(X) < E \ell_{\text{try}C_k} < E \ell_{\text{read}_j}(X'')$. But then this t-read of $X''$ must abort since the value of $X$ has been updated by $T_k$ since $T_j$ first read $X$—contradiction.

- Suppose that $T_j$ is an updating transaction that commits, then $\delta_{T_j} = \ell_{\text{try}C_j}$, which implies that $\ell_{\text{read}_j}(X) < E \ell_{\text{try}C_k} < E \ell_{\text{try}C_j}$. Then, $T_j$ must necessarily perform the validation of its read set in Line 65 and return $A_j$—contradiction.

Claims 11 and 13 establish that Algorithm 2 is opaque.

**Theorem 14.** Algorithm 3 describes a RW DAP, opaque TM implementation $M \in \mathcal{OF}$ such that every execution $E$ of $M$ is an $O(n)$-stall execution for any t-read operation and every read-only transaction $T \in \text{trans}(E)$ performs $O(|\text{Rset}(T)|)$ AWARs in $E$.

**Proof.** (Opacity) Follows from Lemma 10

(TM-liveness and TM-progress) Since none of the implementations of the t-operations in Algorithm 2 contain unbounded loops or waiting statements, every t-operation $\text{op}_k$ returns a matching response after taking a finite number of steps. Thus, Algorithm 2 provides wait-free TM-liveness.

To prove OF TM-progress, we proceed by enumerating the cases under which a transaction $T_k$ may be aborted in any execution.

- Suppose that there exists a $\text{read}_k(X_m)$ performed by $T_k$ that returns $A_k$. If $\text{read}_k(X_m)$ returns $A_k$ in Line 28 then there exists a concurrent transaction that updated a t-object in $\text{Rset}(T_k)$ or changed status/[k] to aborted. In both cases, $T_k$ returns $A_k$ only because there is step contention.
• Suppose that there exists a write\(_k(X_m, v)\) performed by \(T_k\) that returns \(A_k\) in Line 54. Thus, either a concurrent transaction has changed status\([k]\) to aborted or the value in tvar\([m]\) has been updated since the event in Line 37. In both cases, \(T_k\) returns \(A_k\) only because of step contention with another transaction.

• Suppose that a read\(_k(X_m)\) or write\(_k(X_m, v)\) return \(A_k\) in Lines 21 and 45 respectively. Thus, a concurrent transaction has takes steps concurrently by updating the status of owner\(_m\) since the read by \(T_k\) in Lines 13 and 37 respectively.

• Suppose that try\(C_k()\) returns \(A_k\) in Line 62. This is because there exists a t-object in Rset\((T_k)\) that has been updated by a concurrent transaction since i.e. try\(C_k()\) returns \(A_k\) only on encountering step contention.

It follows that in any step contention-free execution of a transaction \(T_k\) from a \(T_k\)-free execution, \(T_k\) must return \(C_k\) after taking a finite number of steps.

(Read-write disjoint-access parallelism) Consider any execution \(E\) of Algorithm 2 and let \(T_i\) and \(T_j\) be any two transactions that contend on a base object \(b\) in \(E\). We need to prove that there is a path between a t-object in \(Dset(T_i)\) and a t-object in \(Dset(T_j)\) in \(G(T_i, T_j, E)\) or there exists \(X \in Dset(T_i) \cap Dset(T_j)\). Recall that there exists an edge between t-objects \(X\) and \(Y\) in \(G(T_i, T_j, E)\) only if there exists a transaction \(T \in txns(E)\) such that \(\{X, Y\} \in Wset(T)\).

• Suppose that \(T_i\) and \(T_j\) contend on base object tvar\([m]\) belonging to t-object \(X_m\) in \(E\). By Algorithm 2 a transaction accesses \(X_m\) only if \(X_m\) is contained in \(Dset(T_m)\). Thus, both \(T_i\) and \(T_j\) must access \(X_m\).

• Suppose that \(T_i\) and \(T_j\) contend on base object status\([i]\) in \(E\) (the case when \(T_i\) and \(T_j\) contend on status\([i]\) is symmetric). \(T_j\) accesses status\([i]\) while performing a t-read of some t-object \(X\) in Lines 15 and 21 only if \(T_i\) is the owner of \(X\). Also, \(T_j\) accesses status\([i]\) while performing a t-write to \(X\) in Lines 39 and 45 only if \(T_i\) is the owner of \(X\). But if \(T_i\) is the owner of \(X\), then \(X \in Wset(T_i)\).

• Suppose that \(T_i\) and \(T_j\) contend on base object status\([m]\) belonging to some transaction \(T_m\) in \(E\). Firstly, observe that \(T_i\) or \(T_j\) access status\([m]\) only if there exist t-objects \(X\) and \(Y\) in \(Dset(T_i)\) and \(Dset(T_j)\) respectively such that \(\{X, Y\} \in Wset(T_m)\). This is because \(T_i\) and \(T_j\) would both read status\([m]\) in Lines 15 (during t-read) and 39 (during t-write) only if \(T_m\) was the previous owner of \(X\) and \(Y\). Secondly, one of \(T_i\) or \(T_j\) applies a nontrivial primitive to status\([m]\) only if \(T_i\) and \(T_j\) read status\([m]\)=live in Lines 15 (during t-read) and 37 (during t-write). Thus, at least one of \(T_i\) or \(T_j\) is concurrent to \(T_m\) in \(E\). It follows that there exists a path between \(X\) and \(Y\) in \(G(T_i, T_j, T_m)\).

(Complexity) Every t-read operation performs at most one AWAR in an execution \(E\) (Line 21) of Algorithm 2. It follows that any read-only transaction \(T_k \in txns(E)\) performs at most |Rset\((T_k)\)| AWARs in \(E\).

The linear step-complexity is immediate from the fact that during the t-read operations, the transaction validates its entire read set (Line 25). All other t-operations incur \(O(1)\) step-complexity since they involve no iteration statements like for and while loops.

Since at most \(n - 1\) transactions may be t-incomplete at any point in an execution \(E\), it follows that \(E\) is at most a \((n - 1)\)-stall execution for any t-read \(op\) and every \(T \in txns(E)\) incurs \(O(n)\) stalls on account of any event performed in \(E\). More specifically, consider the following execution \(E\): for all \(i \in \{1, \ldots, n - 1\}\), each transaction \(T_i\) performs write\(_i(X_m, v)\) in a step-contention free execution until it is poised to apply a nontrivial event on tvar\([m]\) (Line 49). By OF TM-progress, we construct \(E\) such that each of the \(T_i\) is poised to apply a nontrivial event on tvar\([m]\) after \(E\). Consider the execution fragment of read\(_n(X_m)\) that is poised to perform an event \(e\) that reads tvar\([m]\) (Line 13) immediately after \(E\). In the constructed execution, \(T_n\) incurs \(O(n)\) stalls on account of \(e\) and thus, produces the desired \((n - 1)\)-stall execution for read\(_n(X)\).
Algorithm 3 Weak DAP opaque implementation $M \in OF$; code for $T_k$

1: read$_k(X_m)$;
2: \[ [owner_m, oval_m, nval_m] \leftarrow tv[\text{var}].read() \]
3: if $owner_m \neq k$ then
4: $s_m \leftarrow \text{status}[owner_m].read()$
5: if $s_m = \text{committed}$ then
6: $\text{curr} = nval_m$
7: else if $s_m = \text{aborted}$ then
8: $\text{curr} = oval_m$
9: else
10: if $\text{status}[owner_m].\text{cas}([\text{live}, \text{aborted})$ then
11: $\text{curr} = oval_m$
12: Return $A_k$
13: $o_m \leftarrow \text{tv[\text{var}].cas}([owner_m, oval_m, nval_m], [k, oval_m, nval_m])$
14: if $o_m \land \text{status}[k] = \text{live}$ then
15: $\text{Rset}(T_k).\text{add}([X_m, [owner_m, oval_m, nval_m]])$
16: Return $curr$
17: else
18: Return $\text{Rset}(T_k).\text{locate}(X_m)$
19: try$C_k()$:
20: if $\text{status}[k].\text{cas}([\text{live}, \text{committed})$ then
21: Return $C_k$
22: Return $A_k$

B.2 An opaque weak DAP implementation $M \in OF$

Algorithm 3 describes a weak DAP implementation in $OF$ that does not satisfy read-write DAP. The code for the t-write operations is identical to Algorithm 2.

Theorem 15. Algorithm 3 describes a weak TM implementation $M \in OF$ such that in any execution $E$ of $M$, for every transaction $T \in txns(E)$, $T$ performs $O(1)$ steps during the execution of any t-operation in $E$.

Proof. The proofs of opacity, TM-liveness and TM-progress are almost identical to the analogous proofs for Algorithm 2.

(Weak disjoint-access parallelism) Consider any execution $E$ of Algorithm 3 and let $T_i$ and $T_j$ be any two transactions that contend on a base object $b$ in $E$. We need to prove that there is a path between a t-object in $Dset(T_i)$ and a t-object in $Dset(T_j)$ in $\tilde{G}(T_i, T_j, E)$ or there exists $X \in Dset(T_i) \cap Dset(T_j)$. Recall that there exists an edge between t-objects $X$ and $Y$ in $G(T_i, T_j, E)$ only if there exists a transaction $T \in txns(E)$ such that \{X, Y\} $\in Dset(T)$.

- Suppose that $T_i$ and $T_j$ contend on base object $\text{tv[\text{var}]}$ belonging to t-object $X_m$ in $E$. By Algorithm 3, a transaction accesses $X_m$ only if $X_m$ is contained in $Dset(T_m)$. Thus, both $T_i$ and $T_j$ must access $X_m$.
- Suppose that $T_i$ and $T_j$ contend on base object $\text{status}[i]$ in $E$ (the case when $T_i$ and $T_j$ contend on $\text{status}[j]$ is symmetric). $T_j$ accesses $\text{status}[j]$ while performing a t-read of some t-object $X$ in Lines 4 and 10 only if $T_i$ is the owner of $X$. Also, $T_j$ accesses $\text{status}[i]$ while performing a t-write to $X$ in Lines 39 and 45 only if $T_i$ is the owner of $X$. But if $T_i$ is the owner of $X$, then $X \in Dset(T_i)$.
- Suppose that $T_i$ and $T_j$ contend on base object $\text{status}[m]$ belonging to some transaction $T_m$ in $E$. First, observe that if $T_i$ or $T_j$ access $\text{status}[m]$ only if there exist t-objects $X$ and $Y$ in $Dset(T_i)$ and $Dset(T_j)$ respectively such that \{X, Y\} $\in Dset(T_m)$. This is because $T_i$ and $T_j$ would both read $\text{status}[m]$ in Lines 4 (during t-read) and 39 (during t-write) only if $T_m$ was the previous owner of $X$ and $Y$. Secondly, one of $T_i$ or $T_j$ applies a nontrivial primitive to $\text{status}[m]$ only if $T_i$ and $T_j$ read $\text{status}[m]=\text{live}$ in Lines 4 (during t-read) and 37 (during t-write). Thus, at least one of $T_i$ or $T_j$ is concurrent to $T_m$ in $E$. It follows that there exists a path between $X$ and $Y$ in $\tilde{G}(T_i, T_j, E)$. 

26
(Complexity) Since no implementation of any of the t-operation contains any iteration statements like \textit{for} and \textit{while} loops), the proof follows. \hfill \Box