Inherent Limitations of Hybrid Transactional Memory

Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, Nir Shavit

Inherent Limitations of Hybrid Transactional Memory

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

[1] Microsoft Research, Cambridge
[2] Massachusetts Institute of Technology
[3] Télécom ParisTech
[4] TU Berlin
[5] Tel Aviv University

Abstract. Several Hybrid Transactional Memory (HyTM) schemes have recently been proposed to complement the fast, but best-effort nature of Hardware Transactional Memory (HTM) 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.

1 Introduction

Hybrid transactional memory. Since its introduction by Herlihy and Moss [17], *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, 21, 22] have been eagerly anticipated and scrutinized by the community.

* The author is supported by the Agence Nationale de la Recherche, ANR-14-CE35-0010-01, project DISCMAT
** Support is gratefully acknowledged from the National Science Foundation under grants CCF-1217921, CCF-1301926, and IIS-1447786, the Department of Energy under grant ER26116/DE-SC0008923, and the Oracle and Intel corporations

Early experience with programming HTM, e.g. [3,11,12], 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 [9, 10, 18, 19] 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 [10, 18] 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* [14] 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.* [24] 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?

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 [20] suggests that the overhead im-

posed 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 [9, 10, 18, 19, 23].

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.* [7, 13, 14], 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* [14]. To the best of our knowledge, this is the first formal proof of correctness of a HyTM algorithm.

Low-instrumentation HyTM. Our main lower bound result shows that there are high inherent instrumentation costs to progressive HyTM designs [10, 18]. Interestingly, some recent HyTM schemes [9, 19, 20, 24] 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.

Roadmap. 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 Section 6 describes our instrumentation-optimal opaque HyTM implementations. Section 7 presents the related work and Section 8 concludes the paper. The tech report contains the formal proofs of the lower bounds, algorithm pseudo-code and their correctness proofs [4].

2 Preliminaries

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.

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 $Wset(T_k) \neq \emptyset$.

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 $T_k \in \mathcal{T}$, $E|k = E'|k$.

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.

Contention. We assume that base objects are accessed with *read-modify-write* (rmw) primitives. 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.

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_E or is immediately followed by a matching response. An execution E is *t-sequential* if there are no concurrent transactions in E.

Informally, a t-sequential history S is *legal* if every t-read of a t-object returns the *latest written value* of this t-object in S. A history H is *opaque* if there exists a legal t-sequential history S equivalent to H such that S respects the real-time order of transactions in H [14].

3 Hybrid Transactional Memory (HyTM)

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.

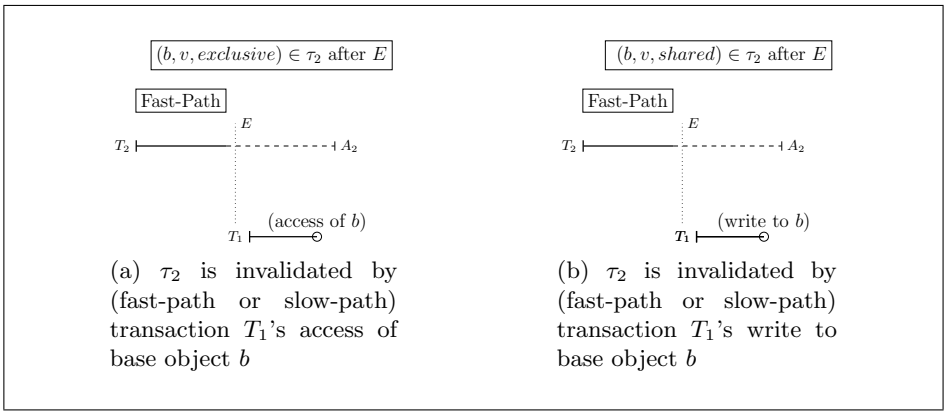
More precisely, τ_i is a set of triples (b, v, m) where b is a base object identifier, v is a value, and $m \in \{shared, exclusive\}$ is an access *mode*. The triple (b, v, m) is added to the tracking set when i performs a cached rmw access of b, where m is set to *exclusive* if the access is nontrivial, 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 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 [22]. The first cause can be modelled probabilistically in the above framework, which would not however significantly affect our claims and proofs, except for a more cumbersome presentation. Also, our lower bounds are based exclusively on executions containing t-reads and t-writes. Therefore, in the following, we only consider tracking set and capacity aborts.



(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. 1: Tracking set aborts in fast-path transactions

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 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 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 $\mathcal{M}$. Moreover:

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

Observation 2. If a fast-path transaction $T_m \in \mathit{txns}(E) \setminus \{T_k\}$ 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.

4 HyTM 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: (i) $txns(E') \subseteq txns(E) \setminus \{T_k\}$ and the configuration after E' is t-quiescent, (ii) every transaction $T_m \in txns(E)$ that precedes T_k in real-time order is included in E' such that $E|m = E'|m$, (iii) 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 (iv) $E' \cdot E|k$ is an execution of $\mathcal{M}$.

Definition 1 (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 $\bot$, $E \cdot e$ 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 will lower bound the number of metadata objects that must be accessed under particular assumptions, thus from a cost perspective, $\mathbb{D}$ should be made as large as possible.

All HyTM proposals we aware of, such as *HybridNOrec* [9, 23], *PhTM* [19] and others [10, 18], conform to our definition of instrumentation in fast-path transactions. For instance, HybridNOrec [9,23] 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.* [10] 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.

Definition 2 (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.

Impossibility of uninstrumented HyTMs. We can now show that any strictly serializable HyTM must be instrumented, even under a very weak liveness and progress assumptions of *sequential TM-liveness* and *sequential TM-progress*. *sequential* TM-liveness Sequential TM-liveness guarantees that t-operations running in the absence of concurrent transactions return in a finite number of its steps. Sequential TM-progress stipulates that a transaction can only be aborted only if it is concurrent with another transaction. Note that sequential TM-liveness and TM-progress allow a transaction not running t-sequentially to abort or block indefinitely.

Theorem 1. *There does not exist a strictly serializable uninstrumented HyTM implementation that ensures sequential TM-progress and TM-liveness.*

Due to space constraints, we defer the proof the technical report [4], and provide an outline below. Suppose by contradiction that such a HyTM exists and let E be a t-sequential execution of it in which a slow-path transaction T_0 reads t-object Z (returning the initial value), then writes a new value nv to t-objects X and Y, and commits. Since the HyTM is uninstrumented, Observation 3 implies that a fast-path transaction running step contention-free cannot detect the presence of a concurrent transaction and, by sequential TM-liveness and TM-progress, the transaction must eventually commit. Thus, there exists E', the longest prefix of E that cannot be extended with the t-complete step-contention-free execution neither of a fast-path transaction T_x reading X and returning nv nor of a fast-path transaction T_y reading Y and returning nv. Without loss of generality, suppose that if T_0 takes one more step e after E', then T_y running step contention-free after $E \cdot e$ would find the new value in Y.

Next, we show the following execution exists: starting from E', a fast-path T_z writes a new value to Z and commits, then a fast-path T_x reads the old value of X and commits, then T_0 takes one more step (setting Y to the new value), and a fast-path T_y reads the new value of Y.

However, such an execution is not strictly serializable. Indeed, as the value written by T_0 is returned by transaction T_y, T_0 must be committed and precede

T_y in any 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, implying a cycle, which creates the contradiction.

5 Linear Instrumentation Lower Bound

In this section, we focus on a natural progress condition called progressiveness [14] by which a transaction can only abort under read-write or write-write conflict with a concurrent transaction:

Definition 3 (Progressiveness). *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 first prove the following auxiliary lemma concerning progressive HyTMs. It shows that a fast path transaction in a progressive HyTM can contend on a base object only with a conflicting transaction. Intuitively, the proof is based on the observation that, if two non-conflicting transactions, of which one is fast-path, concurrently contend on a base object in some execution, the fast-path transaction may incur a tracking set abort. However, this contradicts the fact that in a progressive HyTM, a transaction may be aborted only due to a conflict.

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 $E \cdot E_1 \cdot E_2$.*

We then notice that Lemma 1 can be extended to prove the following key auxiliary result. If a t-operation of a fast-path transaction does not access any *metadata* base object, then the process executing the transaction cannot distinguish two executions that each export identical histories, *i.e.*, the process cannot tell the difference by only looking at the invocation and responses of the t-operations.

After establishing these auxiliary lemmas, we are ready to prove our main result. We show that read-only fast-path transactions in a progressive opaque HyTM providing *obstruction-free (OF) TM-liveness* (every t-operation running step contention-free returns in a finite number of its own steps) may have to access a *linear* (in the size of their data sets) number of distinct metadata memory locations, even in the absence of concurrency. The complete proof can be found in the technical report [4]; here, we provide a high-level overview of the technique.

Theorem 2. *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.*

Proof (Outline). Let κ be the smallest integer such that some fast-path transaction running step contention-free after a t-complete execution performs κ t-reads and incurs a *capacity* abort. In other words, if a fast-path transaction reads less than κ t-objects, it cannot incur 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 ($|S_m| = 2^{\kappa - m}$) of read-only fast-path transactions such that (1) each transaction in S_m reads m t-objects, (2) the data sets of any two transactions in S_m are disjoint, (3) in the step contention-free execution of any transaction in S_m extending E_m, every t-read accesses at least one distinct metadata base object.

By induction, we assume that the induction statement holds for all $m < \kappa - 1$ (the base case $m = 0$ is trivial) and build E_{m+1} and S_{m+1} satisfying the above condition. Pick any two transactions from the set S_m. We construct E'_m, a t-complete extension of E_m by the execution of a slow-path transaction writing to two distinct t-objects X and Y, such that the two picked transactions, running step contention-free after that, cannot distinguish E_m and E'_m.

Next, we let each of the transactions read one of the two t-objects X and Y. Specifically, we construct the execution E'_m as follows. We first extend E_m with the t-incomplete execution of a slow-path transaction writing to X and Y such that this extension cannot be further extended with the step contention-free executions of either of the picked fast-path transactions performing their m t-reads, followed by the $(m+1)^{th}$ t-read of X or Y that returns the respective "new value."

We show that at least one of the two transactions must access a new metadata base object in this $(m+1)^{th}$ t-read when running step contention-free after this slow-path transaction. Otherwise, the resulting execution would not be opaque. Indeed, without accessing a new metadata base object, such an execution appears t-sequential to the fast-path transactions. This allows us to construct the t-complete execution E'_m such that at least one of the fast-path transactions, running step contention-free after this execution is poised to access a distinct new metadata base object during the $(m+1)^{th}$ t-read.

By repeating this argument for each pair of transactions, we derive that there exists E_{m+1}, a t-complete extension of E_m, such that *at least half* of the transactions in S_m must access a new distinct metadata base object in its $(m+1)^{th}$ t-read when it runs t-sequentially after E_{m+1}. Intuitively, we construct E_{m+1} by "gluing" all these executions E'_m together, which is possible thanks to Lemma 1 and its extensions. These transactions constitute $S_{m+1} \subset S_m$, $|S_{m+1}| = |S_m|/2 = 2^{\kappa-(m+1)}$.

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 where a fast-path transaction is not "visible" to a concurrent (slow-path or fast-path) transaction until it has committed (Observations 1 and 2).

A linear upper bound on instrumentation. We prove that the lower bound in Theorem 2 is tight by describing a progressive opaque HyTM implementation that provides *wait-free TM-liveness* (every t-operation returns in a finite number of its steps) and uses *invisible reads* (read-only transactions do not apply any nontrivial primitives). The algorithm works as follows.

(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.

(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.

(Slow-path 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. An updating slow-path transaction T_k attempts to obtain exclusive write access to its entire write set by performing *compare-and-set* (*cas*) primitive that checks if the value of r_j, for each $X_j \in Wset(T_k)$, is not 1 and, if so, replaces it with 1. 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 T_k is aborted if so. Otherwise, T_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.

Theorem 3. *There exists an opaque HyTM implementation that provides uninstrumented writes, invisible reads, progressiveness and wait-free TM-liveness such that in its every execution E, every read-only fast-path transaction $T \in txns(E)$ accesses $O(|Rset(T)|)$ distinct metadata base objects.*

Providing partial concurrency at low cost. 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 *some* transactions concurrently with constant instrumentation cost, while still keeping invisible reads?

We describe 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 *Count* 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, *Count* 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 *Count* and once all writes to data base objects are completed (this part of the algorithm is identical to the implementation above) or the transaction is aborted, it decrements *Count*. Therefore, $Count \neq 0$ means that at least one slow-path updating transaction is incomplete. A fast-path transaction simply checks if $Count \neq 0$ in the beginning and aborts if so, otherwise, its code is identical to the one above. Note that this way, any update of *Count* automatically causes a tracking set abort of any incomplete fast-path transaction.

7 Related work

The term *instrumentation* was originally used in the context of HyTMs [9, 19, 23] 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 [9]. We prove this formally in this paper.

In [6], 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. The proof techniques we employ are also different.

Uninstrumented HTMs may be viewed as being *disjoint-access parallel (DAP)* [7]. As such, some of the techniques used in the proof of Theorem 1 extend those used in [7, 13, 14]. 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 [5, 10, 18] that integrated HTMs with variants of *DSTM* [16]. 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 2 shows that this overhead is unavoidable. Of note, write operations of these HyTMs are also instrumented, but our result shows that it is not necessary. References [15, 23] provide detailed overviews on HyTM implementations.

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* [9, 23], *PhTM* [19] and others [10, 18] 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, 8] and the use of both direct *and* cached accesses within the same hardware transaction to reduce instrumentation overhead [23, 24]. Another recent approach proposed *reduced hardware transactions* [20], 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. 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 STM, 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.

References

1. Advanced Synchronization Facility Proposed Architectural Specification, March 2009. `http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf`.
2. Y. Afek, A. Levy, and A. Morrison. Software-improved hardware lock elision. In *PODC*. ACM, 2014.
3. D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM.
4. D. Alistarh, J. Kopinsky, P. Kuznetsov, S. Ravi, and N. Shavit. Inherent limitations of hybrid transactional memory. *CoRR*, abs/1405.5689, 2014. http://arxiv.org/abs/1405.5689.
5. C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.
6. H. Attiya and E. Hillel. The cost of privatization in software transactional memory. *IEEE Trans. Computers*, 62(12):2531–2543, 2013.
7. H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

8. I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop*. ACM, 2014.
9. L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In R. Gupta and T. C. Mowry, editors, *ASPLOS*, pages 39–52. ACM, 2011.
10. P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, Oct. 2006.
11. D. Dice, Y. Lev, M. Moir, and D. Nussbaum. 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, pages 157–168, New York, NY, USA, 2009. ACM.
12. A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. 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, pages 99–108, New York, NY, USA, 2011. ACM.
13. R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 304–313, New York, NY, USA, 2008. ACM.
14. R. Guerraoui and M. Kapalka. *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.
15. T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.
16. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.
17. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.
18. S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.
19. Y. Lev, M. Moir, and D. Nussbaum. Phtm: Phased transactional memory. In *In Workshop on Transactional Computing (Transact), 2007. research.sun.com/scalable/pubs/ TRANSACT2007PhTM.pdf*.
20. A. Matveev and N. Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2013.
21. M. Ohmacht. Memory Speculation of the Blue Gene/Q Compute Chip, 2011. `http://wands.cse.lehigh.edu/IBM_BQC_PACT2011.ppt`.
22. J. Reinders. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, 2012.
23. T. Riegel. Software Transactional Memory Building Blocks. 2013.
24. T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 53–64. ACM, 2011.