Inherent Limitations of Hybrid Transactional Memory

Dan Alistarh[1] Justin Kopinsky[4] Petr Kuznetsov[2] Srivatsan Ravi[3*] Nir Shavit[4,5]
[1]Microsoft Research, Cambridge
[2]Télécom ParisTech
[3]TU Berlin
[4]Massachusetts Institute of Technology
[5]Tel Aviv University

1 Introduction

Ever since its introduction by Herlihy and Moss [13], *Transactional Memory (TM)* has promised to be an extremely useful tool, with the power to fundamentally change concurrent programming. It is therefore not surprising that the recently introduced Hardware Transactional Memory (HTM) implementations [1, 17, 18] have been eagerly anticipated and scrutinized by the community.

Early experience with programming HTM, e.g. [2, 7, 8], paints an interesting picture: if used carefully, HTM can be an extremely useful construct, and can significantly speed up and simplify concurrent implementations. At the same time, this powerful tool is not without its limitations: since they 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 [4, 6, 14, 15] have been proposed to complement the fast, but best-effort nature of HTM with a slow, reliable software transactional memory (STM) backup. These proposals have explored a wide range of trade-offs between the overhead on hardware transactions, concurrent execution of hardware and software, and the provided progress guarantees.

Early proposals for HyTM implementations [6, 14] shared some interesting features. First, transactions that do not conflict are expected to run concurrently, regardless of their types (software or hardware). This property is referred to as *progressiveness* [10] and is believed to allow for increased parallelism. Second, in addition to exchanging the values of transactional objects, hardware transactions usually employ *code instrumentation* techniques. Intuitively, instrumentation is used by hardware transactions to detect concurrency scenarios and abort in the case of contention. The number of instrumentation steps performed by these implementations within a hardware transaction is usually proportional to the size of the transaction's data set. Recent work by Riegel et al. [19] surveyed the various HyTM algorithms to date, focusing on techniques to reduce instrumentation overheads in the frequently executed hardware fast-path. However, it is not clear whether there are fundamental limitations when building a HyTM with non-trivial concurrency between hardware and software transactions.

In particular, what are the inherent instrumentation costs of building a HyTM, and what are the trade-offs between these costs and the provided *concurrency*, i.e., the ability of the HyTM system to run software and hardware transactions in parallel?

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 precisely defines

*Contact author: srivatsan@srivatsan.in, FG INET, MAR 4-4, Marchstr. 23, 10587 Berlin, Germany

instrumentation costs in a quantifiable way.

We model a hardware transaction as a series of memory accesses that operate on locally cached copies of the variables, followed by a *cache-commit* operation. In case a concurrent transaction performs a (read-write or write-write) conflicting access to a cached object, the cached copy is invalidated and the hardware transaction aborts.

Our model for instrumentation is motivated by recent experimental evidence which suggests that the overhead on hardware transactions imposed by code which detects concurrent software transactions is a significant performance bottleneck [16]. In particular, we say 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* read and updated within transactions. We quantify instrumentation cost by measuring the number of accesses to *metadata objects* which transactions perform.

Once this general model is in place, we investigate lower bounds on the cost of implementing a HyTM. We prove two main results. First, we show that some instrumentation is necessary in a HyTM implementation even if we only intend to provide *sequential* progress, where a transaction is only guaranteed to commit if it runs in the absence of concurrency. Second, we prove that any progressive HyTM implementation that maintains invisible reads by hardware transactions has executions in which an arbitrarily long read-only hardware transaction running in the absence of concurrency *must* access a number of distinct metadata objects proportional to the size of its data set. We show that the lower bound is tight by presenting a matching HyTM algorithm which additionally allows for uninstrumented writes.

The high instrumentation costs of early HyTM designs, which we show to be inherent, stimulated more recent HyTM schemes [4,15,16,19] to sacrifice progressiveness for *constant* instrumentation cost (i.e., not depending on the size of the transaction). In the past two years, Dalessandro et al. [4] and Riegel et al. [19] have proposed HyTMs based on the efficient *NOrec STM* [5]. These HyTMs schemes do not guarantee any parallelism among transactions; only sequential progress is ensured. Despite this, they are among the best-performing HyTMs to date due to the limited instrumentation in the hardware fast-path.

Starting from this observation, we provide a more precise upper bound for *low-instrumentation* HyTMs by presenting a HyTM algorithm 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. However, transactions are guaranteed to commit only if they do not run concurrently with an updating software transaction (or exceed capacity). Therefore, the cost of avoiding the linear lower bound for progressive implementations is that hardware transactions may be aborted by non-conflicting software transactions.

In sum, this work captures for the first time an inherent trade-off between the degree of concurrency a HyTM implementation provides between hardware and software transactions and the amount of instrumentation overhead the implementation must incur.

2 Hybrid transactional memory model

Our baseline model for transactional memory systems is the standard software TM model [11]. In particular, we assume the transactions export an interface to perform *transactional operations* (*t-operations*) *read* and *write* and *transactional objects* (*t-objects*). Process execute these operations by applying primitives (described below) to shared *base objects*. In this section, we describe the operation of a *Hybrid Transactional Memory (HyTM)* implementation, in which conventional memory accesses are combined with hardware transactions modelled as *cached* accesses.

Direct accesses and cached accesses. We assume that every base object can be accessed with two kinds of primitives, *direct* and *cached*, each instantiating a generic read-modify-write (rmw) primitive [9, 12]. 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 is *trivial* if it never affects the value of a base object, otherwise it is *nontrivial*.

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 (initially empty). One can think of τ_i as the *L1 cache* of process i. A series of cached rmw primitives performed on τ_i followed by a *cache-commit* primitive models a *hardware transaction*.

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*). Otherwise, 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 as follows. 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 each subsequent cached primitive invoked by i 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 operation returns *commit*.

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 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 $\perp$.

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.

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 pending 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 $T_m \in txns(E)$, E is indistinguishable to T_m 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 via tracking sets.

Instrumentation. We now define *code instrumentation* in fast-path transactions. We first require a technical definition, which we give informally here. The precise definition is available in the full

version of this paper [3]. An execution E of a HyTM $\mathcal{M}$ *appears sequential* to a transaction T_k which participates in E if there exists an execution E' of $\mathcal{M}$ such that there are no pending transactions in E' and the execution in which T_k runs alone after E' is indistinguishable to T_k from E.

Given a HyTM implementation $\mathcal{M}$ operating on a set of t-objects $\mathcal{X}$, we partition the set of base objects accessed by $\mathcal{M}$ into a set $\mathbb{D}$ of *data* objects and a set $\mathbb{M}$ of *metadata* objects, where $\mathbb{D} \cap \mathbb{M} = \emptyset$. 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 $\mathcal{X}$, $\mathbb{D}_X \cap \mathbb{D}_Y = \emptyset$, and the following properties hold in every execution E of $\mathcal{M}$:

1. Every transaction $T_k \in txns(E)$ only accesses base objects in $\mathbb{M} \cup \bigcup_{X \in DSet(T_k)} \mathbb{D}_X$.
2. For any execution E and any transaction T_k which participates in E, let E' be the execution given by E followed by a single step of T_k. If that step of T_k applied a primitive to a base object in $\mathbb{D}$ and E appears sequential to T_k, then E' also appears sequential to T_k.

Intuitively, the second condition means that base objects in $\mathbb{D}$ may only contain the "values" of t-objects, so they cannot be used to detect concurrent transactions.

We now define a HyTM to be *uninstrumented* if transactions cannot access metadata (i.e. base objects in $\mathbb{M}$) in any execution.

Definition 1. *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) transaction T_k, all primitives performed by T_k in E are performed on base objects in $\mathbb{D}$.*

We make the following observation about uninstrumented transactions:

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 that runs alone after E, the execution E appears sequential to T_k.*

3 On the cost of instrumentation and concurrency

The following theorems can be proven using the model presented in the previous section. The proofs are available in the full version of this paper [3].

Sequential progress. Informally, a HyTM implementation $\mathcal{M}$ guarantees sequential TM-progress for fast-path transactions (and resp. slow-path) if every fast-path transaction (and resp. slow-path) which runs solo from a quiescent configuration commits. The following results concern the instrumentation costs of such implementations.

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

Theorem 2. *There exists an opaque HyTM implementation $\mathcal{M}$ that provides uninstrumented writes and sequential TM-progress for fast-path transactions such that in every execution E of $\mathcal{M}$, every fast-path transaction accesses at most one metadata base object.*

Progressiveness. Informally, a HyTM implementation $\mathcal{M}$ guarantees *progressiveness* if every transaction which does not conflict on a t-object with any concurrent transaction commits. The following theorems show that there is a significant inherent instrumenation cost on progressive implementations.

Theorem 3. *Let $\mathcal{M}$ be any progressive, opaque HyTM implementation that provides invisible fast-path reads. For every $m \in \mathbb{N}$, there exists an execution in which a fast-path read-only transaction T_k satisfies either (1) $|Dset(T_k)| \leq m$ and T_k incurs a capacity abort or (2) $|Dset(T_k)| = m$ and T_k accesses $\Omega(m)$ distinct metadata base objects.*

Theorem 4. *There exists an opaque HyTM implementation $\mathcal{M}$ that provides uninstrumented writes, invisible reads and progressiveness 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.*

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] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] R. Guerraoui and M. Kapalka. Transactional memory: Glimmer of a theory. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 1–15, Berlin, Heidelberg, 2009. Springer-Verlag.

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

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

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

[14] 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.

[15] 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.*

[16] 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.

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

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

[19] 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.