Distrib. Comput. (2018) 31:167–185
https://doi.org/10.1007/s00446-017-0305-3

Inherent limitations of hybrid transactional memory

Dan Alistarh[1] · Justin Kopinsky[2] · Petr Kuznetsov[3] · Srivatsan Ravi[4] · Nir Shavit[5]

Received: 29 December 2015 / Accepted: 7 June 2017 / Published online: 23 June 2017

Abstract Several hybrid transactional memory (HyTM) schemes have recently been proposed to complement the fast, but best-effort nature of hardware transactional memory with a slow, reliable software backup. However, the costs of providing concurrency between hardware and software transactions in HyTM are still not well understood. In this paper, we propose a general model for HyTM implementations, which captures the ability of hardware transactions to buffer memory accesses. The model allows us to formally quantify and analyze the amount of overhead (instrumentation) caused by the potential presence of software transactions. We prove that (1) it is impossible to build a strictly serializable HyTM implementation that has both uninstrumented reads and writes, even for very weak progress guarantees, and (2) the instrumentation cost incurred by a hardware transaction in any progressive opaque HyTM is linear in the size of the transaction's data set. We further describe two implementations which exhibit optimal instrumentation costs for two different progress conditions. In sum, this paper proposes the first formal HyTM model and captures for the first time the trade-off between the degree of hardware-software TM concurrency and the amount of instrumentation overhead.

Keywords hardware transactional memory · Instrumentation · Lower bounds

An earlier version of this work appeared in the Proceedings of the 2015 International Symposium on Distributed Computing (DISC 2015) [5].

✉ Srivatsan Ravi
srivatsr@usc.edu

Dan Alistarh
dan.alistarh@microsoft.com

Justin Kopinsky
jkopin@mit.edu

Petr Kuznetsov
petr.kuznetsov@telecom-paristech.fr

Nir Shavit
shanir@csail.mit.edu

1 Microsoft Research Cambridge, 21 Station Road, Cambridge CB1 1BB, UK

2 MIT Computer Science and Artificial Intelligence Laboratory, 77 Massachusetts Avenue, 32-G604, Cambridge, MA 02139, USA

3 LTCI, Télécom ParisTech, Université Paris-Saclay, C213-2, 46 Rue Barrault, 75013 Paris, France

4 Information Sciences Institute, University of Southern California, Room 1102, 4676 Admiralty Way, Marina del Rey, CA 90292, USA

5 MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, 32-G622, Cambridge, MA 02139, USA

1 Introduction

1.1 Hybrid transactional memory

Since its introduction by Herlihy and Moss [29], *Transactional Memory (TM)* has been a tool with tremendous promise. It is therefore not surprising that the recently introduced Hardware Transactional Memory (HTM) implementations [1,37,39] have been eagerly anticipated and scrutinized by the community.

Early experience with programming HTM, e.g. [4,16, 18,31,45], paints an interesting picture: if used carefully, HTM can significantly speed up and simplify concurrent implementations. At the same time, it 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. In brief, the programmer should not solely rely on HTMs.

Several *Hybrid Transactional Memory (HyTM)* schemes [13,15,32,35] 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 HyTM proposals [15,32] share interesting features. First, transactions that do not conflict on the *data items* they access are expected to run concurrently, regardless of their type (software or hardware). This property is referred to as *progressiveness* [24] and is believed to allow for higher parallelism. Second, hardware transactions usually employ *code instrumentation* techniques. Intuitively, instrumentation is used by hardware transactions to detect concurrency scenarios and abort in the case of data conflicts.

Reducing instrumentation in the frequently executed hardware fast-path is key to efficiency. In particular, recent work by Riegel et al. [41] surveys a series of techniques to reduce instrumentation. Despite considerable algorithmic work on HyTM, there is currently no formal basis for specifying and understanding the cost of building HyTMs with non-trivial concurrency. In particular, what are the inherent instrumentation costs of building a HyTM? What are the trade-offs between these costs and the ability of the HyTM system to run software and hardware transactions in parallel?

1.2 Modelling HyTM

To address these questions, we propose the first model for hybrid TM systems which formally captures the notion of *cached* accesses provided by hardware transactions, and defines instrumentation costs in a precise, quantifiable way.

Specifically, we model a hardware transaction as a series of memory accesses that operate on locally cached copies of the memory locations, followed by a *cache-commit* operation. In case a concurrent (hardware or software) transaction performs a (read-write or write-write) conflicting access to a cached base object, the cached copy is invalidated and the hardware transaction aborts. Thus, detecting contention on memory locations is provided "automatically" to code running inside hardware transactions.

Further, we notice that 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. Recent experimental evidence [36] suggests that the overhead imposed by accessing metadata, and in particular code to detect concurrent software transactions, is a significant performance bottleneck. Therefore, we quantify instrumentation cost by measuring the number of accesses to *metadata* memory locations which transactions perform. Our framework captures all known HyTM proposals which combine HTMs with an STM fallback [13,15,32,35,40].

1.3 The cost of concurrency

We then explore the implications of our model. The first, immediate application is an impossibility result showing that instrumentation *is necessary* in a HyTM implementation, even if we only provide *sequential* progress, *i.e.*, if a transaction is only guaranteed to commit if it runs in isolation.

The second application concerns the *instrumentation overhead* of progressive HyTM schemes, which constitutes our main technical contribution. We prove that any progressive HyTM, satisfying reasonable livenesss guarantees, must, in certain executions, force read-only transactions to access a *linear* (in the size of their data sets) number of metadata memory locations, even in the absence of contention.

Our proof technique is interesting in its own right. Inductively, we 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.* [9,21,25], 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.

This linear lower bound is tight. We match it with an algorithm which, additionally, allows for uninstrumented writes, *invisible reads* and is provably *opaque* [25]. To the best of our knowledge, this is the first formal proof of correctness of a HyTM algorithm.

1.4 Low-instrumentation HyTM

Our main lower bound result shows that there are high inherent instrumentation costs to progressive HyTM designs [15, 32]. Interestingly, some recent HyTM schemes [13,35,36, 41] sacrifice progressiveness for *constant* instrumentation cost (*i.e.*, not depending on the size of the data set). Instead, only sequential progress is ensured. (Despite this fact, these schemes perform well due to the limited instrumentation in hardware transactions.)

We extend these schemes to provide an upper bound for non-progressive *low-instrumentation* HyTMs. We present a HyTM with invisible reads *and* uninstrumented hardware writes which guarantees that a hardware transaction accesses at most *one* metadata object in the course of its execution. Software transactions are mutually progressive, while hardware transactions are guaranteed to commit only if they do not run concurrently with an updating software transaction. This algorithm shows that, by abandoning progressiveness, the instrumentation costs of HyTM can be reduced to the bare minimum required by our first impossibility result. In other words, the cost of avoiding the linear instrumentation lower bound is that hardware transactions may be aborted by non-conflicting software ones.

1.5 Roadmap

The rest of the paper is organized as follows. Section 2 introduces the basic TM model and definitions. Section 3 presents our first contribution: a formal model for HyTM implementations. Section 4 formally defines instrumentation and proves the impossibility of implementing uninstrumented HyTMs. Section 5 establishes a linear lower bound on metadata accesses for progressive HyTMs while Sect. 6 describes our instrumentation-optimal opaque HyTM implementations. Section 7 presents the related work and Sect. 8 concludes the paper.

2 Preliminaries

2.1 Transactional memory (TM)

A *transaction* is a sequence of *transactional operations* (or *t-operations*), reads and writes, performed on a set of *transactional objects* (*t-objects*). A TM *implementation* provides a set of concurrent *processes* with deterministic algorithms that implement reads and writes on t-objects using a set of *base objects*. More precisely, for each transaction T_k, a TM implementation must support the following t-operations: $read_k(X)$, where X is a t-object, that returns a value in a domain V or a special value $A_k \notin V$ (*abort*), $write_k(X, v)$, for a value $v \in V$, that returns *ok* or A_k, and $tryC_k$ that returns $C_k \notin V$ (*commit*) or A_k. Note that a TM interface may additionally provide a $start_k$ t-operation that returns *ok* or A_k, which is the first t-operation transaction T_k must invoke, or a $tryA_k$ t-operation that returns A_k. However, the actions performed inside the $start_k$ may be performed as part of the first t-operation performed by the transaction.

2.2 Configurations and executions

A *configuration* of a TM implementation specifies the state of each base object and each process. In the *initial* configuration, each base object has its initial value and each process is in its initial state. An *event* (or *step*) of a transaction invoked by some process is an invocation of a t-operation, a response of a t-operation, or an atomic *primitive* operation applied to base object along with its response. An *execution fragment* is a (finite or infinite) sequence of events $E = e_1, e_2, \ldots$. An *execution* of a TM implementation $\mathcal{M}$ is an execution fragment where, informally, each event respects the specification of base objects and the algorithms specified by $\mathcal{M}$. In the next section, we define precisely how base objects should behave in a hybrid model combining direct memory accesses with *cached* accesses (hardware transactions).

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.

For any finite execution E and execution fragment E', $E \cdot E'$ denotes the concatenation of E and E' and we say that $E \cdot E'$ is an *extension* of E. For every transaction identifier k, $E|k$ denotes the subsequence of E restricted to events of transaction T_k. If $E|k$ is non-empty, we say that T_k *participates* in E, and let $txns(E)$ denote the set of transactions that participate in E. Two executions E and E' are *indistinguishable* to a set $\mathcal{T}$ of transactions, if for each transaction $T_k \in \mathcal{T}$, $E|k = E'|k$.

2.3 Complete and incomplete transactions

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. 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 E.

2.4 Contention

We assume that base objects are accessed with *read-modify-write* (rmw) primitives [19,27]. A rmw primitive $\langle g, h\rangle$ applied to a base object atomically updates the value of the object with a new value, which is a function $g(v)$ of the old value v, and returns a response $h(v)$. A rmw primitive event on a base object is *trivial* if, in any configuration, its application does not change the state of the object. Otherwise, it is called *nontrivial*.

Events e and e' of an execution E *contend* on a base object b if they are both primitives on b in E and at least one of them is nontrivial.

In a configuration C after an execution E, every incomplete transaction T has exactly one *enabled* event in C, which is the next event T will perform according to the TM implementation. We say that a transaction T is *poised to apply an event e after E* if e is the next enabled event for T in E. We say that transactions T and T' *concurrently contend on b in E* if they are each poised to apply contending events on b after E.

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

2.5 TM correctness

A *history exported* by an execution fragment E is the subsequence of E consisting of only the invocation and response events of t-operations. Let H_E denote the history exported by an execution E. Two histories H and H' are *equivalent* if $txns(H) = txns(H')$ and for every transaction $T_k \in txns(H)$, $H|k = H'|k$. For any two transactions $T_k, T_m \in txns(E)$, we say that T_k *precedes* T_m in the *real-time order* of E ($T_k \prec_E^{RT} T_m$) 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 precedes T_m nor T_m precedes T_k in real-time order, then T_k and T_m are *concurrent* in E. An execution E is *sequential* if every invocation of a t-operation is either the last event in H or is immediately followed by a matching response, where H is the history exported by E. An execution E is *t-sequential* if there are no concurrent transactions in E.

Let E be a t-sequential execution. For every operation $read_k(X)$ in E, we define the *latest written value* of X as follows: (1) If T_k contains a $write_k(X, v)$ preceding $read_k(X)$, then the latest written value of X is the value of the latest such write to X. (2) Otherwise, if E contains a $write_m(X, v)$, 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 $read_k(X)$ is *legal* in a t-sequential execution E if it returns the latest written value of X, and E is *legal* if every $read_k(X)$ in H that does not return A_k is legal in E.

For a history H, a *completion of* H, denoted $\bar{H}$, is a history derived from H as follows:

1. for every incomplete t-operation op_k that is a $read_k \vee write_k$ of $T_k \in txns(H)$ in H, insert A_k somewhere after the last event of T_k in E; otherwise if $op_k = tryC_k$, insert A_k or C_k somewhere after the last event of T_k
2. for every complete transaction T_k in the history derived in (1) that is not t-complete, insert $tryC_k \cdot A_k$ after the last event of transaction T_k.

Definition 1 (Opacity and strict serializability) A finite history H is *opaque* if there is a legal t-complete t-sequential history S, such that for any two transactions $T_k, T_m \in txns(H)$, if $T_k \prec_H^{RT} T_m$, then $T_k \prec_S^{RT} T_m$, and S is equivalent to a completion of H [25].

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

2.6 TM-liveness

A liveness property specifies the conditions under which a t-operation must return. A TM implementation provides *wait-free (WF)* TM-liveness if it ensures that every t-operation returns in a finite number of its steps. A weaker property of *obstruction-freedom (OF)* ensures that every operation running step contention-free returns in a finite number of its own steps. The weakest property we consider here is *sequential* TM-liveness that only guarantees that t-operations running in the absence of concurrent transactions returns in a finite number of its steps.

3 Hybrid transactional memory (HyTM)

3.1 Direct accesses and cached accesses

We now describe the execution model of a *Hybrid Transactional Memory (HyTM)* implementation. In our HyTM model, every base object can be accessed with two kinds of primitives, *direct* and *cached*.

In a direct access, the rmw primitive operates on the memory state: the direct-access event atomically reads the value of the object in the shared memory and, if necessary, modifies it.

In a cached access performed by a process i, the rmw primitive operates on the *cached* state recorded in process i's *tracking set* τ_i. One can think of τ_i as the *L1 cache* of process i. A *hardware transaction* is a series of cached rmw primitives performed on τ_i followed by a *cache-commit* primitive.

More precisely, τ_i is a set of triples (b, v, m) where b is a base object identifier, v is a value, and $m \in \{shared, exclusive\}$ is an access *mode*. The triple (b, v, m) is added to the tracking set when i performs a cached rmw access of b, where m is set to *exclusive* if the access is nontrivial, and to *shared* otherwise. We assume that there exists some constant *TS* (representing the size of the L1 cache) such that the condition $|\tau_i| \leq TS$ must always hold; this condition will be enforced by our model. A base object b is *present* in τ_i with mode m if $\exists v, (b, v, m) \in \tau_i$.

A trivial (resp. nontrivial) cached primitive $\langle g, h \rangle$ applied to b by process i first checks the condition $|\tau_i| = TS$ and if so, it sets $\tau_i = \emptyset$ and immediately returns $\perp$ (we call this event a *capacity abort*). We assume that *TS* is large enough so that no transaction with data set of size 1 can incur a capacity abort. If the transaction does not incur a capacity abort, the process checks whether b is present in exclusive (resp. any) mode in τ_j for any $j \neq i$. If so, τ_i is set to $\emptyset$ and the primitive returns $\perp$. Otherwise, the triple $(b, v, shared)$ (resp. $(b, g(v), exclusive)$) is added to τ_i, where v is the most recent cached value of b in τ_i (in case b was previously accessed by i within the current hardware transaction) or the value of b in the current memory configuration, and finally $h(v)$ is returned.

A tracking set can be *invalidated* by a concurrent process: if, in a configuration C where $(b, v, exclusive) \in \tau_i$ (resp. $(b, v, shared) \in \tau_i$), a process $j \neq i$ applies any primitive (resp. any *nontrivial* primitive) to b, then τ_i becomes *invalid* and any subsequent cached primitive invoked by i sets τ_i to $\emptyset$ and returns $\perp$. We refer to this event as a *tracking set abort*.

Finally, the *cache-commit* primitive issued by process i in configuration C with a valid τ_i does the following: for each base object b such that $(b, v, exclusive) \in \tau_i$, the value of b in C is updated to v. Finally, τ_i is set to $\emptyset$ and the primitive returns *commit*.

Note that HTM may also abort spuriously, or because of unsupported operations [39]. The first cause can be modelled probabilistically in the above framework, which would not however significantly affect our claims and proofs, except make for a more cumbersome presentation. Also, our lower bounds are based exclusively on executions containing t-reads and t-writes. Therefore, since our primary focus in this paper are lower bounds, we only consider contention and capacity aborts.

3.2 Slow-path and fast-path transactions

In the following, we partition HyTM transactions into *fast-path transactions* and *slow-path transactions*. Practically, two separate algorithms (fast-path one and slow-path one) are provided for each t-operation.

A slow-path transaction models a regular software transaction. An event of a slow-path transaction is either an invocation or response of a t-operation, or a rmw primitive on a base object.

A fast-path transaction essentially encapsulates a hardware transaction. An event of a fast-path transaction is either an invocation or response of a t-operation, a cached primitive on a base object, or a *cache-commit*: *t-read* and *t-write* are only allowed to contain cached primitives, and *tryC* consists of invoking *cache-commit*. Furthermore, we assume that a fast-path transaction T_k returns A_k as soon an underlying cached primitive or *cache-commit* returns $\perp$. Figure 1 depicts such a scenario illustrating a tracking set abort: fast-path transaction T_2 executed by process p_2 accesses a base

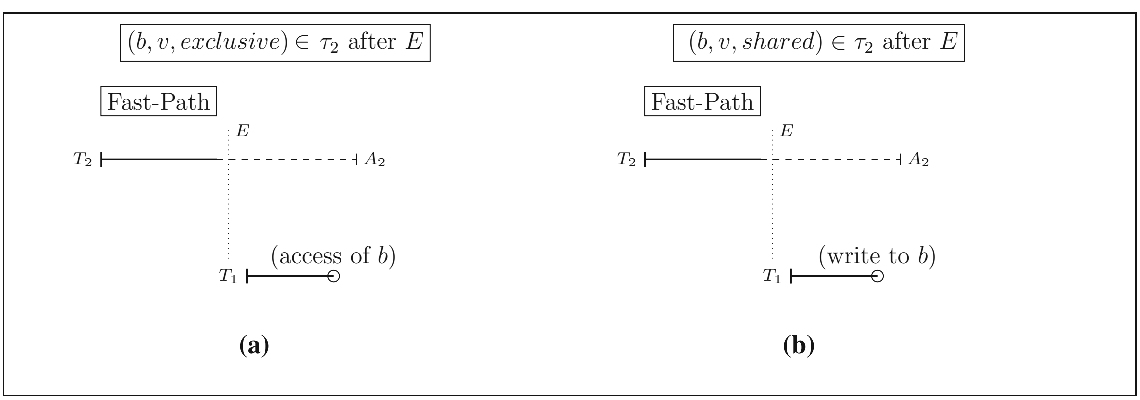

Fig. 1 Tracking set aborts in fast-path transactions **a** τ_2 is invalidated by (fast-path or slow-path) transaction T_1's access of base object b **b** τ_2 is invalidated by (fast-path or slow-path) transaction T_1's write to base object b

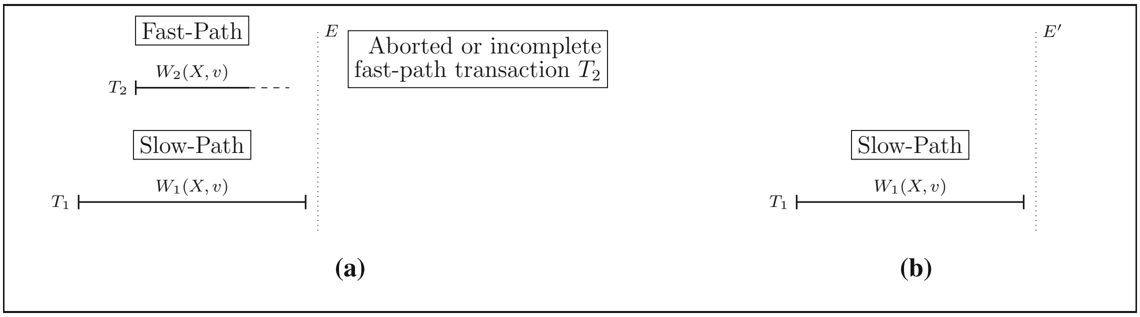

Fig. 2 Execution E in Fig. 2a is indistinguishable to T_1 from the execution E' in Fig. 2b

object b in shared (and resp. exclusive) mode and it is added to its tracking set τ_2. Immediately after the access of b by T_2, a concurrent transaction T_1 applies a nontrivial primitive to b (and resp. accesses b). Thus, the tracking set of p_2 is invalidated and T_2 must be aborted in any extension of this execution.

We provide two key observations on this model regarding the interactions of non-committed fast path transactions with other transactions. Let E be any execution of a HyTM implementation $\mathcal{M}$ in which a fast-path transaction T_k is either t-incomplete or aborted. Then the sequence of events E' derived by removing all events of $E|k$ from E is an execution of $\mathcal{M}$. Moreover:

Observation 1 *To every slow-path transaction $T_m \in txns(E)$, E is indistinguishable from E'.*

Observation 2 *If a fast-path transaction $T_m \in txns(E) \setminus \{T_k\}$ does not incur a tracking set abort in E, then E is indistinguishable to T_m from E'.*

Intuitively, these observations say that fast-path transactions which are not yet committed are invisible to slow-path transactions, and can communicate with other fast-path transactions only by incurring their tracking-set aborts. Figure 2 illustrates Observation 1: a fast-path transaction T_2 is concurrent to a slow-path transaction T_1 in an execution E. Since T_2 is t-incomplete or aborted in this execution, E is indistinguishable to T_1 from an execution E' derived by removing all events of T_2 from E. Analogously, to illustrate Observation 2, if T_1 is a fast-path transaction that does not incur a tracking set abort in E, then E is indistinguishable to T_1 from E'.

4 Defining instrumentation

Now we define the notion of *code instrumentation* in fast-path transactions.

An execution E of a HyTM $\mathcal{M}$ *appears t-sequential* to a transaction $T_k \in txns(E)$ if there exists an execution E' of $\mathcal{M}$ such that:

- $txns(E') \subseteq txns(E) \setminus \{T_k\}$ and the configuration after E' is t-quiescent,
- every transaction $T_m \in txns(E)$ that precedes T_k in real-time order is included in E' such that $E|m = E'|m$,
- for every transaction $T_m \in txns(E')$, $Rset_{E'}(T_m) \subseteq Rset_E(T_m)$ and $Wset_{E'}(T_m) \subseteq Wset_E(T_m)$, and
- $E' \cdot E|k$ is an execution of $\mathcal{M}$.

Intuitively, as the name indicates, execution E appears t-sequential to a transaction T_k participating in E if T_k cannot distinguish E from a t-complete execution E' which includes all the t-complete transactions preceding T_k in E and performing the same steps as in E.

Definition 2 **(Data and metadata base objects)** Let $\mathcal{X}$ be the set of t-objects operated by a HyTM implementation $\mathcal{M}$. Now we partition the set of base objects used by $\mathcal{M}$ into a set $\mathbb{D}$ of *data* objects and a set $\mathbb{M}$ of *metadata* objects ($\mathbb{D} \cap \mathbb{M} = \emptyset$). We further partition $\mathbb{D}$ into sets $\mathbb{D}_X$ associated with each t-object $X \in \mathcal{X}$: $\mathbb{D} = \bigcup_{X \in \mathcal{X}} \mathbb{D}_X$, for all $X \neq Y$ in $\mathcal{X}$, $\mathbb{D}_X \cap \mathbb{D}_Y = \emptyset$, such that:

1. In every execution E, each fast-path transaction $T_k \in txns(E)$ only accesses base objects in $\bigcup_{X \in DSet(T_k)} \mathbb{D}_X$ or $\mathbb{M}$.
2. Let $E \cdot \rho$ and $E \cdot E' \cdot \rho'$ be two t-complete executions, such that E and $E \cdot E'$ are t-complete, ρ and ρ' are complete executions of a transaction $T_k \notin txns(E \cdot E')$, $H_\rho = H_{\rho'}$, and $\forall T_m \in txns(E')$, $Dset(T_m) \cap Dset(T_k) = \emptyset$. Then the states of the base objects $\bigcup_{X \in DSet(T_k)} \mathbb{D}_X$ in the configuration after $E \cdot \rho$ and $E \cdot E' \cdot \rho'$ are the same.
3. Let execution E appear t-sequential to a transaction T_k and let the enabled event e of T_k after E be a primitive on a base object $b \in \mathbb{D}$. Then, unless e returns $\perp$, $E \cdot e$ also appears t-sequential to T_k.

Intuitively, the first condition says that a transaction is only allowed to access data objects based on its data set. The second condition says that transactions with disjoint data sets can communicate only via metadata objects. Finally, the last condition means that base objects in $\mathbb{D}$ may only contain the "values" of t-objects, and cannot be used to detect concurrent transactions. Note that our results lower bound the number of metadata objects that must be accessed by some fast-path transaction in a given execution, thus from a cost perspective, $\mathbb{D}$ should be made as large as possible.

All HyTM proposals we are aware of, such as *Hybrid-NOrec* [13,40], *PhTM* [35] and others [11,15,32,42], conform to our definition of instrumentation in fast-path transactions. For instance, HybridNOrec [13,40] employs a distinct base object in $\mathbb{D}$ for each t-object and a global *sequence lock* as the metadata that is accessed by fast-path transactions to detect concurrency with slow-path transactions. Similarly, the HyTM implementation by *Damron et al.* [15] also associates a distinct base object in $\mathbb{D}$ for each t-object and additionally, a *transaction header* and *ownership record* as metadata base objects. In fact, our framework even characterizes conventional STMs such as *DSTM* [28] and *NOrec* [14] which maintain similar separation between data and matadata base objects.

Definition 3 (Uninstrumented HyTMs) A HyTM implementation $\mathcal{M}$ provides *uninstrumented writes (resp. reads)* if in every execution E of $\mathcal{M}$, for every write-only (resp. read-only) fast-path transaction T_k, all primitives in $E|k$ are performed on base objects in $\mathbb{D}$. A HyTM is uninstrumented if both its reads and writes are uninstrumented.

Observation 3 *Consider any execution E of a HyTM implementation $\mathcal{M}$ which provides uninstrumented reads (resp. writes). For any fast-path read-only (resp. write-only) transaction $T_k \notin txns(E)$, that runs step-contention free after E, the execution E appears t-sequential to T_k.*

4.1 Impossibility of uninstrumented HyTMs

We now show that any strictly serializable HyTM must be instrumented, even under a very weak progress assumption by which a transaction is guaranteed to commit only when run t-sequentially (also known as *minimal progress* [25]):

Definition 4 (Sequential TM-progress for HyTMs) A HyTM implementation $\mathcal{M}$ provides *sequential TM-progress for fast-path transactions (and resp. slow-path)* if in every execution E of $\mathcal{M}$, a fast-path (and resp. slow-path) transaction T_k returns A_k in E only if T_k incurs a capacity abort or T_k is concurrent to another transaction. We say that $\mathcal{M}$ provides sequential TM-progress if it provides sequential TM-progress for fast-path and slow-path transactions.

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

Proof Suppose by contradiction that such a HyTM $\mathcal{M}$ exists. For simplicity, assume that v is the initial value of t-objects X, Y and Z and transactions are executed by distinct processes. Let E be the t-complete step contention-free execution of a slow-path transaction T_0 that performs $read_0(Z) \rightarrow v$, $write_0(X, nv)$, $write_0(Y, nv)$ ($nv \neq v$), and commits. Such an execution exists since $\mathcal{M}$ ensures sequential TM-progress.

By Observation 3, any transaction that runs step contention-free starting from a prefix of E must return a non-abort value. Since any such transaction reading X or Y must return v when it starts from the empty prefix of E and nv when it starts from E.

Thus, there exists E', the longest prefix of E that cannot be extended with the t-complete step contention-free execution of a *fast-path* transaction reading X or Y and returning nv. Let e be the enabled event of T_0 in the configuration after E'. Without loss of generality, suppose that there exists an execution $E' \cdot e \cdot E_y$ where E_y is the t-complete step contention-free execution fragment of some fast-path transaction T_y that reads Y is returns nv (Fig. 3a).

Claim 5 *$\mathcal{M}$ has an execution $E' \cdot E_z \cdot E_x$, where*

- *E_z is the t-complete step contention-free execution fragment of a fast-path transaction T_z that writes $nv \neq v$ to Z and commits*
- *E_x is the t-complete step contention-free execution fragment of a fast-path transaction T_x that performs a single t-read $read_x(X) \rightarrow v$ and commits.*

Proof By Observation 3, the extension of E' in which T_z writes to Z and tries to commit appears t-sequential to T_z. By sequential TM-progress, T_z completes the write and commits. Let $E' \cdot E_z$ (Fig. 3b) be the resulting execution of $\mathcal{M}$.

Similarly, the extension of E' in which T_x reads X and tries to commit appears t-sequential to T_x. By sequential TM-progress, T_x commits and let $E' \cdot E_x$ be the resulting execution of $\mathcal{M}$. By the definition of E', $read_x(X)$ must return v in $E' \cdot E_x$.

Since $\mathcal{M}$ is uninstrumented and the data sets of T_x and T_z are disjoint, the sets of base objects accessed in the execution fragments E_x and E_y are also disjoint. Thus, $E' \cdot E_z \cdot E_x$ is indistinguishable to T_x from the execution $E' \cdot E_x$, which implies that $E' \cdot E_z \cdot E_x$ is an execution of $\mathcal{M}$ (Fig. 3c). □

Finally, we prove that the sequence of events, $E' \cdot E_z \cdot E_x \cdot e \cdot E_y$ is an execution of $\mathcal{M}$.

Since the transactions T_x, T_y, T_z have pairwise disjoint data sets in $E' \cdot E_z \cdot E_x \cdot e \cdot E_y$, no base object accessed in E_y can be accessed in E_x and E_z. The read operation on X performed by T_y in $E' \cdot e \cdot E_y$ returns nv and, by the definition of E' and e, T_y must have accessed the base object b modified in the event e by T_0. Thus, b is not accessed in E_x and E_z and $E' \cdot E_z \cdot E_x \cdot e$ is an execution of $\mathcal{M}$. Summing up, $E' \cdot E_z \cdot E_x \cdot e \cdot E_y$ is indistinguishable to T_y from $E' \cdot e \cdot E_y$,

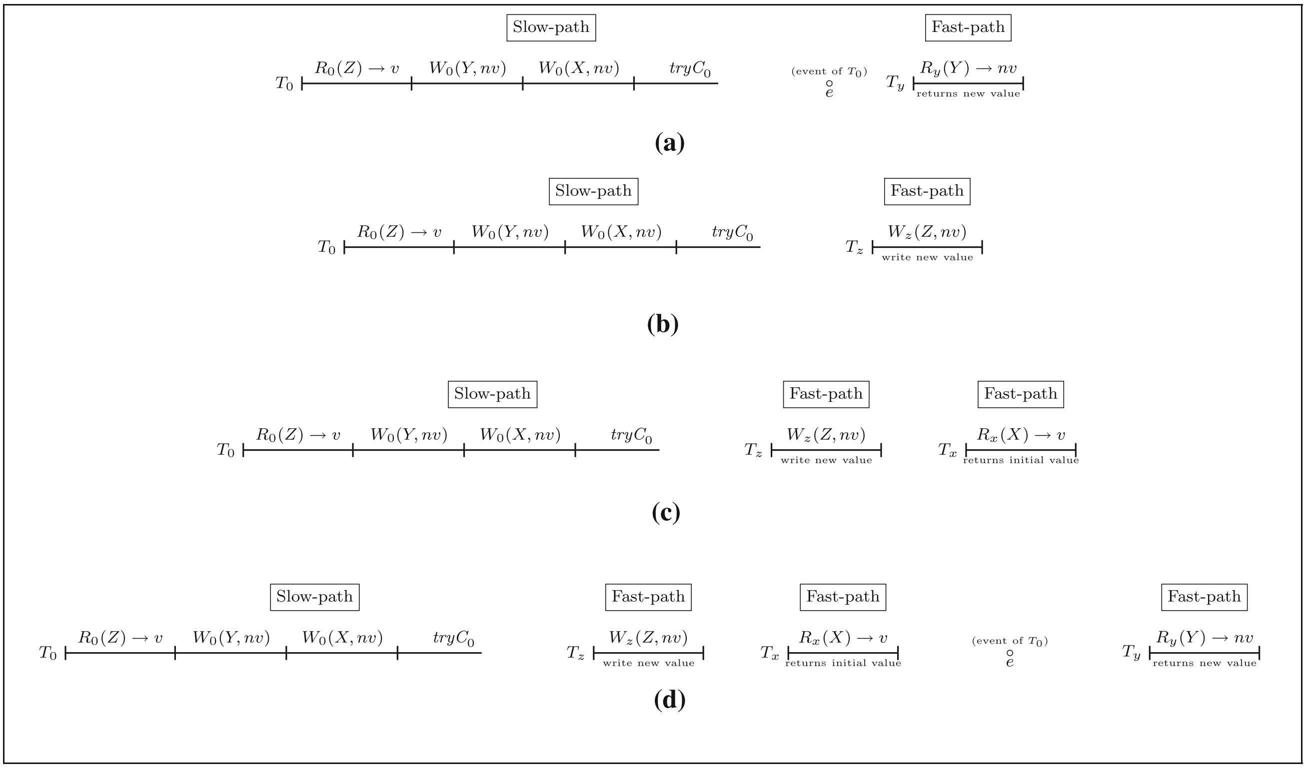

Fig. 3 Executions in the proof of Theorem 4; execution in 3d is not strictly serializable **a** T_y must return the new value **b** since T_z is uninstrumented, by Observation 3 and sequential TM-progress, T_z must commit **c** since T_x does not access any metadata, it cannot abort and must return the initial value value of X **d** T_y does not contend with T_x or T_z on any base object

which implies that $E' \cdot E_z \cdot E_x \cdot e \cdot E_y$ is an execution of $\mathcal{M}$ (Fig. 3d).

But the resulting execution is not strictly serializable. Indeed, suppose that a serialization exists. As the value written by T_0 is returned by a committed transaction T_y, T_0 must be committed and precede T_y in the serialization. Since T_x returns the initial value of X, T_x must precede T_0. Since T_0 reads the initial value of Z, T_0 must precede T_z. Finally, T_z must precede T_x to respect the real-time order. The cycle in the serialization establishes a contradiction. □

5 Linear instrumentation lower bound

In this section, we show that giving HyTM the ability to run and commit transactions in parallel brings considerable instrumentation costs. We focus on a natural progress condition called progressiveness [22–24] that allows a transaction to abort only if it experiences a read-write or write-write conflict with a concurrent transaction:

Definition 5 (Progressiveness for HyTMs) We say that transactions T_i and T_j *conflict* in an execution E on a t-object X if $X \in Dset(T_i) \cap Dset(T_j)$ and $X \in Wset(T_i) \cup Wset(T_j)$.

A HyTM implementation $\mathcal{M}$ is *fast-path* (resp. *slow-path*) *progressive* if in every execution E of $\mathcal{M}$ and for every fast-path (and resp. slow-path) transaction T_i that aborts in E, either A_i is a capacity abort or T_i conflicts with some transaction T_j that is concurrent to T_i in E. We say $\mathcal{M}$ is *progressive* if it is both fast-path and slow-path progressive.

We show that for every opaque fast-path progressive HyTM that provides obstruction-free TM-liveness, an arbitrarily long read-only transaction might access a number of distinct metadata base objects that is linear in the size of its read set or experience a capacity abort.

The following auxiliary results will be crucial in proving our lower bound. We observe first that a fast path transaction in a progressive HyTM can contend on a base object only with a conflicting transaction.

Lemma 1 *Let $\mathcal{M}$ be any fast-path progressive HyTM implementation. Let $E \cdot E_1 \cdot E_2$ be an execution of $\mathcal{M}$ where E_1 (and resp. E_2) is the step contention-free execution fragment of transaction $T_1 \notin txns(E)$ (and resp. $T_2 \notin txns(E)$), T_1 (and resp. T_2) does not conflict with any transaction in $E \cdot E_1 \cdot E_2$, and at least one of T_1 or T_2 is a fast-path transaction. Then, T_1 and T_2 do not contend on any base object in $E \cdot E_1 \cdot E_2$.*

Proof Suppose, by contradiction that T_1 or T_2 contend on the same base object in $E \cdot E_1 \cdot E_2$.

If in E_1, T_1 performs a nontrivial event on a base object on which they contend, let e_1 be the last event in E_1 in which T_1 performs such an event to some base object b and e_2, the first event in E_2 that accesses b. Otherwise, T_1 only performs trivial events in E_1 to base objects on which it contends with T_2 in $E \cdot E_1 \cdot E_2$: let e_2 be the first event in E_2 in which E_2 performs a nontrivial event to some base object b on which they contend and e_1, the last event of E_1 in T_1 that accesses b.

Let E_1' (and resp. E_2') be the longest prefix of E_1 (and resp. E_2) that does not include e_1 (and resp. e_2). Since before accessing b, the execution is step contention-free for T_1, $E \cdot E_1' \cdot E_2'$ is an execution of $\mathcal{M}$. By construction, T_1 and T_2 do not conflict in $E \cdot E_1' \cdot E_2'$. Moreover, $E \cdot E_1 \cdot E_2'$ is indistinguishable to T_2 from $E \cdot E_1' \cdot E_2'$. Hence, T_1 and T_2 are poised to apply contending events e_1 and e_2 on b in the execution $\tilde{E} = E \cdot E_1' \cdot E_2'$. Recall that at least one event of e_1 and e_2 must be nontrivial.

Consider the execution $\tilde{E} \cdot e_1 \cdot e_2'$ where e_2' is the event of p_2 in which it applies the primitive of e_2 to the configuration after $\tilde{E} \cdot e_1$. After $\tilde{E} \cdot e_1$, b is contained in the tracking set of process p_1. If b is contained in τ_1 in the shared mode, then e_2' is a nontrivial primitive on b, which invalidates τ_1 in $\tilde{E} \cdot e_1 \cdot e_2'$. If b is contained in τ_1 in the exclusive mode, then any subsequent access of b invalidates τ_1 in $\tilde{E} \cdot e_1 \cdot e_2'$. In both cases, τ_1 is invalidated and T_1 incurs a tracking set abort. Thus, transaction T_1 must return A_1 in any extension of $E \cdot e_1 \cdot e_2$—a contradiction to the assumption that $\mathcal{M}$ is progressive. □

Iterative application of Lemma 1 implies the following:

Corollary 1 *Let $\mathcal{M}$ be any fast-path progressive HyTM implementation. Let $E \cdot E_1 \cdots E_i \cdot E_{i+1} \cdots E_m$ be any execution of $\mathcal{M}$ where for all $i \in \{1, \ldots, m\}$, E_i is the step contention-free execution fragment of transaction $T_i \notin txns(E)$ and any two transactions in $E_1 \cdots E_m$ do not conflict. For all $i, j = 1, \ldots, m$, $i \neq j$, if T_i is fast-path, then T_i and T_j do not contend on a base object in $E \cdot E_1 \cdots E_m$*

Proof Let T_i be a fast-path transaction. By Lemma 1, in $E \cdot E_1 \cdots E_i \cdots E_m$, T_i does not contend with T_{i-1} (if $i > 1$) or T_{i+1} (if $i < m$) on any base object and, thus, E_i commutes with E_{i-1} and E_{i+1}. Thus, $E \cdot E_1 \cdots E_{i-2} \cdot E_i \cdot E_{i-1} \cdot E_{i+1} \cdots E_m$ (if $i > 1$) and $E \cdot E_1 \cdots E_{i-1} \cdot E_{i+1} \cdot E_i \cdot E_{i+2} \cdots E_m$ (if $i < m$) are executions of $\mathcal{M}$. By iteratively applying Lemma 1, we derive that T_i does not contend with any T_j, $j \neq i$.

We say that execution fragments E and E' are *similar* if they export equivalent histories, *i.e.*, no process can see the difference between them by looking at the invocations and responses of t-operations. We now use Corollary 1 to show that t-operations only accessing data base objects cannot detect contention with non-conflicting transactions.

Lemma 2 *Let E be any t-complete execution of a progressive HyTM implementation $\mathcal{M}$ that provides OF TM-liveness. For any $m \in \mathbb{N}$, consider a set of m executions of $\mathcal{M}$ of the form $E \cdot E_i \cdot \gamma_i \cdot \rho_i$ where E_i is the t-complete step contention-free execution fragment of a transaction T_{m+i}, γ_i is a complete step contention-free execution fragment of a* fast-path *transaction T_i such that $Dset(T_i) \cap Dset(T_{m+i}) = \emptyset$ in $E \cdot E_i \cdot \gamma_i$, and ρ_i is the execution fragment of a t-operation by T_i that does not contain accesses to any metadata base object. If, for all $i, j \in \{1, \ldots, m\}$, $i \neq j$, $Dset(T_i) \cap Dset(T_{m+j}) = \emptyset$, $Dset(T_i) \cap Dset(T_j) = \emptyset$ and $Dset(T_{m+i}) \cap Dset(T_{m+j}) = \emptyset$, then there exists a t-complete step contention-free execution fragment E' that is similar to $E_1 \cdots E_m$ such that for all $i \in \{1, \ldots, m\}$, $E \cdot E' \cdot \gamma_i \cdot \rho_i$ is an execution of $\mathcal{M}$.*

Proof Observe that any two transactions in the execution fragment $E_1 \cdots E_m$ access mutually disjoint data sets. Since $\mathcal{M}$ is progressive and provides OF TM-liveness, there exists a t-sequential execution fragment $E' = E_1' \cdots E_m'$ such that, for all $i \in \{1, \ldots, m\}$, the execution fragments E_i and E_i' are similar and $E \cdot E'$ is an execution of $\mathcal{M}$. Corollary 1 implies that, for all $i \in \{1, \ldots, m\}$, $\mathcal{M}$ has an execution of the form $E \cdot E_1' \cdots E_i' \cdots E_m' \cdot \gamma_i$. More specifically, $\mathcal{M}$ has an execution of the form $E \cdot \gamma_i \cdot E_1' \cdots E_i' \cdots E_m'$. Recall that the execution fragment ρ_i of fast-path transaction T_i that extends γ_i contains accesses only to base objects in $\bigcup_{X \in DSet(T_i)} \mathbb{D}_X$. Moreover, for all $i, j \in \{1, \ldots, m\}$; $i \neq j$, $Dset(T_i) \cap Dset(T_{m+j}) = \emptyset$ and $Dset(T_{m+i}) \cap Dset(T_{m+j}) = \emptyset$.

It follows that $\mathcal{M}$ has an execution of the form $E \cdot \gamma_i \cdot E_1' \cdots E_i' \cdot \rho_i \cdot E_{i+1}' \cdots E_m'$. and the states of each of the base objects $\bigcup_{X \in DSet(T_i)} \mathbb{D}_X$ accessed by T_i in the configuration after $E \cdot \gamma_i \cdot E_1' \cdots E_i'$ and $E \cdot \gamma_i \cdot E_i$ are the same. But $E \cdot \gamma_i \cdot E_i \cdot \rho_i$ is an execution of $\mathcal{M}$. Thus, for all $i \in \{1, \ldots, m\}$, $\mathcal{M}$ has an execution of the form $E \cdot E' \cdot \gamma_i \cdot \rho_i$. □

Finally, we are now ready to derive our lower bound.

Theorem 6 *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.*

Here is a high-level overview of the proof technique. Let κ be the smallest integer such that some fast-path transaction running step contention-free after a t-quiescent configuration performs κ t-reads and incurs a capacity abort.

We prove that, for all $m \leq \kappa - 1$, there exists a t-complete execution E_m and a set S_m with $|S_m| = 2^{\kappa - m}$ of read-only fast-path transactions that access mutually disjoint data sets such that each transaction in S_m that runs step contention-free from E_m and performs t-reads of m distinct t-objects accesses at least one distinct metadata base object within the execution of each t-read operation.

We proceed by induction. Assume that the induction statement holds for all $m < \kappa - 1$. We prove that a set S_{m+1}; $|S_{m+1}| = 2^{\kappa-(m+1)}$ of fast-path transactions, each of which run step contention-free after the same t-complete execution E_{m+1}, perform $m+1$ t-reads of distinct t-objects so that at least one distinct metadata base object is accessed within the execution of each t-read operation. In our construction, we pick any two new transactions from the set S_m and show that one of them running step contention-free from a t-complete execution that extends E_m performs $m+1$ t-reads of distinct t-objects so that at least one distinct metadata base object is accessed within the execution of each t-read operation. In this way, the set of transactions is reduced by half in each step of the induction until one transaction remains which must have accessed a distinct metadata base object in every one of its $m+1$ t-reads.

Intuitively, since all the transactions that we use in our construction access mutually disjoint data sets, we can apply Lemma 1 to construct a t-complete execution E_{m+1} such that each of the fast-path transactions in S_{m+1} when running step contention-free after E_{m+1} perform $m+1$ t-reads so that at least one distinct metadata base object is accessed within the execution of each t-read operation.

We now present the formal proof:

Proof In the constructions which follow, every fast-path transaction executes at most $m+1$ t-reads. Let κ be the smallest integer such that some fast-path transaction running step contention-free after a t-quiescent configuration performs κ t-reads and incurs a capacity abort. We proceed by induction.

Induction statement. We prove that, for all $m \le \kappa - 1$, there exists a t-complete execution E_m and a set S_m with $|S_m| = 2^{\kappa-m}$ of read-only fast-path transactions that access mutually disjoint data sets such that each transaction $T_{f_i} \in S_m$ that runs step contention-free from E_m and performs t-reads of m distinct t-objects accesses at least one distinct metadata base object within the execution of each t-read operation. Let E_{f_i} be the step contention-free execution of T_{f_i} after E_m and let $Dset(T_{f_i}) = \{X_{i,1}, \ldots, X_{i,m}\}$.

The induction. Assume that the induction statement holds for all $m \le \kappa - 1$. The statement is trivially true for the base case $m = 0$ for every $\kappa \in \mathbb{N}$.

We will prove that a set S_{m+1}; $|S_{m+1}| = 2^{\kappa-(m+1)}$ of fast-path transactions, each of which run step contention-free from the same t-quiescent configuration E_{m+1}, perform $m+1$ t-reads of distinct t-objects so that at least one distinct metadata base object is accessed within the execution of each t-read operation.

The construction proceeds in *phases*: there are exactly $\frac{|S_m|}{2}$ phases. In each phase, we pick any two new transactions from the set S_m and show that one of them running step contention-free after a t-complete execution that extends E_m performs $m+1$ t-reads of distinct t-objects so that at least one distinct metadata base object is accessed within the execution of each t-read operation.

Throughout this proof, we will assume that any two transactions (and resp. execution fragments) with distinct subscripts represent distinct identifiers.

For all $i \in \{0, \ldots, \frac{|S_m|}{2} - 1\}$, let $X_{2i+1}, X_{2i+2} \notin \bigcup_{i=0}^{|S_m|-1} \{X_{i,1}, \ldots, X_{i,m}\}$ be distinct t-objects and let v be the value of X_{2i+1} and X_{2i+2} after E_m. Let T_{s_i} denote a slow-path transaction which writes $nv \neq v$ to X_{2i+1} and X_{2i+2}. Let E_{s_i} be the t-complete step contention-free execution fragment of T_{s_i} running immediately after E_m.

Let E'_{s_i} be the longest prefix of the execution E_{s_i} such that $E_m \cdot E'_{s_i}$ can be extended neither with the complete step contention-free execution fragment of transaction $T_{f_{2i+1}}$ that performs its m t-reads of $X_{2i+1,1}, \ldots, X_{2i+1,m}$ and then performs $read_{f_{2i+1}}(X_{2i+1})$ and returns nv, nor with the complete step contention-free execution fragment of some transaction $T_{f_{2i+2}}$ that performs t-reads of $X_{2i+2_1}, \ldots, X_{2i+2,m}$ and then performs $read_{f_{2i+2}}(X_{2i+2})$ and returns nv. Progressiveness and OF TM-liveness of $\mathcal{M}$ stipulates that such an execution exists.

Let e_i be the enabled event of T_{s_i} in the configuration after $E_m \cdot E'_{s_i}$. By construction, the execution $E_m \cdot E'_{s_i}$ can be extended with at least one of the complete step contention-free executions of transaction $T_{f_{2i+1}}$ performing $(m+1)$ t-reads of $X_{2i+1,1}, \ldots, X_{2i+1,m}, X_{2i+1}$ such that $read_{f_{2i+1}}(X_{2i+1}) \rightarrow nv$ or transaction $T_{f_{2i+2}}$ performing t-reads of $X_{2i+2,1}, \ldots, X_{2i+2,m}, X_{2i+2}$ such that $read_{f_{2i+2}}(X_{2i+2}) \rightarrow nv$. Without loss of generality, suppose that $T_{f_{2i+1}}$ reads the value of X_{2i+1} to be nv after $E_m \cdot E'_{0_i} \cdot e_i$.

For any $i \in \{0, \ldots, \frac{|S_m|}{2} - 1\}$, we will denote by α_i the execution fragment which we will construct in phase i. For any $i \in \{0, \ldots, \frac{|S_m|}{2} - 1\}$, we prove that $\mathcal{M}$ has an execution of the form $E_m \cdot \alpha_i$ in which $T_{f_{2i+1}}$ (or $T_{f_{2i+2}}$) running step contention-free after a t-complete execution that extends E_m performs $m+1$ t-reads of distinct t-objects so that at least one distinct metadata base object is accessed within the execution of each first m t-read operations and $T_{f_{2i+1}}$ (or $T_{f_{2i+2}}$) is poised to apply an event after $E_m \cdot \alpha_i$ that accesses a distinct metadata base object during the $(m+1)^{th}$ t-read. Furthermore, we will show that $E_m \cdot \alpha_i$ appears t-sequential to $T_{f_{2i+1}}$ (or $T_{f_{2i+2}}$).

(Construction of phase i)

Let $E_{f_{2i+1}}$ (and resp. $E_{f_{2i+2}}$) be the complete step contention-free execution of the t-reads of $X_{2i+1,1}, \ldots, X_{2i+1,m}$ (and resp. $X_{2i+2,1}, \ldots, X_{2i+2,m}$) running after E_m

by $T_{f_{2i+1}}$ (and resp. $T_{f_{2i+2}}$). By the inductive hypothesis, transaction $T_{f_{2i+1}}$ (and resp. $T_{f_{2i+2}}$) accesses m distinct metadata objects in the execution $E_m \cdot E_{f_{2i+1}}$ (and resp. $E_m \cdot E_{f_{2i+2}}$). Recall that transaction $T_{f_{2i+1}}$ does not conflict with transaction T_{s_i}. Thus, by Corollary 1, $\mathcal{M}$ has an execution of the form $E_m \cdot E'_{s_i} \cdot e_i \cdot E_{f_{2i+1}}$ (and resp. $E_m \cdot E'_{s_i} \cdot e_i \cdot E_{f_{2i+2}}$).

Let $E_{rf_{2i+1}}$ be the complete step contention-free execution fragment of $read_{f_{2i+1}}(X_{2i+1})$ that extends $E_{2i+1} = E_m \cdot E'_{s_i} \cdot e_i \cdot E_{f_{2i+1}}$. By OF TM-liveness, $read_{f_{2i+1}}(X_{2i+1})$ must return a matching response in $E_{2i+1} \cdot E_{rf_{2i+1}}$. We now consider two cases.

Case I: *Suppose $E_{rf_{2i+1}}$ accesses at least one metadata base object b not previously accessed by $T_{f_{2i+1}}$.*

Let $E'_{rf_{2i+1}}$ be the longest prefix of $E_{rf_{2i+1}}$ which does not apply any primitives to any metadata base object b not previously accessed by $T_{f_{2i+1}}$. The execution $E_m \cdot E'_{s_i} \cdot e_i \cdot E_{f_{2i+1}} \cdot E'_{rf_{2i+1}}$ appears t-sequential to $T_{f_{2i+1}}$ because $E_{f_{2i+1}}$ does not contend with T_{s_i} on any base object and any common base object accessed in the execution fragments $E'_{rx_{2i+1}}$ and E_{s_i} by $T_{f_{2i+1}}$ and T_{s_i} respectively must be data objects contained in $\mathbb{D}$. Thus, we have that $|Dset(T_{f_{2i+1}})| = m+1$ and that $T_{f_{2i+1}}$ accesses m distinct metadata base objects within each of its first m t-read operations and is poised to access a distinct metadata base object during the execution of the $(m+1)^{th}$ t-read. In this case, let $\alpha_i = E_m \cdot E'_{s_i} \cdot e_i \cdot E_{f_{2i+1}} \cdot E'_{rf_{2i+1}}$.

Case II: *Suppose $E_{rf_{2i+1}}$ does not access any metadata base object not previously accessed by $T_{f_{2i+1}}$.*

In this case, we will first prove the following:

Claim 7 *$\mathcal{M}$ has an execution of the form $E_{2i+2} = E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}}$ where $\bar{E}_{f_{2i+1}}$ is the t-complete step contention-free execution of $T_{f_{2i+1}}$ in which $read_{f_{2i+1}}(X_{2i+1}) \rightarrow nv$, $T_{f_{2i+1}}$ invokes $tryC_{f_{2i+1}}$ and returns a matching response.*

Proof Since $E_{rf_{2i+1}}$ does not contain accesses to any distinct metadata base objects, the execution $E_m \cdot E'_{s_i} \cdot e_i \cdot E_{f_{2i+1}} \cdot E_{rf_{2i+1}}$ appears t-sequential to $T_{f_{2i+1}}$. By definition of the event e_i, $read_{f_{2i+1}}(X_{2i+1})$ must access the base object to which the event e_i applies a nontrivial primitive and return the response nv in $E'_{s_i} \cdot e_i \cdot E_{f_{2i+1}} \cdot E_{rf_{2i+1}}$. By OF TM-liveness, it follows that $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}}$ is an execution of $\mathcal{M}$.

Now recall that $E_m \cdot E'_{s_i} \cdot e_i \cdot E_{f_{2i+2}}$ is an execution of $\mathcal{M}$ because transactions $T_{f_{2i+2}}$ and T_{s_i} do not conflict in this execution and thus, cannot contend on any base object. Finally, because $T_{f_{2i+1}}$ and $T_{f_{2i+2}}$ access disjoint data sets in $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}}$, by Lemma 1 again, we have that $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}}$ is an execution of $\mathcal{M}$.

Let $E_{rf_{2i+2}}$ be the complete step contention-free execution fragment of $read_{f_{2i+2}}(X_{2i+2})$ after $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}}$. By the induction hypothesis and Claim 7, transaction $T_{f_{2i+2}}$ must access m distinct metadata base objects in the execution $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}}$.

If $E_{rf_{2i+2}}$ accesses some metadata base object, then by the argument given in Case I applied to transaction $T_{f_{2i+2}}$, we get that $T_{f_{2i+2}}$ accesses m distinct metadata base objects within each of the first m t-read operations and is poised to access a distinct metadata base object during the execution of the $(m+1)^{th}$ t-read.

Thus, suppose that $E_{rf_{2i+2}}$ does not access any metadata base object previously accessed by $T_{f_{2i+2}}$. We claim that this is impossible and proceed to derive a contradiction. In particular, $E_{rf_{2i+2}}$ does not contend with T_{s_i} on any metadata base object. Consequently, the execution $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}}$ appears t-sequential to $T_{x_{2i+2}}$ since $E_{rx_{2i+2}}$ only contends with T_{s_i} on base objects in $\mathbb{D}$. It follows that $E_{2i+2} \cdot E_{rf_{2i+2}}$ must also appear t-sequential to $T_{f_{2i+2}}$ and so $E_{rf_{2i+2}}$ cannot abort. Recall that the base object, say b, to which T_{s_i} applies a nontrivial primitive in the event e_i is accessed by $T_{f_{2i+1}}$ in $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}}$; thus, $b \in \mathbb{D}_{X_{2i+1}}$. Since $X_{2i+1} \notin Dset(T_{f_{2i+2}})$, b cannot be accessed by $T_{f_{2i+2}}$. Thus, the execution $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}} \cdot E_{rf_{2i+2}}$ is indistinguishable to $T_{f_{2i+2}}$ from the execution $\hat{E}_i \cdot E'_{s_i} \cdot E_{f_{2i+2}} \cdot E_{rf_{2i+2}}$ in which $read_{f_{2i+2}}(X_{2i+2})$ must return the response v (by construction of E'_{s_i}).

But we observe now that the execution $E_m \cdot E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}} \cdot E_{rf_{2i+2}}$ is not opaque. In any serialization corresponding to this execution, T_{s_i} must be committed and must precede $T_{f_{2i+1}}$ because $T_{f_{2i+1}}$ read nv from X_{2i+1}. Also, transaction $T_{f_{2i+2}}$ must precede T_{s_i} because $T_{f_{2i+2}}$ read v from X_{2i+2}. However $T_{f_{2i+1}}$ must precede $T_{f_{2i+2}}$ to respect real-time ordering of transactions. Clearly, there exists no such serialization—contradiction.

Letting $E'_{rf_{2i+2}}$ be the longest prefix of $E_{rf_{2i+2}}$ which does not access a base object $b \in \mathbb{M}$ not previously accessed by $T_{f_{2i+2}}$, we can let $\alpha_i = E'_{s_i} \cdot e_i \cdot \bar{E}_{f_{2i+1}} \cdot E_{f_{2i+2}} \cdot E'_{rf_{2i+2}}$ in this case.

Combining Cases I and II, the following claim holds.

Claim 8 *For each $i \in \{0, \ldots, \frac{|S_m|}{2} - 1\}$, $\mathcal{M}$ has an execution of the form $E_m \cdot \alpha_i$ in which*

(1) some fast-path transaction $T_i \in txns(\alpha_i)$ performs t-reads of $m+1$ distinct t-objects so that at least one distinct metadata base object is accessed within the execution of each of the first m t-reads, T_i is poised to access a distinct metadata base object after $E_m \cdot \alpha_i$ during the execution of the $(m+1)^{th}$ t-read and the execution appears t-sequential to T_i,

(2) the two fast-path transactions in the execution fragment α_i do not contend on the same base object.

(Collecting the phases)

We will now describe how we can construct the set S_{m+1} of fast-path transactions from these $\frac{|S_m|}{2}$ phases and force each of them to access $m+1$ distinct metadata base objects when running step contention-free after the same t-complete execution.

For each $i \in \{0, \ldots, \frac{|S_m|}{2} - 1\}$, let β_i be the subsequence of the execution α_i consisting of all the events of the fast-path transaction that is poised to access a $(m+1)^{th}$ distinct metadata base object. Henceforth, we denote by T_i the fast-path transaction that participates in β_i. Then, from Claim 8, it follows that, for each $i \in \{0, \ldots, \frac{|S_m|}{2} - 1\}$, $\mathcal{M}$ has an execution of the form $E_m \cdot E'_{s_i} \cdot e_i \cdot \beta_i$ in which the fast-path transaction T_i performs t-reads of $m+1$ distinct t-objects so that at least one distinct metadata base object is accessed within the execution of each of the first m t-reads, T_i is poised to access a distinct metadata base object after $E_m \cdot E'_{s_i} \cdot e_i \cdot \beta_i$ during the execution of the $(m+1)^{th}$ t-read and the execution appears t-sequential to T_i.

The following result is a corollary to the above claim that is obtained by applying the definition of "appears t-sequential". Recall that $E'_{s_i} \cdot e_i$ is the t-incomplete execution of slow-path transaction T_{s_i} that accesses t-objects X_{2i+1} and X_{2i+2}.

Corollary 2 *For all $i \in \{0, \ldots, \frac{|(S_m|}{2} - 1\}$, $\mathcal{M}$ has an execution of the form $E_m \cdot E_i \cdot \beta_i$ such that the configuration after $E_m \cdot E_i$ is t-quiescent, $txns(E_i) \subseteq \{T_{s_i}\}$ and $Dset(T_{s_i}) \subseteq \{X_{2i+1}, X_{2i+2}\}$ in E_i.*

We can represent the execution $\beta_i = \gamma_i \cdot \rho_i$ where fast-path transaction T_i performs complete t-reads of m distinct t-objects in γ_i and then performs an incomplete t-read of the $(m+1)^{th}$ t-object in ρ_i in which T_i only accesses base objects in $\bigcup_{X \in DSet(T_i)} \{X\}$. Recall that T_i and T_{s_i} do not contend on the same base object in the execution $E_m \cdot E_i \cdot \gamma_i$. Thus, for all $i \in \{0, \ldots, \frac{|S_m|}{2} - 1\}$, $\mathcal{M}$ has an execution of the form $E_m \cdot \gamma_i \cdot E_i \cdot \rho_i$.

Observe that the fast-path transaction $T_i \in \gamma_i$ does not access any t-object that is accessed by any slow-path transaction in the execution fragment $E_0 \cdots E_{\frac{|S_m|}{2}-1}$. By Lemma 2, there exists a t-complete step contention-free execution fragment E' that is similar to $E_0 \cdots E_{\frac{|S_m|}{2}-1}$ such that for all $i \in \{0, \ldots, \frac{|S_m|}{2} - 1\}$, $\mathcal{M}$ has an execution of the form $E_m \cdot E' \cdot \gamma_i \cdot \rho_i$. By our construction, the enabled event of each fast-path transaction $T_i \in \beta_i$ in this execution is an access to a distinct metadata base object.

Let S_{m+1} denote the set of all fast-path transactions that participate in the execution fragment $\beta_0 \cdots \beta_{\frac{|(S_m|}{2}-1}$ and $E_{m+1} = E_m \cdot E'$. Thus, $|S_{m+1}|$ fast-path transactions, each of which run step contention-free from the same t-quiescent configuration, perform $m+1$ t-reads of distinct t-objects so that at least one distinct metadata base object is accessed within the execution of each t-read operation. This completes the proof. □

6 Instrumentation-optimal HyTM algorithms

In this section, we describe two "instrumentation-optimal" progressive HyTMs. We show that these implementations are provably opaque in our HyTM model.

6.1 A linear upper bound on instrumentation

We prove that the lower bound in Theorem 6 is tight by describing an 'instrumentation-optimal' HyTM implementation (Algorithm 1) that is opaque, progressive, provides wait-free TM-liveness, uses *invisible reads*.

Algorithm 1 Progressive opaque HyTM implementation that provides uninstrumented writes and invisible reads; code for process p_i executing transaction T_k

```
Shared objects:
  v_j ∈ 𝔻, for each t-object X_j
      allows reads, writes and cas
  r_j ∈ 𝕄, for each t-object X_j
      allows reads, writes and cas

Local objects:
  Lset(T_k) ⊆ Wset(T_k), initially empty
  Oset(T_k) ⊆ Wset(T_k), initially empty

    Code for slow-path transactions

read_k(X_j):   // slow-path
  if X_j ∉ Rset_k then
    [ov_j, k_j] := read(v_j)
    Rset(T_k) := Rset(T_k) ∪ {X_j, [ov_j, k_j]}
    if r_j ≠ 0 then
      Return A_k
    if ∃X_j ∈ Rset(T_k):(ov_j, k_j) ≠ read(v_j) then
      Return A_k
    Return ov_j
  else
    ov_j := Rset(T_k).locate(X_j)
    Return ov_j

write_k(X_j, v):   // slow-path
  (ov_j, k_j) := read(v_j)
  nv_j := v
  Wset(T_k) := Wset(T_k) ∪ {X_j, [ov_j, k_j]}
  Return ok

tryC_k():   // slow-path
  if Wset(T_k) = ∅ then
    Return C_k
  locked := acquire(Wset(T_k))
  if ¬ locked then
```

```
    Return A_k
  if isAbortable() then
    release(Lset(T_k))
    Return A_k
  for all X_j ∈ Wset(T_k) do
    if v_j.cas([ov_j, k_j], [nv_j, k]) then
      Oset(T_k) := Oset(T_k) ∪ {X_j}
    else
      undo(Oset(T_k))
  release(Wset(T_k))
  Return C_k

Function: isAbortable() :
  if ∃X_j ∈ Rset(T_k): X_j ∉ Wset(T_k) ∧ read(r_j) ≠ 0 then
    Return true
  if ∃X_j ∈ Rset(T_k):[ov_j, k_j] ≠ read(v_j) then
    Return true
  Return false
Function: acquire(Q):
  for all X_j ∈ Q do
    if r_j.cas(0, 1) then
      Lset(T_k) := Lset(T_k) ∪ {X_j}
    else
      release(Lset(T_k))
      Return false
  Return true

Function: release(Q):
  for all X_j ∈ Q do
    r_j.write(0)
  Return ok

Function: undo(Oset(T_k)):
  for all X_j ∈ Oset(T_k) do
    v_j.cas([nv_j, k], [ov_j, k_j])
  release(Wset(T_k))
  Return A_k
```

Code for fast-path transactions

```
read_k(X_j):   // fast-path
  [ov_j, k_j] := read(v_j) // cached read
  if read(r_j) ≠ 0 then
    Return A_k
  Return ov_j

write_k(X_j, v):   // fast-path
  write(v_j, [nv_j, k]) // cached write
  Return ok

tryC_k():   // fast-path
  commit − cache_i // returns C_k or A_k
```

6.1.1 Base objects

For every t-object X_j, our implementation maintains a base object $v_j \in \mathbb{D}$ that stores the value of X_j and a metadata base object r_j, which is a *lock bit* that stores 0 or 1.

6.1.2 Fast-path transactions

For a fast-path transaction T_k, the $read_k(X_j)$ implementation first reads r_j to check if X_j is locked by a concurrent updating transaction. If so, it returns A_k, else it returns the value of X_j. Updating fast-path transactions use uninstrumented writes: $write(X_j, v)$ simply stores the cached state of X_j along with its value v and if the cache has not been invalidated, updates the shared memory during $tryC_k$ by invoking the $commit - cache$ primitive.

6.1.3 Slow-path read-only transactions

Any $read_k(X_j)$ invoked by a slow-path transaction first reads the value of the object from v_j, checks if r_j is set and then performs *value-based validation* on its entire read set to check if any of them have been modified. If either of these conditions is true, the transaction returns A_k. Otherwise, it returns the value of X_j. A read-only transaction simply returns C_k during the tryCommit.

6.1.4 Slow-path updating transactions

The $write_k(X, v)$ implementation of a slow-path transaction stores v and the current value of X_j locally, deferring the actual update in shared memory to tryCommit.

During $tryC_k$, an updating slow-path transaction T_k attempts to obtain exclusive write access to its entire write set as follows: for every t-object $X_j \in Wset(T_k)$, it writes 1 to each base object r_j by performing a *compare-and-set* (*cas*) primitive that checks if the value of r_j is not 1 and, if so, replaces it with 1. If the *cas* fails, then T_k releases the locks on all objects X_ℓ it had previously acquired by writing 0 to r_ℓ and then returns A_k. Intuitively, if the *cas* fails, some concurrent transaction is performing a t-write to a t-object in $Wset(T_k)$. If all the locks on the write set were acquired successfully, T_k checks if any t-object in $Rset(T_k)$ is concurrently being updated by another transaction and then performs value-based validation of the read set. If a conflict is detected from the these checks, the transaction is aborted. Finally, $tryC_k$ attempts to write the values of the t-objects via *cas* operations. If any *cas* on the individual base objects fails, there must be a concurrent fast-path writer, and so T_k rolls back the state of the base objects that were updated, releases locks on its write set and returns A_k. The roll backs are performed with *cas* operations, skipping any which fail to allow for concurrent fast-path writes to locked locations. Note that if a concurrent read operation of a fast-path transaction T_ℓ finds an "invalid" value in v_j that was written by such transaction T_k but has not been rolled back yet, then T_ℓ either incurs a tracking set abort later because T_k has updated v_j or finds r_j to be 1. In both cases, the read operation of T_ℓ aborts.

The implementation uses invisible reads (no nontrivial primitives are applied by reading transactions). Every t-operation returns a matching response within a finite number of its steps.

6.1.5 Complexity

Every t-read operation performed by a fast-path transaction accesses a metadata base object once (the lock bit corresponding to the t-object), which is the price to pay for detecting conflicting updating slow-path transactions. Write operations of fast-path transactions are uninstrumented.

Lemma 3 *Algorithm 1 implements an opaque TM.*

Proof Let E by any execution of Algorithm 1. Since opacity is a safety property, it is sufficient to prove that every finite execution is opaque [7]. 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 ℓ_{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 or adding responses to the remaining pending invocations as follows:

- incomplete $read_k$, $write_k$ operation performed by a slow-path transaction T_k is removed from E; an incomplete $tryC_k$ is removed from E if T_k has not performed any write to a base object r_j; $X_j \in Wset(T_k)$ in Line 36, otherwise it is completed by including C_k after E.
- every incomplete $read_k$, $tryA_k$, $write_k$ and $tryC_k$ performed by a fast-path transaction T_k is removed from E.

Linearization points. Now a linearization H of E is obtained by associating linearization points to t-operations in the obtained completion of E. For all t-operations performed a slow-path transaction T_k, linearization points as assigned as follows:

- For every t-read op_k that returns a non-A_k value, ℓ_{op_k} is chosen as the event in Line 11 of Algorithm 1, else, ℓ_{op_k} is chosen as invocation event of op_k
- For every $op_k = write_k$ that returns, ℓ_{op_k} is chosen as the invocation event of op_k
- For every $op_k = tryC_k$ that returns C_k such that $Wset(T_k) \neq \emptyset$, ℓ_{op_k} is associated with the first write to a base object performed by release when invoked in Line 40, else if op_k returns A_k, ℓ_{op_k} is associated with the invocation event of op_k
- For every $op_k = tryC_k$ that returns C_k such that $Wset(T_k) = \emptyset$, ℓ_{op_k} is associated with Line 28

For all t-operations performed a fast-path transaction T_k, linearization points as assigned as follows:

- For every t-read op_k that returns a non-A_k value, ℓ_{op_k} is chosen as the event in Line 66 of Algorithm 1, else, ℓ_{op_k} is chosen as invocation event of op_k
- For every op_k that is a $tryC_k$, ℓ_{op_k} is the $commit - cache_k$ primitive invoked by T_k
- For every op_k that is a $write_k$, ℓ_{op_k} is the event in Line 71.

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

Serialization points. The serialization of a transaction T_j, denoted as δ_{T_j} is associated with the linearization point of a t-operation performed by the transaction.

We obtain a t-complete history $\bar{H}$ from H as follows. A serialization S is obtained by associating serialization points to transactions in $\bar{H}$ as follows: for every transaction T_k in H that is complete, but not t-complete, we insert $tryC_k \cdot A_k$ immediately after the last event of T_k in H.

- If T_k is an updating transaction that commits, then δ_{T_k} is ℓ_{tryC_k}
- If T_k is a read-only or aborted transaction, then δ_{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 9 *If $T_i \prec_H T_j$, then $T_i <_S T_j$*

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 $\delta_{T_i} <_E \delta_{T_j}$ implies $T_i <_S T_j$. □

Claim 10 *S is legal.*

Proof We claim that for every $read_j(X_m) \rightarrow v$, there exists some slow-path transaction T_i (or resp. fast-path) that performs $write_i(X_m, v)$ and completes the event in Line 36 (or resp. Line 71) such that $read_j(X_m) \not\prec_H^{RT} write_i(X_m, v)$.

Suppose that T_i is a slow-path transaction: since $read_j(X_m)$ returns the response v, the event in Line 11 succeeds the event in Line 36 performed by $tryC_i$. Since $read_j(X_m)$ can return a non-abort response only after T_i writes 0 to r_m in Line 58, T_i must be committed in S. Consequently, $\ell_{tryC_i} <_E \ell_{read_j(X_m)}$. Since, for any updating committing transaction T_i, $\delta_{T_i} = \ell_{tryC_i}$, it follows that $\delta_{T_i} <_E \delta_{T_j}$.

Otherwise if T_i is a fast-path transaction, then clearly T_i is a committed transaction in S. Recall that $read_j(X_m)$ can read v during the event in Line 11 only after T_i applies the $commit - cache$ primitive. By the assignment of linearization points, $\ell_{tryC_i} <_E \ell_{read_j(X_m)}$ and thus, $\delta_{T_i} <_E \ell_{read_j(X_m)}$.

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 $write_k(X_m, v')$; $v' \neq v$ such that $T_i <_S T_k <_S T_j$.

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

$$(T_i <_S T_k) \Longleftrightarrow (\delta_{T_i} <_E \delta_{T_k})$$
$$(\delta_{T_i} <_E \delta_{T_k}) \Longleftrightarrow (\ell_{tryC_i} <_E \ell_{tryC_k})$$

Since, T_j reads the value of X written by T_i, one of the following is true: $\ell_{tryC_i} <_E \ell_{tryC_k} <_E \ell_{read_j(X_m)}$ or $\ell_{tryC_i} <_E \ell_{read_j(X_m)} <_E \ell_{tryC_k}$.

Suppose that $\ell_{tryC_i} <_E \ell_{tryC_k} <_E \ell_{read_j(X_m)}$.

(*Case I:*) T_i and T_k are slow-path transactions.

Thus, T_k returns a response from the event in Line 29 before the read of the base object associated with X_m by T_j in Line 11. Since T_i and T_k are both committed in E, T_k returns *true* from the event in Line 29 only after T_i writes 0 to r_m in Line 58.

If T_j is a slow-path transaction, recall that $read_j(X_m)$ checks if X_j is locked by a concurrent transaction, then performs read-validation (Line 13) before returning a matching response. We claim that $read_j(X_m)$ must return A_j in any such execution.

Consider the following possible sequence of events: T_k returns *true* from *acquire* function invocation, updates the value of X_m to shared-memory (Line 36), T_j reads the base object v_m associated with X_m, T_k releases X_m by writing 0 to r_m and finally T_j performs the check in Line 13. But in this case, $read_j(X_m)$ is forced to return the value v' written by T_m— contradiction to the assumption that $read_j(X_m)$ returns v.

Otherwise suppose that T_k acquires exclusive access to X_m by writing 1 to r_m and returns *true* from the invocation of *acquire*, updates v_m in Line 36), T_j reads v_m, T_j performs the check in Line 13 and finally T_k releases X_m by writing 0 to r_m. Again, $read_j(X_m)$ must return A_j since T_j reads that r_m is 1—contradiction.

A similar argument applies to the case that T_j is a fast-path transaction. Indeed, since every *data* base object read by T_j is contained in its tracking set, if any concurrent transaction updates any t-object in its read set, T_j is aborted immediately by our model(cf. Sect. 3).

Thus, $\ell_{tryC_i} <_E \ell_{read_j(X)} <_E \ell_{tryC_k}$.

(*Case II:*) T_i is a slow-path transaction and T_k is a fast-path transaction. Thus, T_k returns C_k before the read of the base object associated with X_m by T_j in Line 11, but after the response of *acquire* by T_i in Line 29. Since $read_j(X_m)$ reads the value of X_m to be v and not v', T_i performs the *cas* to v_m in Line 36 after the T_k performs the $commit - cache$ primitive (since if otherwise, T_k would be aborted in E). But then the *cas* on v_m performed by T_i would return *false* and T_i would return A_i—contradiction.

(*Case III:*) T_k is a slow-path transaction and T_i is a fast-path transaction. This is analogous to the above case.

(*Case IV:*) T_i and T_k are fast-path transactions. Thus, T_k returns C_k before the read of the base object associated with X_m by T_j in Line 11, but before T_i returns C_i (this follows from Observations 1 and 2). Consequently, $read_j(X_m)$ must read the value of X_m to be v' and return v'—contradiction.

We now need to prove that δ_{T_j} indeed precedes ℓ_{tryC_k} in E.

Consider the two possible cases:

– Suppose that T_j is a read-only transaction. Then, δ_{T_j} is assigned to the last t-read performed by T_j that returns a non-A_j value. If $read_j(X_m)$ is not the last t-read that returned a non-A_j value, then there exists a $read_j(X')$ such that $\ell_{read_j(X_m)} <_E \ell_{tryC_k} <_E \ell_{read_j(X')}$. But then this t-read of X' must abort by performing the checks in Line 13 or incur a tracking set abort—contradiction.
– Suppose that T_j is an updating transaction that commits, then $\delta_{T_j} = \ell_{tryC_j}$ which implies that $\ell_{read_j(X)} <_E \ell_{tryC_k} <_E \ell_{tryC_j}$. Then, T_j must neccesarily perform the checks in Line 32 and return A_j or incur a tracking set abort—contradiction to the assumption that T_j is a committed transaction.

The proof follows. □

The conjunction of Claims 9 and 10 establish that Algorithm 1 is opaque.

Theorem 11 *There exists an opaque HyTM implementation $\mathcal{M}$ that provides uninstrumented writes, invisible reads, progressiveness and wait-free TM-liveness such that in every execution E of $\mathcal{M}$, every read-only fast-path transaction $T \in txns(E)$ accesses $O(|Rset(T)|)$ distinct metadata base objects.*

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

Consider the cases under which a slow-path transaction T_k may be aborted in any execution.

– Suppose that there exists a $read_k(X_j)$ performed by T_k that returns A_k from Line 13. Thus, there exists a transaction that has written 1 to r_j in Line 50, but has not yet written 0 to r_j in Line 58 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 $Rset(T_k)$, thus forcing a read-write conflict.
– Suppose that $tryC_k$ performed by T_k that returns A_k from Line 30. Thus, there exists a transaction that has written 1 to r_j in Line 50, but has not yet written 0 to r_j in Line 58. Thus, T_k encounters write-write conflict 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 32. Since T_k returns A_k from Line 32 for the same reason it returns A_k after Line 13, the proof follows.

Consider the cases under which a fast-path transaction T_k may be aborted in any execution E.

– Suppose that a $read_k(X_m)$ performed by T_k returns A_k from Line 67. Thus, there exists a concurrent slow-path transaction that is pending in its tryCommit and has written 1 to r_m, but not released the lock on X_m i.e. T_k conflicts with another transaction in E.
– Suppose that T_k returns A_k while performing a cached access of some base object b via a trivial (and resp. non-trivial) primitive. Indeed, this is possible only if some concurrent transaction writes (and resp. reads or writes) to b. However, two transactions T_k and T_m may contend on b in E only if there exists $X \in Dset(T_i) \cap Dset(T_j)$ and $X \in Wset(T_i) \cup Wset(T_j)$. from Line 30. The same argument applies for the case when T_k returns A_k while performing $commit - cache_k$ in E.

(Complexity) The implementation uses uninstrumented writes since each $write_k(X_m)$ simply writes to $v_m \in \mathbb{D}_{X_m}$ and does not access any metadata base object. The complexity of each $read_k(X_m)$ is a single access to a metadata base object r_m in Line 67 that is not accessed any other transaction T_i unless $X_m \in Dset(T_i)$. while the $tryC_k$ just calls $cache - commit_k$ that returns C_k. Thus, each read-only transaction T_k accesses $O(|Rset(T_k)|)$ distinct metadata base objects in any execution. □

6.2 Providing partial concurrency at low cost

We showed that allowing fast-path transactions to run concurrently in HyTM results in an instrumentation cost that is proportional to the read-set size of a fast-path transaction. But can we run at least *some* transactions concurrently with constant instrumentation cost, while still keeping invisible reads?

Algorithm 2 Opaque HyTM implementation with progressive slow-path and sequential fast-path TM-progress; code for T_k by process p_i

```
Shared objects:
  v_j ∈ 𝔻, for each t-object X_j
      allows reads, writes and cas
  r_j ∈ 𝕄, for each t-object X_j
      allows reads, writes and cas
  Count, fetch-and-add object

    Code for slow-path transactions

tryC_k():   // slow-path
  if Wset(T_k) = ∅ then
     Return C_k
  locked := acquire(Wset(T_k))
  if ¬ locked then
     Return A_k
  Count.add(1)
  if isAbortable() then
     release(Lset(T_k))
     Return A_k
  for all X_j ∈ Wset(T_k) do
     if v_j.cas((ov_j, k_j), (nv_j, k)) then
        Oset(T_k) := Oset(T_k) ∪ {X_j}
     else
        Return undo(Oset(T_k))
  release(Wset(T_k))
  Count.add(−1)
  Return C_k
      Code for fast-path transactions

read_k(X_j):   // fast-path
  if Rset(T_k) = ∅ then
     l ← read(Count)  // cached read
     if l ≠ 0 then
        Return A_k
  (ov_j, k_j) := read(v_j)  // cached read
  Return ov_j

write_k(X_j, v):   // fast-path
  v_j.write(nv_j, k)  // cached write
  Return ok

tryC_k():   // fast-path
  commit − cache_i  // returns C_k or A_k
```

Algorithm 2 implements a *slow-path progressive* opaque HyTM with invisible reads and wait-free TM-liveness. To fast-path transactions, it only provides *sequential* TM-progress (they are only guaranteed to commit in the absence of concurrency), but in return the algorithm is only using a single metadata base object fa that is read once by a fast-path transaction and accessed twice with a *fetch-and-add* primitive by an updating slow-path transaction. Thus, the instrumentation cost of the algorithm is constant.

Intuitively, fa allows fast-path transactions to detect the existence of concurrent updating slow-path transactions. Each time an updating slow-path updating transaction tries to commit, it increments fa and once all writes to data base objects are completed (this part of the algorithm is identical to Algorithm 1) or the transaction is aborted, it decrements fa. Therefore, $fa \neq 0$ means that at least one slow-path updating transaction is incomplete. A fast-path transaction simply checks if $fa \neq 0$ in the beginning and aborts if so, otherwise, its code is identical to that in Algorithm 1. Note that this way, any update of fa automatically causes a tracking set abort of any incomplete fast-path transaction.

Theorem 12 *There exists an opaque HyTM implementation that provides uninstrumented writes, invisible reads, progressiveness for slow-path transactions, sequential TM-progress for fast-path transactions and wait-free TM-liveness such that in every its execution E, every fast-path transaction accesses at most one metadata base object.*

Proof The proof of opacity is almost identical to the analogous proof for Algorithm 1 in Lemma 3.

As with Algorithm 1, enumerating the cases under which a slow-path transaction T_k returns A_k proves that Algorithm 2 satisfies progressiveness for slow-path transactions. Any fast-path transaction T_k; $Rset(T_k) \neq \emptyset$ reads the metadata base object *Count* and adds it to the process's tracking set (Line 27). If the value of *Count* is not 0, indicating that there exists a concurrent slow-path transaction pending in its tryCommit, T_k returns A_k. Thus, the implementation provides sequential TM-progress for fast-path transactions.

Also, in every execution E of $\mathcal{M}$, no fast-path write-only transaction accesses any metadata base object and a fast-path reading transaction accesses the metadata base object *Count* exactly once, during the first t-read. □

7 Related work

The term *instrumentation* was originally used in the context of HyTMs [13,35,40] to indicate the overhead a hardware transaction induces in order to detect pending software transactions. The impossibility of designing HyTMs without any code instrumentation was informally suggested in [13]. We prove this formally in this paper.

In [8], Attiya and Hillel considered the instrumentation cost of *privatization*, *i.e.*, allowing transactions to isolate data items by making them private to a process so that no other process is allowed to modify the privatized item. The model we consider is fundamentally different, in that we model hardware transactions at the level of cache coherence, and do not consider non-transactional accesses. (In particular, neither data nor meta-data objects are private in our model.) The proof techniques we employ are also different.

Uninstrumented HTMs may be viewed as being *disjoint-access parallel (DAP)* [9,30,34]. As such, some of the techniques used in the proof of Theorem 4 extend those used in [9,21,25]. However, proving lower bounds on the instrumentation costs of the HyTM fast-path is challenging, since such t-operations can automatically abort due to any contending concurrent step.

Circa 2005, several papers introduced HyTM implementations [6,15,32] that integrated HTMs with variants of *DSTM* [28]. These implementations provide nontrivial concurrency between hardware and software transactions (progressiveness), by imposing instrumentation on hardware transactions: every t-read operation incurs at least one extra access to a metadata base object. Our Theorem 6 shows that this overhead is unavoidable. Of note, write operations of these HyTMs are also instrumented, but our Algorithm 1 shows that it is not necessary.

Implementations like *PhTM* [35] and *HybridNOrec* [13] overcome the per-access instrumentation cost of [15,32] by realizing that if one is prepared to sacrifice progress, hardware transactions need instrumentation only at the boundaries of transactions to detect pending software transactions, à la Transactional Lock Elision (TLE) [38]. Inspired by this observation, our HyTM implementation described in Algorithm 2 overcomes the linear per-read instrumentation cost by allowing hardware readers to abort due to a concurrent software writer, but maintains progressiveness for software transactions, unlike [13,35,36]. Recent experimental results on today Intel and IBM POWER8 HTMs show that instrumentation is indeed a huge cost to concurrency in opaque HyTMs [10,11,20], thus demonstrating that the lower bound costs established in this paper also exist in practice.

References [26,40] provide detailed overviews on HyTM designs and implementations. The software component of the HyTM algorithms presented in this paper is inspired by progressive STM implementations [14,17,33] and is subject to the lower bounds for progressive STMs established in [8, 23,25,33].

8 Concluding remarks

We have introduced an analytical model for HyTM that captures the notion of cached accesses as performed by hardware transactions. We then derived lower and upper bounds in this model that capture the inherent tradeoff between the degree of concurrency between hardware and software transactions, and the metadata-access overhead introduced on the hardware.

To precisely characterize the costs incurred by hardware transactions, we made a distinction between the set of memory locations which store the data values of the t-objects, and the locations that store the metadata information. To the best of our knowledge, all known HyTM proposals, such as *HybridNOrec* [13,40], *PhTM* [35] and others [15,32] avoid co-locating the data and metadata within a single base object.

Recent work has investigated alternatives to the STM fallback, such as *sandboxing* [2,12], or *hardware-accelerated* STM [43,44], and the use of both direct *and* cached accesses within the same hardware transaction to reduce instrumentation overhead [20,32,40,41]. Specifically, [20] showed how to build efficient HyTMs for IBM POWER8 architectures which allow the use of direct accesses within hardware transactions to reduce instrumentation overhead. Another recent approach proposed *reduced hardware transactions* [36], 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. Amalgamated lock elision (ALE) was proposed in [3] which improves over TLE by executing the slow-path as a series of segments, each of which is a dynamic length hardware transaction. We plan to extend our model to incorporate such schemes in future work.

Our HyTM model is a natural extension of previous frameworks developed for Software Transactional Memory, and has the advantage of being relatively simple. We hope that our model and techniques will enable more research on the limitations and power of HyTM systems, and that our results will prove useful for practitioners.

Acknowledgements The study was funded by Agence Nationale de la Recherche (Grant No. ANR-14-CE35-0010-01, project DISCMAT), National Science Foundation (Grant Nos. CCF-1217921, CCF-1301926), U.S. Department of Energy (Grant No. IIS-1447786).

References

1. Advanced Synchronization Facility Proposed Architectural Specification (2009). http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf
2. Afek, Y., Levy, A., Morrison, A.: Software-improved hardware lock elision. In: PODC. ACM (2014)
3. Afek, Y., Matveev, A., Moll, O.R., Shavit, N.: Amalgamated lock-elision. In: Distributed Computing—29th International Symposium, DISC 2015, Tokyo, Japan, October 7–9, 2015, Proceedings, pp. 309–324 (2015). doi:10.1007/978-3-662-48653-5_21
4. Alistarh, D., Eugster, P., Herlihy, M., Matveev, A., Shavit, N.: Stacktrack: An automated transactional approach to concurrent memory reclamation. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14, pp. 25:1–25:14. ACM, New York, NY, USA (2014). doi:10.1145/2592798.2592808
5. Alistarh, D., Kopinsky, J., Kuznetsov, P., Ravi, S., Shavit, N.: Inherent limitations of hybrid transactional memory. In: Distributed Computing—29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings, pp. 185–199 (2015). doi:10.1007/978-3-662-48653-5_13
6. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, HPCA '05, pp. 316–327. IEEE Computer Society, Washington, DC, USA (2005). doi:10.1109/HPCA.2005.41
7. Attiya, H., Hans, S., Kuznetsov, P., Ravi, S.: Safety of deferred update in transactional memory. In: IEEE 33rd International Conference on Distributed Computing Systems **0**, pp. 601–610 (2013). doi:10.1109/ICDCS.2013.57
8. Attiya, H., Hillel, E.: The cost of privatization in software transactional memory. IEEE Trans. Computers **62**(12), 2531–2543 (2013). http://dblp.uni-trier.de/db/journals/tc/tc62.html#AttiyaH13
9. Attiya, H., Hillel, E., Milani, A.: Inherent limitations on disjoint-access parallel implementations of transactional memory. Theory Comput. Syst. **49**(4), 698–719 (2011). doi:10.1007/s00224-010-9304-5
10. Brown, T., Ravi, S.: Cost of concurrency in hybrid transactional memory. In: Workshop on Transactional Computing (Transact), 2017 (2017)
11. Calciu, I., Gottschlich, J., Shpeisman, T., Pokam, G., Herlihy, M.: Invyswell: a hybrid transactional memory for haswell's restricted transactional memory. In: International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014, pp. 187–200 (2014). doi:10.1145/2628071.2628086
12. Calciu, I., Shpeisman, T., Pokam, G., Herlihy, M.: Improved single global lock fallback for best-effort hardware transactional memory. In: Transact 2014 Workshop. ACM (2014)
13. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M.L., Spear, M.F.: Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In: R. Gupta, T.C. Mowry (eds.) ASPLOS, pp. 39–52. ACM (2011). http://dblp.uni-trier.de/db/conf/asplos/asplos2011.html#DalessandroCWLMSS11
14. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining stm by abolishing ownership records. SIGPLAN Not. **45**(5), 67–78 (2010). doi:10.1145/1837853.1693464
15. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. SIGPLAN Not. **41**(11), 336–346 (2006). doi:10.1145/1168918.1168900
16. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early experience with a commercial hardware transactional memory implementation. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, pp. 157–168. ACM, New York, NY, USA (2009). doi:10.1145/1508244.1508263
17. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: Proceedings of the 20th International Conference on Distributed Computing, DISC'06, pp. 194–208. Springer, Berlin, Heidelberg (2006). doi:10.1007/11864219_14
18. Dragojević, A., Herlihy, M., Lev, Y., Moir, M.: On the power of hardware transactional memory to simplify memory management. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC '11, pp. 99–108. ACM, New York, NY, USA (2011). doi:10.1145/1993806.1993821
19. Ellen, F., Hendler, D., Shavit, N.: On the inherent sequentiality of concurrent objects. SIAM J. Comput. **41**(3), 519–536 (2012)
20. Felber, P., Issa, S., Matveev, A., Romano, P.: Hardware read-write lock elision. In: Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, pp. 34:1–34:15. ACM, New York, NY, USA (2016). doi:10.1145/2901318.2901346
21. Guerraoui, R., Kapalka, M.: On obstruction-free transactions. In: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, SPAA '08, pp. 304–313. ACM, New York, NY, USA (2008). doi:10.1145/1378533.1378587
22. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium

on Principles and Practice of Parallel Programming, PPoPP '08, pp. 175–184. ACM, New York, NY, USA (2008). doi:10.1145/1345206.1345233
23. Guerraoui, R., Kapalka, M.: The semantics of progress in lock-based transactional memory. SIGPLAN Not. **44**(1), 404–415 (2009). doi:10.1145/1594834.1480931
24. Guerraoui, R., Kapalka, M.: Transactional memory: Glimmer of a theory. In: Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09, pp. 1–15. Springer-Verlag, Berlin, Heidelberg (2009). doi:10.1007/978-3-642-02658-4_1
25. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool, San Rafael (2010)
26. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory. Synthesis Lectures on Computer Architecture, 2nd edn. Morgan & Claypool Publishers, San Rafael (2010)
27. Herlihy, M.: Wait-free synchronization. ACM Trans. Progr. Lang. Syst. **13**(1), 123–149 (1991)
28. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing, PODC '03, pp. 92–101. ACM, New York, NY, USA (2003). doi:10.1145/872035.872048
29. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA, pp. 289–300 (1993)
30. Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In: PODC, pp. 151–160 (1994)
31. Karnagel, T., Dementiev, R., Rajwar, R., Lai, K., Legler, T., Schlegel, B., Lehner, W.: Improving in-memory database index performance with intel ® transactional synchronization extensions. In: 20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014, pp. 476–487 (2014). doi:10.1109/HPCA.2014.6835957
32. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid transactional memory. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, pp. 209–220. ACM, New York, NY, USA (2006). doi:10.1145/1122971.1123003
33. Kuznetsov, P., Ravi, S.: On the cost of concurrency in transactional memory. In: OPODIS, pp. 112–127 (2011). Full version: http://arxiv.org/abs/1103.1302
34. Kuznetsov, P., Ravi, S.: Grasping the gap between blocking and non-blocking transactional memories. J. Parallel Distrib. Comput. **101**, 1–16 (2017). doi:10.1016/j.jpdc.2016.10.008
35. Lev, Y., Moir, M., Nussbaum, D.: Phtm: Phased transactional memory. In: In Workshop on Transactional Computing (Transact), 2007. research.sun.com/scalable/pubs/ TRANSACT2007PhTM.pdf
36. Matveev, A., Shavit, N.: Reduced hardware transactions: a new approach to hybrid transactional memory. In: Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures, pp. 11–22. ACM (2013)
37. Ohmacht, M.: Memory Speculation of the Blue Gene/Q Compute Chip (2011). http://wands.cse.lehigh.edu/IBM_BQC_PACT2011.ppt
38. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 34, pp. 294–305. IEEE Computer Society, Washington, DC, USA (2001). http://dl.acm.org/citation.cfm?id=563998.564036
39. Reinders, J.: Transactional Synchronization in Haswell (2012). http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/
40. Riegel, T.: Software Transactional Memory Building Blocks (2013). Chapter 7. Thesis
41. Riegel, T., Marlier, P., Nowack, M., Felber, P., Fetzer, C.: Optimizing hybrid transactional memory: The importance of nonspeculative operations. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, pp. 53–64. ACM (2011)
42. Ruan, W., Spear, M.F.: Hybrid transactional memory revisited. In: Distributed Computing—29th International Symposium, DISC 2015, Tokyo, Japan, October 7–9, 2015, Proceedings, pp. 215–231 (2015). doi:10.1007/978-3-662-48653-5_15
43. Saha, B., Adl-Tabatabai, A.R., Jacobson, Q.: Architectural support for software transactional memory. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pp. 185–196. IEEE Computer Society, Washington, DC, USA (2006). doi:10.1109/MICRO.2006.9
44. Spear, M.F., Shriraman, A., Dalessandro, L., Dwarkadas, S., Scott, M.L.: Nonblocking transactions without indirection using alert-on-update. In: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '07, pp. 210–220. ACM, New York, NY, USA (2007). doi:10.1145/1248377.1248414
45. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pp. 19:1–19:11. ACM, New York, NY, USA (2013). doi:10.1145/2503210.2503232