LOWER BOUNDS FOR TRANSACTIONAL MEMORY

Srivatsan Ravi

Purdue University

Abstract

Transactional memory allows the user to declare sequences of instructions as speculative *transactions* that can either *commit* or *abort*. If a transaction commits, it appears to be executed sequentially, so that the committed transactions constitute a correct sequential execution. If a transaction aborts, none of its update operations can affect other transactions. The TM implementation endeavors to execute these instructions in a manner that efficiently utilizes the concurrent computing facilities provided by multicore architectures.

The TM abstraction, in its original manifestation, extended the processor's instruction set with instructions to indicate which memory accesses must be transactional. Most popular TM designs, subsequent to the original proposal have implemented all the functionality in *software*. More recently, processors have included hardware extensions to support small transactions. Hardware transactions may be spuriously aborted due to several reasons: cache capacity overflow, interrupts *etc*. This has led to proposals for *hybrid* TMs in which the fast, but potentially unreliable hardware transactions are complemented with slower, but more reliable software transactions.

The complexity of TM implementations, whether realized in hardware or software, is characterized by several measures: ordering semantics for transactions, conditions under which transactions must terminate, conditions under which transactions must commit/abort, bound on the number of versions that can be maintained, choice of the complexity metric and complexity of *read-only* or *updating* transactions as well as a multitude of other implementation strategies. In this work, we survey known complexity bounds for implementing TM as a shared object and the implicit assumptions underlying these results.

1 Introduction

The wickedness and the foolishness of no man can avail against the fond optimism of mankind.

James Branch Cabell-The Silver Stallion

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.

TM implementations typically ensure that all committed transactions appear to execute sequentially in some total order respecting the timing of non-overlapping transactions. Moreover, intermediate states witnessed by the read operations of an incomplete transaction may affect the user application. Thus, to ensure that the TM implementation is *safe* and does not export any pathological executions, it is additionally expected that every transaction (including aborted and incomplete ones) must return responses that is consistent with some *sequential* execution of the TM implementation.

TM implementations. As a *synchronization abstraction*, TM came as an alternative to conventional lock-based synchronization. The TM abstraction, in its original manifestation, augmented the processor's *cache-coherence protocol* and extended the CPU's instruction set with instructions to indicate which memory accesses must be transactional [39]. Most popular TM designs, subsequent to the original proposal in [39] have implemented all the functionality in software [18, 29, 38, 50, 61]. Early software transactional memory (STM) implementations [29, 38, 50, 61, 63] 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* [37, 40] stipulating that every transaction running in the absence of *step contention*, *i.e.*, not encountering steps of concurrent transactions, must commit.

In 2005, Ennals [28] 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 [27] that he claimed to outperform *DSTM* [38], the most referenced obstruction-free TM implementation at
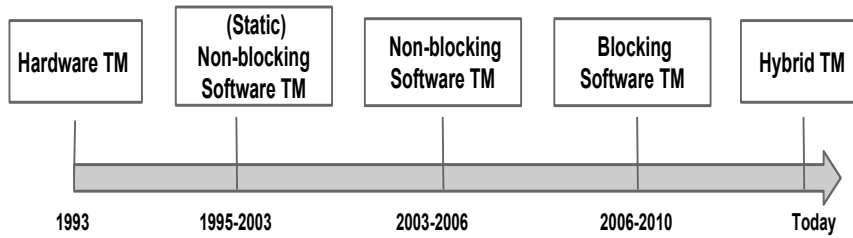
Figure 1: TM implementations: a brief history

the time. Inspired by [28], more recent TM implementations like *TL* [21], *TL2* [20] and *NOrec* [18] 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 [32]. Nonetheless, TM designs that are implemented entirely in software still incur significant performance overhead. Thus, current CPUs have included instructions to mark a block of memory accesses as transactional [1, 53, 56], allowing them to be executed *atomically* in hardware. Hardware transactions promise better performance than STMs, but they offer no progress guarantees since they may experience *spurious* aborts. This motivates the need for *hybrid* TMs in which the *fast* hardware transactions are complemented with *slower* software transactions that do not have spurious aborts.

Our focus. This work surveys lower bounds and (im)possibility results for TM implementations. We identify the popular complexity metrics (e.g. *expensive synchronization patterns* [8], *memory stalls* [26], number of memory steps etc.) and their relevance in the TM context. We survey known lower and upper bounds on the complexity of three classes of safe (software) TMs: *blocking* TMs that allow transactions to delay or abort due to *overlapping* transactions (Section 3), *non-blocking* TMs which adapt to *step contention* by ensuring that a transaction not encountering steps of overlapping transactions must commit (Section 4), and *partially non-blocking* TMs that provide strong non-blocking guarantees (*wait-freedom*) to only a subset of transactions (Section 5). We then survey attempts at modelling HyTMs and lower bounds that exhibit inherent trade-offs on the degree of concurrency allowed between hardware and software transactions and the costs introduced on the hardware (Section 6). We conclude with an overview of future research directions and open questions concerning complexity of TMs (Section 7).

2 Transactional memory model and preliminaries

TM interface. *Transactional memory* (in short, *TM*) allows a set of data items (called *t-objects*) to be accessed via atomic *transactions*. A transaction T_k may contain the following *t-operations*: $read_k(X)$ returns a value in some domain V (denoted $read_k(X) \rightarrow v$) or a special value $A_k \notin V$ (*abort*); $write_k(X, v)$, for a value $v \in V$, returns *ok* or A_k; $tryC_k$ returns $C_k \notin V$ (*commit*) or A_k.

TM implementations. We consider an asynchronous shared-memory system in which a set of *n* processes, communicate by applying *primitives* on shared *base objects*. We assume that processes issue transactions sequentially, *i.e.*, a process starts a new transaction only after its previous transaction has *completed* (committed or aborted). A TM *implementation* provides processes with algorithms for implementing $read_k$, $write_k$ and $tryC_k()$ of a transaction T_k by *applying primitives* from a set of shared *base objects*, each of which is assigned an *initial value*. A primitive is a generic *read-modify-write* (*rmw*) procedure applied to a base object [26, 36]. It is characterized by a pair of functions $\langle g, h \rangle$: given the current state of the base object, g is an *update function* that computes its state after the primitive is applied, while h is a *response function* that specifies the outcome of the primitive returned to the process. A rmw primitive is *trivial* if it never changes the value of the base object to which it is applied. Otherwise, it is *nontrivial*. A trivial rmw primitive is *conditional* if there exist configurations in which the primitive does not change the value of the base object. Observe that this model explicitly precludes the use of atomic primitives that access multiple base objects in a single step [24].

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

A *configuration* (of a TM implementation) specifies the value of each base object and the state of each process. The *initial configuration* is the configuration in which all base objects have their initial values and all processes are in their initial states.

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

Let E be an execution fragment. For a transaction T_k (and resp. process p_k), $E|k$ denotes the subsequence of E restricted to events of T_k (and resp. p_k). If $E|k$ is non-empty, we say that T_k (resp. p_k) *participates* in E, else we say E is

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

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

Orders on transactions. Let $txns(E)$ denote the set of transactions that participate in E. An execution E is *sequential* if every invocation of a t-operation is either the last event in the history H exported by E or is immediately followed by a matching response. We assume that executions are *well-formed*, *i.e.*, for all T_k, $E|k$ begins with the invocation of a t-operation, is sequential and has no events after A_k or C_k. A transaction $T_k \in txns(E)$ is *complete in E* if $E|k$ ends with a response event. The execution E is *complete* if all transactions in $txns(E)$ are complete in E. A transaction $T_k \in txns(E)$ is *t-complete* if $E|k$ ends with A_k or C_k; otherwise, T_k is *t-incomplete*. T_k is *committed* (resp., *aborted*) in E if the last event of T_k is C_k (resp., A_k). The execution E is *t-complete* if all transactions in $txns(E)$ are t-complete.

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

Contention. We say that a configuration C after an execution E is *quiescent* (resp., *t-quiescent*) if every transaction $T_k \in txns(E)$ 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* op_k if the events of $E|op_k$ are contiguous in E. We say that an execution fragment E is *step contention-free for* T_k if the events of $E|k$ are contiguous in E. We say that E is *step contention-free* if E is step contention-free for all transactions that participate in E.

TM-correctness. Informally, a t-sequential history S is *legal* if every t-read of a t-object returns the *latest written value* of this t-object. 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 [33].

A weaker condition called *strict serializability* ensures opacity only with respect to committed transactions while definitions like virtual-world consistency (VWC) [41] and transactional memory specification (TMS1) ensure strict serializability and restricted safety for aborted transactions [25]. We direct the reader to [9] for details on these definitions.

TM-progress. One may notice that a TM implementation that forces, in every execution to abort every transaction is trivially strictly serializable, but not very useful. A TM-progress condition specifies the conditions under which a transaction is allowed to abort. Technically, a TM-progress condition specified this way is a *safety property* since it can be violated in a finite execution.

TM-liveness. Observe that a TM-progress condition only specifies the conditions under which a transaction is aborted, but does not specify the conditions under which it must commit. Thus, in addition to a progress condition, we must stipulate a *liveness* [5, 49] condition.

Read invisibility. Informally, in a TM using *invisible reads*, a transaction cannot reveal any information about its read set to other transactions. Thus, given an execution E and some transaction T_k with a non-empty read set, transactions other than T_k cannot distinguish E from an execution in which T_k's read set is empty. This prevents TMs from applying nontrivial primitives during t-read operations and from announcing read sets of transactions during tryCommit. Most popular TM implementations like *TL2* [20] and *NOrec* [18] satisfy this property.

The notion of *weak* invisible that prevents t-read operations from applying nontrivial primitives only in the absence of concurrent transactions. Specifically, weak read invisibility allows t-read operations of a transaction T to be "visible", *i.e.*, write to base objects, only if T is concurrent with another transaction. For example, the popular TM implementation *DSTM* [38] satisfies weak invisible reads, but not 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

$Dset(T_i) \cap Dset(T_j) \neq \emptyset$ [33].

Informally, *weak* DAP [11] ensures that two transactions concurrently contend on the same base object only if their data sets are connected in the *conflict graph*, capturing data-set overlaps among all concurrent transactions [11]. *Read-write (RW) DAP* [45], a restriction of weak DAP and a relaxation of strict DAP, defines the conflict graph based on the *write-set overlaps* among concurrent transactions and is satisfied by several popular obstruction-free implementations [29, 38, 63].

Observe that every RW DAP TM implementation satisfies weak DAP, but not vice versa. Consider the following execution E that begins with the incomplete execution of a transaction T_0 that reads X and writes to Y, followed by the execution of two transactions T_1 and T_2 that access X and Y respectively. If E is an execution of a weak DAP TM, 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.

For any two DAP definitions D_1 and D_2, if every TM implementation that satisfies D_1 also satisfies D_2, but the converse is not true, we say that $D_2 \ll D_1$. The following observation is immediate.

Observation 1. *Weak DAP $\ll$ RW DAP $\ll$ Strict DAP $\ll$ Strict data-partitioning.*

3 Complexity of blocking TMs

We begin by overviewing TM implementations that are *blocking*. Figure 2 characterizes the class of blocking TMs: *single-lock* TMs that satisfy *sequential* TM-progress (a transaction may abort due to a concurrent transaction), *(strongly) progressive* TMs that allow transactions to abort only due to read-write conflicts on t-objects and finally *permissive* TMs that provide maximal concurrency allowing a transaction to abort only if committing it would violate TM-correctness.

3.1 Sequential TMs

A quadratic lower bound on step complexity. [47] showed that a read-only transaction in an opaque TM featured with weak DAP, weak invisible reads, *interval contention-free* (ICF) TM-liveness and sequential TM-progress must *incrementally* validate every next read operation. This results in a quadratic (in the size of the transaction's read set) step-complexity lower bound. Here ICF TM-liveness means, for every finite execution E such that the configuration after E is quiescent,
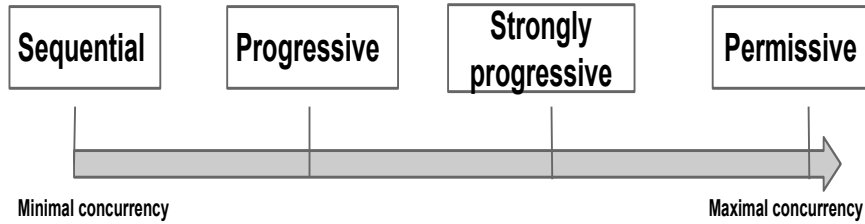
Figure 2: Classification of blocking TMs based on TM-progress: sequential (aborts due to concurrent transaction); progressive (aborts due to read-write conflicts; strongly progressive (progressive but at least one transaction from a set conflicting on single t-object must not be aborted); permissive (abort only due to TM-correctness violation)

and every transaction T_k that applies the invocation of a t-operation op_k immediately after E, the finite step contention-free extension for op_k contains a response. Secondly, [47] prove that if the TM-correctness property is weakened to strict serializability, there exist executions in which the tryCommit of some transaction must access a linear (in the size of the transaction's read set) number of distinct base objects.

Theorem 2 ([47]). *For every weak DAP TM implementation M that provides ICF TM-liveness, sequential TM-progress and uses weak invisible reads,*
  (1) If M is opaque, for every $m \in \mathbb{N}$, there exists an execution E of M such that some transaction $T \in txns(E)$ performs $\Omega(m^2)$ steps, where $m = |Rset(T_k)|$.
  (2) if M is strictly serializable, for every $m \in \mathbb{N}$, there exists an execution E of M such that some transaction $T_k \in txns(E)$ accesses at least $m - 1$ distinct base objects during the executions of the mth t-read operation and $tryC_k()$, where $m = |Rset(T_k)|$.

Theorem 2 improves the read-validation step-complexity lower bound [32, 33] derived for *strict-data partitioning* (a very strong version of DAP) and *invisible reads*. In a *strict data partitioned* TM, the set of base objects used by the TM is split into disjoint sets, each storing information only about a single data item. Indeed, every TM implementation that is strict data-partitioned satisfies weak DAP, but not vice-versa. The definition of invisible reads assumed in [32,33] requires that a t-read operation does not apply nontrivial events in any execution. Theorem 2 however, assumes *weak* invisible reads, stipulating that t-read operations of a transaction T do not apply nontrivial events only when T is not concurrent with any other transaction. We believe that the TM-progress and TM-liveness restrictions as well as the definitions of DAP and invisible reads considered for this result are

TM-correctness	TM-liveness	DAP	Invisible reads	Read-write	Complexity
Opacity	ICF	weak	yes	yes	$\Theta(|Rset|^2)$ step-complexity
Strict serializability	ICF	weak	yes	yes	$\Theta(|Rset|)$ step-complexity for tryCommit
Opacity	WF	strict	yes	yes	$O(1)$ RAW/AWAR, $O(1)$ stalls for t-reads
Opacity	starvation-free	strict			$\Theta(|Wset|)$ protected data

Table 1: Complexity bounds for progressive TMs.

the weakest possible assumptions that may be made. To the best of our knowledge, these assumptions cover every TM implementation that is subject to the validation step-complexity [18, 20, 38].

3.2 Progressive TMs

We turn our focus to *progressive* TM implementations which allow a transaction to be aborted only due to read-write conflicts with concurrent transactions.

A linear lower bound on protected data size. Kuznetsov *et al.* [44] introduce a new metric called *protected data size* that, intuitively, captures the amount of data that a transaction must exclusively control at some point of its execution. All progressive TM implementations (see, *e.g.*, an overview in [32]) use locks or timing assumptions to give an updating transaction exclusive access to all objects in its write set at some point of its execution. For example, lock-based progressive implementations like *TL* [21] and *TL2* [20] require that a transaction grabs all locks on its write set before updating the corresponding base objects. [44] shows that this is an inherent price to pay for providing progressive concurrency: every committed transaction in a progressive and strict DAP TM implementation providing *starvation-free* (each t-operation eventually returns a matching response, assuming that no concurrent t-operation stops indefinitely before returning) TM-liveness must, at some point of its execution, protect every t-object in its write set.

Let M be a progressive TM implementation providing starvation-free TM-liveness. Intuitively, a t-object X_j is protected at the end of some finite execution π of M if some transaction T_0 is about to atomically change the value of X_j in its next step (*e.g.*, by performing a compare-and-swap) or does not allow any concurrent transaction to read X_j (*e.g.*, by holding a "lock" on X_j).

Formally, let $\alpha \cdot \pi$ be an execution of M such that π is a t-sequential t-complete execution of a transaction T_0, where $Wset(T_0) = \{X_1, \ldots, X_m\}$. Let u_j ($j = 1, \ldots, m$) denote the value written by T_0 to t-object X_j in π. In this section, let π^t denote the t-th shortest prefix of π. Let π^0 denote the empty prefix.

For any $X_j \in Wset(T_0)$, let T_j denote a transaction that tries to read X_j and commit. Let $E_j^t = \alpha \cdot \pi^t \cdot \rho_j^t$ denote the extension of $\alpha \cdot \pi^t$ in which T_j runs solo until it completes. Note that, since we only require the implementation to be starvation-free, ρ_j^t can be infinite.

We say that $\alpha \cdot \pi^t$ is $(1, j)$-valent if the read operation performed by T_j in $\alpha \cdot \pi^t \cdot \rho_j^t$ returns u_j (the value written by T_0 to X_j). We say that $\alpha \cdot \pi^t$ is $(0, j)$-valent if the read operation performed by T_j in $\alpha \cdot \pi^t \cdot \rho_j^t$ does not abort and returns an "old" value $u \neq u_j$. Otherwise, if the read operation of T_j aborts or never returns in $\alpha \cdot \pi^t \cdot \rho_j^t$, we say that $\alpha \cdot \pi^t$ is $(\perp, j)$-valent.

Definition 1 ([44]). *We say that T_0 protects an object X_j in $\alpha \cdot \pi^t$, where π^t is the t-th shortest prefix of π (t > 0) if one of the following conditions holds: (1) $\alpha \cdot \pi^t$ is $(0, j)$-valent and $\alpha \cdot \pi^{t+1}$ is $(1, j)$-valent, or (2) $\alpha \cdot \pi^t$ or $\alpha \cdot \pi^{t+1}$ is $(\perp, j)$-valent.*

Theorem 3 ([44]). *Let M be a progressive, opaque and strict disjoint-access-parallel TM implementation that provides starvation-free TM-liveness. Let $\alpha \cdot \pi$ be an execution of M, where π is a t-sequential t-complete execution of a transaction T_0. Then, there exists π^t, a prefix of π, such that T_0 protects $|Wset(T_0)|$ t-objects in $\alpha \cdot \pi^t$.*

A constant stall and constant expensive synchronization strict DAP opaque TM. Attiya *et al.* identified two common expensive synchronization patterns that frequently arise in the design of concurrent algorithms: *read-after-write (RAW) or atomic write-after-read (AWAR)* [8, 52] and showed 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*. RAW (read-after-write) or AWAR (atomic-write-after-read) patterns [8] capture the amount of "expensive synchronization", *i.e.*, the number of costly memory barriers or conditional primitives [2] incurred by the implementation in relaxed CPU architectures. The metric appears to be more practically relevant than simply counting the number of steps performed by a process, as it accounts for expensive cache-coherence operations or instructions like compare-and-swap.

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

Intuitively, the stall metric captures the fact that the time a process might have to spend before it applies a primitive on a base object can be proportional to the

number of processes that try to update the object concurrently. 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 α and β are execution fragments and $e_1 \cdots e_m$ is a maximal sequence of $m \geq 1$ consecutive nontrivial events by distinct 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 [7, 26].

Theorem 4 ([45]). *There exists a progressive, opaque and strict DAP TM implementation LP that provides wait-free TM-liveness, uses invisible reads, uses only read-write base objects, and for every execution E and transaction $T_k \in txns(E)$:*
- *T_k performs at most a single RAW, and*
- *every t-read operation invoked by T_k incurs $O(1)$ memory stalls in E, and*
- *every complete t-read operation invoked by T_k performs $O(|Rset(T_k)|)$ steps in E.*

Proof sketch. There exists a cheap progressive, opaque TM implementation *LP* in which every transaction performs at most a single RAW, every t-read operation incurs $O(1)$ memory stalls and maintains exactly one version of every t-object at any prefix of an execution. Moreover, the implementation is strict DAP and uses only read-write base objects.

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

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

The $write_k(X, v)$ implementation by process p_i simply stores the value v locally, deferring the actual updates to $tryC_k$. During $tryC_k$, process p_i attempts to obtain exclusive write access to every $X_j \in Wset(T_k)$. This is realized through the single-writer bits, which ensure that no other transaction may write to base objects v_j and L_j until T_k relinquishes its exclusive write access to $Wset(T_k)$. Specifically, process p_i writes 1 to each r_{ij}, then checks that no other process p_t has written 1 to any r_{tj} by executing a series of reads (incurring a single RAW). If there exists such a process that concurrently contends on write set of T_k, for each $X_j \in Wset(T_k)$, p_i writes 0 to r_{ij} and aborts T_k. If successful in obtaining exclusive write access to

TM-correctness	TM-liveness	Invisible reads	rmw primitives	Complexity
Strict serializability	WF		read-write	Impossible
Strict serializability			read-write, conditional	$\Omega(n \log n)$ RMRs
Opacity	starvation-free	yes	read-write	$O(1)$ RAW/AWAR

Table 2: Complexity bounds for strongly progressive TMs.

$Wset(T_k)$, p_i sets the bit L_j for each X_j in its write set. Implementation of $tryC_k$ now checks if any t-object in its read set is concurrently contended by another transaction and then validates its read set. If there is contention on the read set or validation fails (indicating the presence of a conflicting transaction), the transaction is aborted. If not, p_i writes the values of the t-objects to shared memory and relinquishes exclusive write access to each $X_j \in Wset(T_k)$ by writing 0 to each of the base objects L_j and r_{ij}.

Read-only transactions do not apply any nontrivial primitives. Any updating transaction performs at most a single RAW in the course of acquiring exclusive write access to the transaction's write set. Thus, every transaction performs $O(1)$ non-overlapping RAWs in any execution.

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

3.3 Strongly progressive TMs

We then turn our focus to *strongly progressive* TMs [33] that, in addition to progressiveness, ensure that *not all* concurrent transactions conflicting over a single data item abort.

A $\Omega(n \log n)$ lower bound on remote memory references. [47] showed that in any *strongly progressive* strictly serializable TM implementation that accesses the shared memory with read, write and conditional primitives, such as compare-and-swap and load-linked/store-conditional, the total number of *remote memory references* (RMRs) that take place in an execution of a progressive TM in which

n concurrent processes perform transactions on a single t-object might reach $\Omega(n \log n)$.

Modern shared memory CPU architectures employ a *memory hierarchy* [35]: a hierarchy of memory devices with different capacities and costs. Some of the memory is *local* to a given process while the rest of the memory is *remote*. Accesses to memory locations (*i.e.* base objects) that are *remote* to a given process are often orders of magnitude slower than a *local* access of the base object. Thus, the performance of concurrent implementations in the shared memory model may depend on the number of *remote memory references* made to base objects [6].

The RMR lower bound in [47] is obtained via a reduction to an analogous lower bound for mutual exclusion [10]. The reduction shows that any TM with the above properties can be used to implement a *deadlock-free* mutual exclusion, employing transactional operations on only one t-object and incurring a constant RMR overhead. The lower bound applies to RMRs in both the *cache-coherent (CC)* and *distributed shared memory (DSM)* models, and it appears to be the first RMR complexity lower bound for transactional memory.

Theorem 5 ([47]). *Any strictly serializable, strongly progressive TM implementation M that accesses a single t-object implies a* deadlock-free, *finite exit* mutual exclusion implementation L(M) *such that the RMR complexity of M is within a constant factor of the RMR complexity of L(M).*

Strongly progressive TMs from read-write primitives. Guerraoui *et al.* [33] proved the impossibility of implementing strongly progressive strictly serializable TMs providing *wait-free* TM-liveness from read-write base objects.

Theorem 6 ([33]). *It is impossible to implement strictly serializable strongly progressive TMs that provide* wait-free *TM-liveness (every t-operation returns a matching response within a finite number of steps) using only read and write primitives.*

[44] describes one means to circumvent this impossibility result: specifically, they prove the existence an opaque strongly progressive TM implementation from read-write base objects that provides starvation-free TM-liveness.

Theorem 7. *There exists a strongly progressive opaque TM implementation with starvation-free t-operations that uses invisible reads and employs at most four RAWs per transaction.*

3.4   On the cost of permissive opaque TMs

(Strongly) progressive TMs that allow a transaction to be aborted only on read-write conflicts have constant RAW/AWAR complexity. However, not aborting on

conflicts may not necessarily affect TM-correctness. Ideally, we would like to derive TM implementations that are *permissive* [30], in the sense that a transaction is aborted only if committing it would violate TM-correctness.

Kuznetsov *et al.* [44] establish a linear (in the transaction's data set size) separation between the worst-case transaction expensive synchronization complexity of strongly progressive TMs and *permissive* TMs that allow a transaction to abort only if committing it would violate opacity. Specifically, [44] show that an execution of a transaction in a *permissive opaque* TM implementation that provides starvation-free TM-liveness may require to perform at least one RAW/AWAR pattern *per* t-read.

Definition 2 (Permissiveness). *A TM implementation M is* permissive *with respect to TM-correctness C if for every history H of M such that H ends with a response r_k and replacing r_k with some $r_k \neq A_k$ gives a history that satisfies C, we have $r_k \neq A_k$.*

Therefore, permissiveness does not allow a transaction to abort, unless committing it would violate the execution's correctness.

[44] show that an execution of a transaction in a *permissive opaque* TM implementation (providing starvation-free TM-liveness) may require to perform at least one RAW/AWAR pattern *per* t-read.

Theorem 8 ([44]). *Let M be a permissive opaque TM implementation providing starvation-free TM-liveness. Then, for any $m \in \mathbb{N}$, M has an execution in which some transaction performs m t-reads such that the execution of each t-read contains at least one RAW or AWAR.*

Proof. Consider an execution E of M consisting of transactions T_1, T_2, T_3 as shown in Figure 3: T_3 performs a t-read of X_1, then T_2 performs a t-write on X_1 and commits, and finally T_1 performs a series of reads from objects $X_1, \ldots, X_m$. Since the implementation is permissive, no transaction can be forcefully aborted in E, and the only valid serialization of this execution is T_3, T_2, T_1. Note also that the execution generates a sequential history: each invocation of a t-operation is immediately followed by a matching response. Thus, since we assume starvation-freedom as a liveness property, such an execution exists.

We consider $read_1(X_k)$, $2 \leq k \leq m$ in execution E. Imagine that we modify the execution E as follows. Immediately after $read_1(X_k)$ executed by T_1 we add $write_3(X, v)$, and $tryC_3$ executed by T_3 (let $TC_3(X_k)$ denote the complete execution of $W_3(X_k, v)$ followed by $tryC_3$). Obviously, $TC_3(X_k)$ must return abort: neither T_3 can be serialized before T_1 nor T_1 can be serialized before T_3. On the other hand if $TC_3(X_k)$ takes place just before $read_1(X_k)$, then $TC_3(X_k)$ must return commit but $read_1(X_k)$ must return the value written by T_3. In other words, $read_1(X_k)$ and
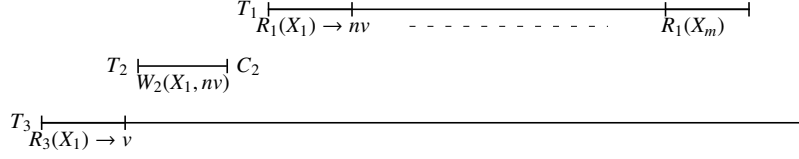
Figure 3: Execution E of a permissive, opaque TM: T_2 and T_3 force T_1 to perform a RAW/AWAR in each $R_1(X_k)$, $2 \le k \le m$

$TC_3(X_k)$ are *strongly non-commutative* [8]: both of them see the difference when ordered differently. As a result, intuitively, $read_1(X_k)$ needs to perform a RAW or AWAR to make sure that the order of these two "conflicting" operations is properly maintained. We formalize this argument below.

Consider a modification E' of E, in which T_3 performs $write_3(X_k)$ immediately after $read_1(X_k)$ and then tries to commit. In any serialization of E', T_3 must precede T_2 ($read_3(X_1)$ returns the initial value of X_1) and T_2 must precede T_1 to respect the real-time order of transactions. The execution of $read_1(X_k)$ does not modify base objects, hence, T_3 does not observe $read_1(X_k)$ in E'. Since M is permissive, T_3 must commit in E'. But since T_1 performs $read_1(X_k)$ before T_3 commits and T_3 updates X_k, we also have T_1 must precede T_3 in any serialization. Thus, T_3 cannot precede T_1 in any serialization—contradiction. Consequently, each $read_1(X_k)$ must perform a write to a base object.

Let π be the execution fragment that represents the complete execution of $read_1(X_k)$ and E^k, the prefix of E up to (but excluding) the invocation of $read_1(X_k)$.

Clearly, π contains a write to a base object. Let π_w be the first write to a base object in π. Thus, π can be represented as $\pi_s \cdot \pi_w \cdot \pi_f$. Suppose that π does not contain a RAW or AWAR. Consider the execution fragment $E^k \cdot \pi_s \cdot \rho$, where ρ is the complete execution of $TC_3(X_k)$ by T_3. Such an execution of M exists since π_s does not perform any base object write, hence, $E^k \cdot \pi_s \cdot \rho$ is indistinguishable to T_3 from $E^k \cdot \rho$.

Since, by our assumption, $\pi_w \cdot \pi_f$ contains no RAW, any read performed in $\pi_w \cdot \pi_f$ can only be applied to base objects previously written in $\pi_w \cdot \pi_f$. Since π_w is not an AWAR, $E^k \cdot \pi_s \cdot \rho \cdot \pi_w \cdot \pi_f$ is an execution of M since it is indistinguishable to T_1 from $E^k \cdot \pi$. In $E^k \cdot \pi_s \cdot \rho \cdot \pi_w \cdot \pi_f$, T_3 commits (as in ρ) but T_1 ignores the value written by T_3 to X_k. But there exists no serialization that justifies this execution—contradiction to the assumption that M is opaque. Thus, each $read_1(X_k)$, $2 \le k \le m$ must contain a RAW/AWAR.

Note that since all t-reads of T_1 are executed sequentially, all these RAW/AWAR patterns are pairwise non-overlapping, which completes the proof. □

The following result is a simple corollary to Theorem 8.

Corollary 9 ([16]). *There does not exist any permissive opaque TM implementation with invisible reads and starvation-free TM-liveness.*

4   Complexity of non-blocking TMs

We focus on TMs that avoid using locks and rely on non-blocking synchronization: a prematurely halted transaction cannot prevent other transactions from committing. Possibly the weakest non-blocking progress condition is obstruction-freedom [37, 40] stipulating that every transaction running in the absence of *step contention*, *i.e.*, not encountering steps of concurrent transactions, must commit. In fact, several early TM implementations [29, 38, 50, 61, 63] satisfied obstruction-freedom.

Let $\mathcal{OF}$ denote the class of non-blocking TMs that provide obstruction-free TM-progress (a transaction is allowed to abort only in executions that are not step contention-free) and *obstruction-free* (every t-operation must return a matching response within a finite number of steps in step contention-free executions) TM-liveness. Observe that there exists an execution exported by an obstruction-free TM, but not by any progressive TM and vice-versa. Consider a t-read X by a transaction T that runs step contention-free from a configuration that contains an incomplete write to X. Weak progressiveness does not preclude T from being aborted in such an execution. Obstruction-free TMs however, must ensure that T must complete its read of X without blocking or aborting in such executions. On the other hand, weak progressiveness requires two non-conflicting transactions to not be aborted even in executions that are not step contention-free; but this is not guaranteed by obstruction-freedom.

4.1   Lower bounds for obstruction-free TMs

On the cost of disjoint-access parallelism.   Complexity of obstruction-free TMs was first studied by Guerraoui and Kapalka [31, 33] who proved that they cannot provide strict DAP. However, it is possible to realize weaker than strict DAP variants of obstruction-free opaque TMs. For example, DSTM [38] satisfies RW DAP (and hence weak DAP), but not strict DAP.

Theorem 10 ([31]). *There does not exist any strict DAP strictly serializable TM implementation in $\mathcal{OF}$.*

The next result we survey focuses on strictly serializable TM implementations that satisfy two important properties: weak DAP and read invisibility. There exist weak DAP lock-based TM implementations that use invisible reads [21, 27]. In contrast, [45] establish that it is impossible to implement an obstruction-free

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

3:      r_{ij}, for each process p_i and t-object X_j

4:        single-writer bit

5:      L_j, for each t-object X_j

6:    *read$_k$(X_j)*:

7:      **if** $X_j \notin Rset(T_k)$ **then**

8:        $[ov_j, k_j] := read(v_j)$

9:        $Rset(T_k) := Rset(T_k) \cup \{X_j, [ov_j, k_j]\}$

10:        **if** $read(L_j) \neq 0$ **then**

11:          **Return** A_k

12:        **if** validate() **then**

13:          **Return** A_k

14:        **Return** ov_j

15:      **else**

16:        $[ov_j, \perp] := Rset(T_k).\text{locate}(X_j)$

17:        **Return** ov_j

18:    *write$_k$(X_j, v)*:

19:      $nv_j := v$

20:      $Wset(T_k) := Wset(T_k) \cup \{X_j\}$

21:      **Return** ok

22:    *tryC$_k$()*:

23:      **if** $|Wset(T_k)| = \emptyset$ **then**

24:        **Return** C_k

25:      locked := acquire($Wset(T_k)$)

26:      **if** $\neg$ locked **then**

27:        **Return** A_k

28:      **if** isAbortable() **then**

29:        release($Wset(T_k)$)

30:        **Return** A_k

31:      **for all** $X_j \in Wset(T_k)$ **do**

32:        $write(v_j, [nv_j, k])$

33:      release($Wset(T_k)$)

34:      **Return** C_k

35: **Function:** release(Q)**:**

36:      **for all** $X_j \in Q$ **do**

37:        $write(L_j, 0)$

38:      **for all** $X_j \in Q$ **do**

39:        $write(r_{ij}, 0)$

40:      **Return** ok

41: **Function:** acquire(Q)**:**

42:      **for all** $X_j \in Q$ **do**

43:        $write(r_{ij}, 1)$

44:      **if** $\exists X_j \in Q; t \neq k : read(r_{tj}) = 1$ **then**

45:        **for all** $X_j \in Q$ **do**

46:          $write(r_{ij}, 0)$

47:        **Return** *false*

48:      **for all** $X_j \in Q$ **do**

49:        $write(L_j, 1)$

50:      **Return** *true*

51: **Function:** isAbortable() **:**

52:      **if** $\exists X_j \in Rset(T_k) : X_j \notin Wset(T_k) \wedge read(L_j) \neq 0$ **then**

53:        **Return** *true*

54:      **if** validate() **then**

55:        **Return** *true*

56:      **Return** *false*

57: **Function:** validate() **:**

58:      **if** $\exists X_j \in Rset(T_k):[ov_j, k_j] \neq read(v_j)$ **then**

59:        **Return** *true*

60:      **Return** *false*

T_0 $\vdash\!\!\overset{R_0(Z) \to v}{\rule{3em}{0pt}}\!\!+\!\!\overset{W_0(X, nv)}{\rule{3em}{0pt}}\!\!+\!\!\overset{tryC_0}{\rule{3em}{0pt}}\!\!\dashv$ (event of T_0) $\underset{e}{\circ}$ T_1 $\vdash\!\!\underset{\text{new value}}{\overset{R_1(X) \to nv}{\rule{4em}{0pt}}}\!\!\dashv$

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

T_0 $\vdash\!\!\overset{R_0(Z) \to v}{\rule{3em}{0pt}}\!\!+\!\!\overset{W_0(X, nv)}{\rule{3em}{0pt}}\!\!+\!\!\overset{tryC_0}{\rule{3em}{0pt}}\!\!\dashv$ T_2 $\vdash\!\!\underset{\text{initial value}}{\overset{R_2(X) \to v}{\rule{3em}{0pt}}}\!\!\dashv$ (event of T_0) $\underset{e}{\circ}$ T_1 $\vdash\!\!\underset{\text{new value}}{\overset{R_1(X) \to nv}{\rule{3em}{0pt}}}\!\!\dashv$
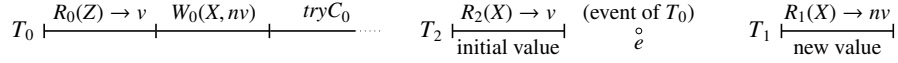
(b) T_1 returns new value of X since T_2 is invisible

T_0 $\vdash\!\!\overset{R_0(Z) \to v}{\rule{3em}{0pt}}\!\!+\!\!\overset{W_0(X, nv)}{\rule{3em}{0pt}}\!\!+\!\!\overset{tryC_0}{\rule{3em}{0pt}}\!\!\dashv$ T_3 $\vdash\!\!\underset{\text{write new value}}{\overset{W_3(Z, nv)}{\rule{3em}{0pt}}}\!\!\dashv$ T_2 $\vdash\!\!\underset{\text{initial value}}{\overset{R_2(X) \to v}{\rule{3em}{0pt}}}\!\!\dashv$ (event of T_0) $\underset{e}{\circ}$ T_1 $\vdash\!\!\underset{\text{new value}}{\overset{R_1(X) \to nv}{\rule{3em}{0pt}}}\!\!\dashv$
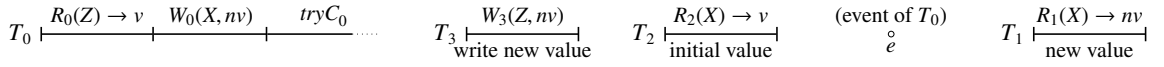
(c) By weak DAP and invisible reads, T_1 and T_2 do not observe the presence of T_3

Figure 4: Executions describing the proof sketch of Theorem 11; execution in 4c is not strictly serializable

TM that provides both weak DAP and read invisibility. Indeed, DSTM [38] and FSTM [29] are weak DAP, but use *visible* reads for aborting pending writing transactions.

Theorem 11 ([45]). *There does not exist a weak DAP strictly serializable TM implementation in $\mathcal{OF}$ that uses invisible reads.*

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

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

Since M uses invisible reads, the following execution exists: E' can be extended with the t-complete step contention-free execution of a transaction T_2 that reads the initial value v in X and commits, followed by the step e of T_0 after which transaction T_1 running step contention-free reads nv in X and commits. Moreover, this execution is indistinguishable to T_1 and T_2 from an execution in which the read set of T_0 is empty. Thus, we can modify this execution by inserting the step contention-free execution of a committed transaction T_3 that writes a new value to Z after E', but preceding T_2 in real-time order. Intuitively, by weak DAP,

transactions T_1 and T_2 cannot distinguish this execution from the original one in which T_3 does not participate.

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

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

A linear lower bound on memory stall complexity. [45] prove a linear (in n) lower bound for strictly serializable TM implementations in $\mathcal{OF}$ on the total number of *memory stalls* incurred by a single t-read operation.

Theorem 12 ([45]). *Every strictly serializable TM implementation $M \in \mathcal{OF}$ has a $(n-1)$-stall execution E for a t-read operation performed in E.*

Proof sketch. Inductively, for each $k \leq n-1$, construct a specific *k-stall execution* [26] in which some t-read operation by a process p incurs k stalls. In the k-stall execution, k processes are partitioned into disjoint subsets $S_1, \ldots, S_i$. The execution can be represented as $\alpha \cdot \sigma_1 \cdots \sigma_i$; α is p-free, where in each σ_j, $j = 1, \ldots, i$, p first runs by itself, then each process in S_j applies a *nontrivial* event on a base object b_j, and then p applies an event on b_j. Moreover, p does not detect step contention in this execution and, thus, must return a non-abort value in its t-read and commit in the solo extension of it. Additionally, it is guaranteed that in any extension of α by the processes other than $\{p\} \cup S_1 \cup S_2 \cup \ldots \cup S_i$, no nontrivial primitive is applied on a base object accessed in $\sigma_1 \cdots \sigma_i$.

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

Observe that, since there are at most n concurrent transactions, we cannot do better than $(n-1)$ stalls. Thus, the lower bound of Theorem 12 is tight.

RAW/AWAR complexity. [45] prove that opaque, RW DAP TM implementations in $\mathcal{OF}$ have executions in which some read-only transaction performs a linear (in n) number of non-overlapping RAWs or AWARs.

	Obstruction-free	Progressive LP
strict DAP	No	Yes
invisible reads+weak DAP	No	Yes
stall complexity of reads	$\Omega(n)$	$O(1)$
RAW/AWAR complexity	$\Omega(n)$	$O(1)$
read-write base objects, wait-free termination	No	Yes

Figure 5: Complexity gap between blocking and non-blocking TMs; n is the number of processes

Theorem 13. *Every RW DAP opaque TM implementation $M \in \mathcal{OF}$ has an execution E in which some read-only transaction $T \in txns(E)$ performs $\Omega(n)$ non-overlapping RAW/AWARs.*

Impossibility of obstruction-free TMs from read-write primitives. Guerraoui and Kapalka [31, 33] 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. An interesting open question is whether we can implement strict serializable TMs in $\mathcal{OF}$ using only read and write primitives.

4.2 Blocking versus non-blocking TMs

Some benefits of obstruction-free TMs, namely their ability to make progress even if some transactions prematurely fail, are not provided by progressive TMs. However, several papers [20,21,28] argued that lock-based TMs tend to outperform obstruction-free ones by allowing for simpler algorithms with lower overhead, and their inherent progress issues may be resolved using timeouts and *contention-managers* [60].

As highlighted in [21,28], obstruction-free TMs typically must forcefully abort pending conflicting transactions. This observation inspires the impossibility of invisible reads (Theorem 11). Typically, to detect the presence of a conflicting transaction and abort it, the reading transaction must employ a RAW or a read-modify-write primitive like *compare-and-swap*, motivating the linear lower bound on expensive synchronization (Theorem 13). Also, in obstruction-free TMs, a transaction may not wait for a concurrent inactive transaction to complete and, as a result, we may have an execution in which a transaction incurs a distinct stall due to a transaction run by each other process, hence the linear stall complexity (Theorem 12). Intuitively, since transactions in progressive TMs may abort themselves in case of conflicts, they can employ invisible reads and maintain constant stall and RAW/AWAR complexities.

Overcoming the lower bounds for obstruction-free TMs individually is comparatively easy. Say, TL [21] combines strict DAP with invisible reads, but it

is not read-write (for base object primitives), and it does not provide constant RAW/AWAR and stall complexities. However, the progressive TM *LP* overcomes most of the lower bounds known for obstruction-free TMs. Observe that the opaque implementation *LP*, (1) uses only read-write 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 per read operation. In contrast, from the lower bounds summarized in this survey we know that (i) no OF TM that provides wait-free transactional operations can be implemented using only read-write base objects; (ii) no OF TM can provide strict DAP; (iii) no weak DAP OF TM has invisible reads and (iv) no OF TM ensures a constant number of stalls incurred by a read operation. Finally, (v) no RW DAP *opaque* OF TM has constant RAW/AWAR complexity. Thus, (iv) and (v) exhibit a linear separation between blocking and non-blocking TMs w.r.t expensive synchronization and memory stall complexity, respectively.

The results are summarized and put in perspective in Figure 5 [45]. 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.

5   Lower bounds for partially non-blocking TMs

It is easy to see that *dynamic* TMs where the patterns in which transactions access t-objects are not known in advance do not allow for *wait-free* TMs [33], *i.e.*, every transaction must commit in a finite number of steps of the process executing it, regardless of the behavior of concurrent processes. Suppose that a transaction T_1 reads t-object X, then a concurrent transaction T_2 reads t-object Y, writes to X and commits, and finally T_2 writes to Y. Since T_1 has read the "old" value in X and T_2 has read the "old" value in Y, there is no way to commit T_1 and order the two transactions in a sequential execution. As this scenario can be repeated arbitrarily often, even the weaker guarantee of *local progress* that only requires that each transaction *eventually* commits if repeated sufficiently often, cannot be ensured by *any* strictly serializable TM implementation, regardless of the base objects it uses [14].[1]

Theorem 14 ([14]). *There does not exist any strictly serializable TM implementation that provides local progress.*

---

[1]Note that the counter-example would not work if we imagine that the data sets accessed by a transaction can be known in advance. However, we consider the conventional dynamic TM programming model.

But can we ensure that at least *some* transactions commit wait-free and what are the inherent costs? It is often argued that many realistic workloads are *read-dominated*: the proportion of read-only transactions is higher than that of updating ones, or read-only transactions have much larger data sets than updating ones [12, 34]. Therefore, it seems natural to require that read-only transactions commit wait-free. Moreover, we require that updating transaction provide only an extremely weak sequential TM-progress.

We denote by $\mathcal{RWF}$ the class of partially non-blocking TMs originally studied and motivated by Attiya *et al.* [11].

Definition 3 ([46]). *(The class $\mathcal{RWF}$) A TM implementation $M \in \mathcal{RWF}$ iff in its every execution:*
- *(*wait-free TM-progress for read-only transactions*) every read-only transaction commits in a finite number of its steps, and*
- *(*sequential TM-progress and sequential TM-liveness for updating transactions*) i.e., every transaction running step contention-free from a t-quiescent configuration, commits in a finite number of its steps.*

5.1 The space complexity of invisible reads

[46] prove that every strictly serializable TM implementation $M \in \mathcal{RWF}$ that uses invisible reads must keep unbounded sets of values for every t-object. To do so, for every $c \in \mathbb{N}$, construct an execution of M that *maintains at least c distinct values for every t-object.*

Definition 4 ([46]). *Let E be any execution of a TM implementation M. We say that E maintains c distinct values $\{v_1, \ldots, v_c\}$ of t-object X, if there exists an execution $E \cdot E'$ of M such that*
- *E' contains the complete executions of c t-reads of X and,*
- *for all $i \in \{1, \ldots, c\}$, the response of the ith t-read of X in E' is v_i.*

Theorem 15 ([46]). *Let M be any strictly serializable TM implementation in $\mathcal{RWF}$ that uses invisible reads, and $\mathcal{X}$, any set of t-objects. Then, for every $c \in \mathbb{N}$, there exists an execution E of M such that E maintains at least c distinct values of each t-object $X \in \mathcal{X}$.*

Proof. Let v_{0_ℓ} be the initial value of t-object $X_\ell \in \mathcal{X}$. For every $c \in \mathbb{N}$, we iteratively construct an execution E of M of the form depicted in Figure 6a. The construction of E proceeds in phases: there are at most $c - 1$ phases. For all $i \in \{0, \ldots c - 1\}$, we denote the execution after phase i as E_i which is defined as follows:
- E_0 is the complete step contention-free execution fragment α_0 of read-only transaction T_0 that performs $read_0(X_1) \rightarrow v_{0_1}$

Phase 0 | T_0 $\vdash\!\!\overset{R_0(X_1) \to v_{0_1}}{\rule{3cm}{0pt}}\!\!\dashv$

Phase 1 | T_1 $\vdash\!\!\overset{\forall X_\ell \in \mathcal{X}:\ \text{write } v_{1_\ell}}{\underset{T_1 \text{ commits}}{\rule{3cm}{0pt}}}\!\!\dashv$ T_2 $\vdash\!\!\overset{R_2(X_1) \to v_{1_1}}{\rule{3cm}{0pt}}\!\!\dashv$

Phase i | T_{2i-1} $\vdash\!\!\overset{\forall X_\ell \in \mathcal{X}:\ \text{write } v_{i_\ell}}{\underset{T_{2i-1} \text{ commits}}{\rule{3cm}{0pt}}}\!\!\dashv$ T_{2i} $\vdash\!\!\overset{R_{2i}(X_1) \to v_{i_1}}{\rule{3cm}{0pt}}\!\!\dashv$

$\downarrow$

extend to $c-1$ phases

(a) for all $i \in \{1, \ldots, c-1\}$, T_{2i-1} writes v_{i_ℓ} to each X_ℓ; $read_{2i}(X_1)$ must return v_{i_1}

T_0 $\vdash\!\!\overset{R_0(X_1) \to v_{0_1}}{\rule{2cm}{0pt}}\!\!\dashv$ $\vdash\!\!\overset{R_0(X_2) \to v_{0_2} \cdots R_0(X_\ell) \to v_{0_\ell} \cdots}{\rule{3.5cm}{0pt}}\!\!\dashv$

T_1 $\vdash\!\!\overset{\forall X_\ell \in \mathcal{X}:\ \text{write } v_{1_\ell}}{\underset{T_1 \text{ commits}}{\rule{2cm}{0pt}}}\!\!\dashv$ T_2 $\vdash\!\!\overset{R_2(X_1) \to v_{1_1}}{\rule{2cm}{0pt}}\!\!\dashv$ $\vdash\!\!\overset{R_2(X_2) \to v_{1_2} \cdots R_2(X_\ell) \to v_{1_\ell} \cdots}{\rule{3.5cm}{0pt}}\!\!\dashv$

T_{2i-1} $\vdash\!\!\overset{\forall X_\ell \in \mathcal{X}:\ \text{write } v_{i_\ell}}{\underset{T_{2i-1} \text{ commits}}{\rule{2cm}{0pt}}}\!\!\dashv$ T_{2i} $\vdash\!\!\overset{R_{2i}(X_1) \to v_{i_1}}{\rule{2cm}{0pt}}\!\!\dashv$ $\vdash\!\!\overset{R_{2i}(X_2) \to v_{i_2} \cdots R_{2i}(X_\ell) \to v_{i_\ell} \cdots}{\rule{3.5cm}{0pt}}\!\!\dashv$

$\downarrow$

extend to $c-1$ phases

(b) extend every read-only transaction T_{2i} in phase i with t-reads of $X_2, \ldots X_\ell, \ldots$; note that each $read_{2i}(X_\ell)$ must return v_{i_ℓ}
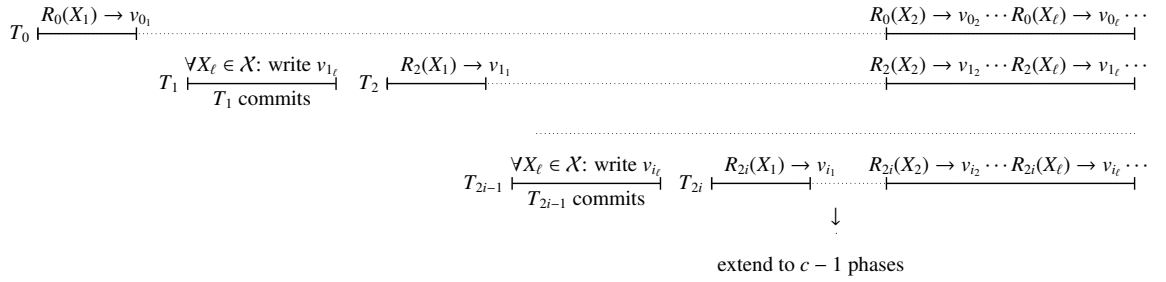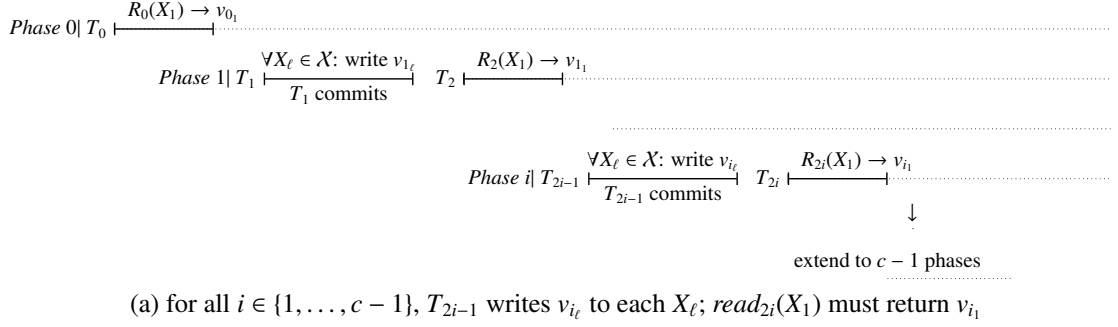
Figure 6: Executions in the proof of Theorem 15; execution in 6a must maintain c distinct values of every t-object

- for all $i \in \{1, \ldots, c-1\}$, E_i is defined to be an execution of the form $\alpha_0 \cdot \rho_1 \cdot \alpha_1 \cdots \rho_i \cdot \alpha_i$ such that for all $j \in \{1, \ldots, i\}$,
    - ρ_j is the t-complete step contention-free execution fragment of an updating transaction T_{2j-1} that, for all $X_\ell \in \mathcal{X}$ writes the value v_{j_ℓ} and commits
    - α_j is the complete step contention-free execution fragment of a read-only transaction T_{2j} that performs $read_{2j}(X_1) \to v_{j_1}$

Since read-only transactions are invisible, for all $i \in \{0, \ldots, c-1\}$, the execution fragment α_i does not contain any nontrivial events. Consequently, for all $i < j \le c-1$, the configuration after E_i is indistinguishable to transaction T_{2j-1} from a t-quiescent configuration and it must be committed in ρ_j (by sequential progress for updating transactions). Observe that, for all $1 \le j < i$, $T_{2j-1} \prec_E^{RT} T_{2i-1}$. Strict serializability of M now stipulates that, for all $i \in \{1, \ldots, c-1\}$, the t-read of X_1 performed by transaction T_{2i} in the execution fragment α_i must return the value v_{i_1} of X_1 as written by transaction T_{2i-1} in the execution fragment ρ_i (in any serialization, T_{2i-1} is the latest committed transaction writing to X_1 that precedes T_{2i}). Thus, M indeed has an execution E of the form depicted in Figure 6a.

Consider the execution fragment E' that extends E in which, for all $i \in \{0, \ldots, c-1\}$, read-only transaction T_{2i} is extended with the complete execution of the t-reads of every t-object $X_\ell \in \mathcal{X} \setminus \{X_1\}$ (depicted in Figure 6b).

We claim that, for all $i \in \{0, \ldots, c-1\}$, and for all $X_\ell \in \mathcal{X} \setminus \{X_1\}$, $read_{2i}(X_\ell)$ performed by transaction T_{2i} must return the value v_{i_ℓ} of X_ℓ written by transaction T_{2i-1} in the execution fragment ρ_i. Indeed, by wait-free progress, $read_i(X_\ell)$ must return a non-abort response in such an extension of E. Suppose by contradiction that $read_i(X_\ell)$ returns a response that is not v_{i_ℓ}. There are two cases:

- $read_{2i}(X_\ell)$ returns the value v_{j_ℓ} written by transaction T_{2j-1}; $j < i$. However, since for all $j < i$, $T_{2j} \prec_E^{RT} T_{2i}$, the execution is not strictly serializable—contradiction.
- $read_{2i}(X_\ell)$ returns the value v_{j_ℓ} written by transaction T_{2j}; $j > i$. Since $read_i(X_1)$ returns the value v_{i_1} and $T_{2i} \prec_E^{RT} T_{2j}$, there exists no such serialization—contradiction.

Thus, E maintains at least c distinct values of every t-object $X \in \mathcal{X}$. $\qquad\square$

Perelman *et al.* [55] considered the closely related (to $\mathcal{RWF}$) class of *mv-permissive* TMs: a transaction can only be aborted if it is an updating transaction that conflicts with another updating transaction. $\mathcal{RWF}$ is incomparable with the class of mv-permissive TMs. On the one hand, mv-permissiveness guarantees that read-only transactions never abort, but does not imply that they commit in a wait-free manner. On the other hand, $\mathcal{RWF}$ allows an updating transaction to abort in the presence of a concurrent read-only transaction, which is disallowed by mv-permissive TMs. Observe that, technically, mv-permissiveness is a blocking TM-progress condition, although when used in conjunction with wait-free TM-liveness, it is a partially non-blocking TM-progress condition that is strictly stronger than $\mathcal{RWF}$.

[55] proved that mv-permissive TMs cannot be *online space optimal*, *i.e.*, no mv-permissive TM can keep the minimum number of old object versions for any TM history. The result on the space complexity of implementations in $\mathcal{RWF}$ that use invisible reads (Theorem 15) is different since it proves that the implementation must maintain an unbounded number of versions of every t-object. The above proof technique can however be used to show that mv-permissive TMs considered in [55] should also maintain unbounded number of versions.

5.2 On the cost of disjoint-access parallelism

Kuznetsov *et al.* [46] prove that it is impossible to derive strictly serializable TM implementations in $\mathcal{RWF}$ which ensure that any two transactions accessing pairwise disjoint data sets can execute without contending on the same base object.

Theorem 16 ([46]). *There exists no strictly serializable strict DAP TM implementation in $\mathcal{RWF}$.*

Kuznetsov *et al.* [46] also prove a linear lower bound (in the size of the transaction's read set) on the number of RAWs or AWARs for weak DAP TM implementations in $\mathcal{RWF}$. Specifically, there exist executions in which each t-read operation of an arbitrarily long read-only transaction contains a RAW or an AWAR.

Theorem 17 ([46]). *Every strictly serializable weakly DAP TM implementation $M \in \mathcal{RWF}$ has, for all $m \in \mathbb{N}$, an execution in which some read-only transaction T_0 with $m = |Rset(T_0)|$ performs $\Omega(m)$ RAWs/AWARs.*

Since Theorem 17 implies that read-only transactions must perform nontrivial events, we have the following corollary that was proved directly in [11].

Corollary 18 ([11]). *There does not exist any strictly serializable weak DAP TM implementation $M \in \mathcal{RWF}$ that uses invisible reads.*

Attiya et al. [11] also considered a stronger "disjoint-access" property, called simply DAP, referring to the original definition proposed Israeli and Rappoport [42]. In DAP, two transactions are allowed to *concurrently access* (even for reading) the same base object only if they are disjoint-access. For an n-process DAP TM implementation, it is shown in [11] that a read-only transaction must perform at least $n - 3$ writes. The lower bound in Theorem 17 is strictly stronger than the one in [11], as it assumes only weak DAP, considers a more precise RAW/AWAR metric, and does not depend on the number of processes in the system. (Technically, the last point follows from the fact that the execution constructed in the proof of Theorem 17 uses only 3 concurrent processes.) Thus, the theorem subsumes the two lower bounds of [11] within a single proof.

Assuming starvation-free TM-liveness, [55] showed that implementing a weak DAP strictly serializable mv-permissive TM is impossible. The proof of this result is immediate from the analogous results for $\mathcal{RWF}$ in [11] and [46].

6 Hybrid Transactional Memory

If used carefully, HTM can be an extremely useful construct, and can significantly speed up and simplify concurrent implementations. At the same time, this powerful tool is not without its limitations: since HTMs are usually implemented on top of the cache coherence mechanism, hardware transactions have inherent *capacity constraints* on the number of distinct memory locations that can be accessed inside a single transaction. Moreover, all current proposals are *best-effort*, as they may

abort under imprecisely specified conditions (cache capacity overflow, interrupts *etc*). In brief, the programmer should not solely rely on HTMs.

Several HyTM schemes [17, 19, 43, 48] have been proposed to complement the fast, but best-effort nature of HTM with a slow, reliable software transactional memory (STM) backup. These proposals have explored a wide range of trade-offs between the overhead on hardware transactions, concurrent execution of hardware and software, and the provided progress guarantees. Early proposals for HyTM implementations [19, 43] shared some interesting features. First, transactions that do not conflict are expected to run concurrently, regardless of their types (software or hardware), à la progressiveness. Second, in addition to changing the values of transactional objects, hardware transactions usually employ *code instrumentation* techniques. Intuitively, instrumentation is used by hardware transactions to detect concurrency scenarios and abort in the case of contention. The number of instrumentation steps performed by these implementations within a hardware transaction is usually proportional to the size of the transaction's data set.

Recent work by Riegel *et al.* [58] surveyed the various HyTM algorithms to date, focusing on techniques to reduce instrumentation overheads in the frequently executed hardware fast-path. However, it is not clear whether there are fundamental limitations when building a HyTM with non-trivial concurrency between hardware and software transactions. In particular, what are the inherent instrumentation costs of building a HyTM, and what are the trade-offs between these costs and the provided *concurrency*, *i.e.*, the ability of the HyTM system to run software and hardware transactions in parallel?

Modelling HyTM. To address these questions, [4] proposes a model for hybrid TM systems which formally captures the notion of *cached* accesses provided by hardware transactions, and precisely defines instrumentation costs in a quantifiable way. [4] models a hardware transaction as a series of memory accesses that operate on locally cached copies of the variables, followed by a *cache-commit* operation. In case a concurrent transaction performs a (read-write or write-write) conflicting access to a cached object, the cached copy is invalidated and the hardware transaction aborts. The model for instrumentation is motivated by recent experimental evidence which suggests that the overhead on hardware transactions imposed by code which detects concurrent software transactions is a significant performance bottleneck [51]. In particular, a HyTM implementation imposes a logical partitioning of shared memory into *data* and *metadata* locations. Intuitively, metadata is used by transactions to exchange information about contention and conflicts while data locations only store the *values* of data items read and updated within transactions. [4] quantifies instrumentation cost by measuring the number of accesses to *metadata objects* which transactions perform. All known HyTM

proposals, such as *HybridNOrec* [17, 57], *PhTM* [48] and others [19, 43] avoid co-locating the data and metadata within a single base object.

Complexity. Once this general model is in place, Alistarh *et al.* [4] derive two lower bounds on the cost of implementing a HyTM. First, they show that some instrumentation is necessary in a HyTM implementation even if we only intend to provide *sequential* progress, where any transaction is only guaranteed to commit if it runs in the absence of concurrency.

Theorem 19 ([4]). *There does not exist a strictly serializable uninstrumented HyTM implementation that ensures sequential TM-progress and TM-liveness.*

Second, [4] prove that any progressive HyTM implementation providing *obstruction-free liveness* (every operation running *solo* returns some response) and has executions in which an arbitrarily long read-only hardware transaction running in the absence of concurrency *must* access a number of distinct metadata objects proportional to the size of its data set.

Theorem 20 ([4]). *Let $\mathcal{M}$ be any progressive, opaque HyTM implementation that provides OF TM-liveness. For every $m \in \mathbb{N}$, there exists an execution E in which some fast-path read-only transaction $T_k \in txns(E)$ satisfies either (1) $Dset(T_k) \leq m$ and T_k incurs a capacity abort in E or (2) $Dset(T_k) = m$ and T_k accesses $\Omega(m)$ distinct metadata base objects in E.*

The proof of the above theorem proceeds inductively. Start with a sequential execution in which a "large" set S_m of read-only hardware transactions, each accessing m distinct data items and m distinct metadata memory locations, run after an execution E_m. We then construct execution E_{m+1}, an extension of E_m, which forces at least half of the transactions in S_m to access a *new* metadata base object when reading a new $(m+1)^{th}$ data item, running after E_{m+1}. The technical challenge, and the key departure from prior work on STM lower bounds, *e.g.* [11, 31, 33], is that hardware transactions practically possess "automatic" conflict detection, aborting on contention. This is in contrast to STMs, which must take steps to detect contention on memory locations.

Algorithms. The inherent high instrumentation costs of early HyTM designs, stimulated more recent HyTM schemes [17, 48, 51, 58] to sacrifice progressiveness for *constant* instrumentation cost (*i.e.*, not depending on the size of the transaction). In the past few years, Dalessandro *et al.* [17] and Riegel *et al.* [58] have proposed HyTMs based on the efficient NOrec STM [18]. These HyTMs schemes do not guarantee any parallelism among transactions; only sequential progress is ensured. Despite this, they are among the best-performing HyTMs to date due to the limited

instrumentation in hardware transactions. Therefore, the cost of avoiding the linear lower bound for progressive implementations is that hardware transactions may be aborted by non-conflicting software ones.

7   Research directions and open questions

Weak TM-correctness. In this survey, we focussed on TM implementations providing the TM-correctness properties of opacity or the weaker strict serializability. However, one may observe that as long as committed transactions constitute a serial execution and every transaction witnesses a consistent state, the execution can be considered "safe": no run-time error that cannot occur in a serial execution can happen. TM-correctness properties like virtual-world consistency (VWC) [41] and transactional memory specification (TMS1) [25] ensure strict serializability, but are strictly weaker than opacity. Are TM implementations that satisfy VWC or TMS1, but not opacity subject to the lower bounds surveyed in this paper? For instance, it is easy to see that the lower bound of Theorem 8 on the complexity of permissive opaque TMs is not subject to permissive VWC TMs [16]. Furthermore, [16] described a permissive VWC TM implementation that ensures that t-read operations do not perform nontrivial primitives, but the tryCommit invoked by a read-only transaction perform a linear (in the size of the transaction's data set) number of RAW/AWARs.

Bushkov et al. [13] improved on the impossibility result in [31] and showed that a variant of strict DAP cannot be combined with obstruction-free TM-progress, even if a weaker (than strictly serializability) TM-correctness property is assumed.

Peluso et al. [54] study the complexity of TM implementations in the class $\mathcal{RWF}$ and show that deriving DAP implementations is impossible even if the TM-correctness assumed is weaker than strict serializability.

Exploring the complexity of STM and HyTM implementations satisfying the TM-correctness properties of VWC and TMS1 as well as properties weaker than strict serializability opens up several open questions and research directions.

HyTM models and complexity. Recent work has investigated alternatives to HyTMs that rely on STM fallback, such as *sandboxing* [3, 15] or *hardware-accelerated* STM [59, 62], and the use of both direct *and* cached accesses within the same hardware transaction to reduce instrumentation overhead [57, 58]. Another recent approach proposed *reduced hardware transactions* [51], where a part of the slow-path is executed using a short fast-path transaction, which allows to partially eliminate instrumentation from the hardware fast-path.

Verifying the correctness and understanding the complexity of these protocols is an important research direction as is identifying techniques for automatically

deploying the best TM implementation for a given workload [22] and scheduling techniques for HyTMs [23].

References

[1] Advanced Synchronization Facility Proposed Architectural Specification, March 2009. `http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf`.

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

[3] Y. Afek, A. Levy, and A. Morrison. Software-improved hardware lock elision. In *PODC*. ACM, 2014.

[4] D. Alistarh, J. Kopinsky, P. Kuznetsov, S. Ravi, and N. Shavit. Inherent limitations of hybrid transactional memory. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 185–199, 2015.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, Oct. 1985.

[6] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

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

[8] 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.

[9] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety and deferred update in transactional memory. In R. Guerraoui and P. Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *Lecture Notes in Computer Science*, pages 50–71. Springer International Publishing, 2015.

[10] H. Attiya, D. Hendler, and P. Woelfel. Tight rmr lower bounds for mutual exclusion and other problems. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 447–447, New York, NY, USA, 2008. ACM.

[11] 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.

[12] H. Attiya and A. Milani. Transactional scheduling for read-dominated workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] V. Bushkov, D. Dziuma, P. Fatourou, and R. Guerraoui. The pcl theorem: Transactions cannot be parallel, consistent and live. In *SPAA*, pages 178–187, 2014.

[14] V. Bushkov, R. Guerraoui, and M. Kapalka. On the liveness of transactional memory. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 9–18, New York, NY, USA, 2012. ACM.

[15] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop*. ACM, 2014.

[16] T. Crain, D. Imbs, and M. Raynal. Read invisibility, virtual world consistency and permissiveness are compatible. Research Report, ASAP - INRIA - IRISA - CNRS : UMR6074 - INRIA - Institut National des Sciences Appliquées de Rennes - Université de Rennes I, 11 2010.

[17] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In R. Gupta and T. C. Mowry, editors, *ASPLOS*, pages 39–52. ACM, 2011.

[18] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, Jan. 2010.

[19] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, Oct. 2006.

[20] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

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

[22] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano. Proteustm: Abstraction meets performance in transactional memory. *SIGOPS Oper. Syst. Rev.*, 50(2):757–771, Mar. 2016.

[23] N. Diegues, P. Romano, and S. Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 224–233, New York, NY, USA, 2015. ACM.

[24] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele, Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 216–224, New York, NY, USA, 2004. ACM.

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

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

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

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

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

[30] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC*, pages 305–319, 2008.

[31] 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.

[32] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. *SIGPLAN Not.*, 44(1):404–415, Jan. 2009.

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

[34] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: A benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, Mar. 2007.

[35] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

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

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

[38] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[39] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[40] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[41] D. Imbs and M. Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theor. Comput. Sci.*, 444, July 2012.

[42] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*, pages 151–160, 1994.

[43] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[44] P. Kuznetsov and S. Ravi. On the cost of concurrency in transactional memory. In *OPODIS*, pages 112–127, 2011.

[45] P. Kuznetsov and S. Ravi. Grasping the gap between blocking and non-blocking transactional memories. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 232–247, 2015.

[46] P. Kuznetsov and S. Ravi. On partial wait-freedom in transactional memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, page 10, 2015.

[47] P. Kuznetsov and S. Ravi. Progressive transactional memory in time and space. In *Parallel Computing Technologies - 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 - September 4, 2015, Proceedings*, pages 410–425, 2015.

[48] Y. Lev, M. Moir, and D. Nussbaum. Phtm: Phased transactional memory. In *In Workshop on Transactional Computing (Transact), 2007. research.sun.com/scalable/pubs/ TRANSACT2007PhTM.pdf*.

[49] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[50] V. J. Marathe, W. N. S. Iii, and M. L. Scott. Adaptive software transactional memory. In *In Proc. of the 19th Intl. Symp. on Distributed Computing*, pages 354–368, 2005.

[51] A. Matveev and N. Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2013.

[52] P. E. McKenney. Memory barriers: a hardware view for software hackers. Linux Technology Center, IBM Beaverton, June 2010.

[53] M. Ohmacht. Memory Speculation of the Blue Gene/Q Compute Chip, 2011. `http://wands.cse.lehigh.edu/IBM_BQC_PACT2011.ppt`.

[54] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 217–226, New York, NY, USA, 2015. ACM.

[55] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC*, pages 16–25, 2010.

[56] J. Reinders. Transactional Synchronization in Haswell, 2012. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`.

[57] T. Riegel. Software Transactional Memory Building Blocks. 2013.

[58] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 53–64. ACM, 2011.

[59] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[60] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[61] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[62] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Non-blocking transactions without indirection using alert-on-update. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 210–220, New York, NY, USA, 2007. ACM.

[63] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. Nztm: Nonblocking zero-indirection transactional memory. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 204–213, New York, NY, USA, 2009. ACM.