Cost of Concurrency in Hybrid Transactional Memory

Trevor Brown   Srivatsan Ravi
University of Toronto   Purdue University

Abstract

State-of-the-art *software transactional memory (STM)* implementations achieve good performance by carefully avoiding the overhead of *incremental validation* (i.e., re-reading previously read data items to avoid inconsistency) while still providing *progressiveness* (allowing transactional aborts only due to *data conflicts*). Hardware transactional memory (HTM) implementations promise even better performance, but offer no progress guarantees. Thus, they must be combined with STMs, leading to *hybrid* TMs (HyTM) in which hardware transactions must be *instrumented* (i.e., access metadata) to detect contention with software transactions.

We show that, unlike in progressive STMs, software transactions in progressive HyTMs cannot avoid incremental validation. In fact, this result holds even if hardware transactions can *read* metadata *non-speculatively*. We then present *opaque* HyTM algorithms providing *progressiveness for a subset of transactions* that are optimal in terms of hardware instrumentation. We explore the concurrency vs. hardware instrumentation vs. software validation tradeoffs for these algorithms. Preliminary experiments with Intel's HTM seem to suggest that the inherent *cost to concurrency* in HyTMs also exists in practice. Finally, we discuss algorithmic techniques for cicumventing this cost.

1. Introduction

The *Transactional Memory (TM)* abstraction is a synchronization mechanism that allows the programmer to *speculatively* execute sequences of shared-memory operations as *atomic transactions*. Several software TM designs [6, 9, 11, 16, 20] have been introduced subsequent to the original proposal TM proposal based in hardware [12]. The original dynamic STM implementation DSTM [11] ensures *progressiveness*: a transaction aborts only if there is a read-write *data conflict* with a concurrent transaction. However, read operations in DSTM must *incrementally* validate the responses of all previous read operations to avoid inconsistent executions. This results in a quadratic (in the size of the transaction's read set) step-complexity bound. Subsequent STM implementations like NOrec [6] and TL2 [8] minimize the impact on performance due to incremental validation. NOrec uses a global sequence lock that is read at the start of a transaction and performs *value-based* validation during read operations only if the value of the global lock has changed (by an updating transaction) since reading it. TL2, on the other hand, eliminates incremental validation completely. Like NOrec, it uses a global sequence lock, but each data item also has an associated sequence lock value that is updated alongside the data item. When a data item is read, if its associated sequence lock value is different from the value that was read from the sequence lock at the start of the transaction, then the transaction aborts.

In fact, STMs like TL2 and NOrec ensure progress in the absence of data conflicts with $O(1)$ step complexity read operations and *invisible reads* (read operations do not apply any nontrivial primitives on the shared memory). Nonetheless, TM designs that are implemented entirely in software still incur significant performance overhead. Thus, current CPUs have included instructions to mark a block of memory accesses as transactional [1, 15, 18], allowing them to be executed *atomically* in hardware. Hardware transactions promise better performance than STMs, but they offer no progress guarantees since they may experience *spurious* aborts. This motivates the need for *hybrid* TMs in which the *fast* hardware transactions are complemented with *slower* software transactions that do not have spurious aborts.

To allow hardware transactions in a HyTM to detect conflicts with software transactions, they must be *instrumented* to perform additional metadata accesses, which introduces overhead. Hardware transactions typically provide automatic conflict detection at cacheline granularity, thus ensuring that the transaction itself would be aborted if it experiences memory contention on the cacheline. This is at least the case with the Intel's Transactional Synchronization Extensions [21]. The IBM Power8 ISA additionally allows hardware transactions to access metadata *non-speculatively*, thus bypassing automatic conflict detection. While this has the advantage of potentially reducing contention aborts in hardware, this makes the design of HyTM implementations potentially harder to prove correct.

In [2], it was shown that hardware transactions in progressive HyTMs must perform at least one metadata access per transactional read and write. In this paper, we show that in progressive HyTMs with invisible reads, software transactions *cannot* avoid incremental validation. Specifically, we prove that each read operation of a software transaction in a progressive HyTM must necessarily incur a validation cost that is *linear* in the size of the transaction's read set. This is in contrast to TL2 which is progressive and has constant complexity read operations. Thus, in addition to the linear instrumentation cost on the hardware transactions, there is the quadratic step complexity cost on the software transactions.

We then present *opaque* HyTM algorithms providing *progressiveness for a subset of transactions* that are optimal in terms of hardware instrumentation. Algorithm 1 is progressive for all transactions, but it incurs high instrumentation overhead in practice. Algorithm 2 avoids all instrumentation in fast-path read operations, but is progressive only for slow-path reading transactions. We also sketch how *some* hardware instrumentation can be performed *non-speculatively* without violating opacity.

We performed preliminary experiments comparing our HyTM algorithms to TL2, Transactional Lock Elision (TLE) and Hybrid NOrec [19] using a binary search tree microbenchmark. In these experiments, we studied two types of workloads: workloads in which essentially all transactions commit on the fast-path, and workloads in which some thread periodically performs transactions on the slow-path. These experiments demonstrate that hardware instrumentation is a dominating factor in the performance of HyTMs, and that simplistic algorithms like TLE perform very well unless transactions periodically run on the slow-path.

Viewed collectively, our results demonstrate that there is an inherent cost to concurrency in HyTMs.

2. Hybrid transactional memory (HyTM)

In this section, we adopt the formal model of HyTMs originally proposed in [2].

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*.

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 $\mathcal{M}$.

We consider the dynamic programming model: the *read set* (resp., the *write set*) of a transaction T_k in an execution E, denoted $Rset_E(T_k)$ (and resp. $Wset_E(T_k)$), is the set of t-objects that T_k attempts to read (and resp. write) by issuing a t-read (and resp. t-write) invocation in E (for brevity, we sometimes omit the subscript E from the notation).

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. Two executions E and E' are *indistinguishable* to a set $\mathcal{T}$ of transactions, if for each transaction $T_k \in \mathcal{T}$, $E|k = E'|k$. A 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*.

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

Hybrid transactional memory executions. We now describe the execution model of a *Hybrid transactional memory (HyTM)* implementation. In our HyTM model, shared memory configurations may be modified by accessing base objects via two kinds of primitives: *direct* and *cached*. (i) 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. (ii) 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.

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. A base object b is *present* in τ_i with mode m if $\exists v, (b, v, m) \in \tau_i$.

Hardware aborts. 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 event invoked by i sets τ_i to $\emptyset$ and returns $\perp$. We refer to this event as a *tracking set abort*.

Any transaction $T_k \in txns(E)$ that performs at least one cached access necessarily performs a *cache-commit* primitive as the last event of $E|k$. A *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*.

Slow-path and fast-path transactions. We partition HyTM transactions into *fast-path transactions* and *slow-path transactions*. 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 direct rmw primitive on a base object. A fast-path transaction essentially encapsulates a hardware transaction. Specifically, in any execution E, we say that a transaction $T_k \in txns(E)$ is a fast-path transaction if $E|k$ contains at least one cached event. An event of a *hardware transaction* includes series of direct trivial accesses and at least one cached access followed by a *cache-commit* primitive.

We assume that a fast-path transaction T_k returns A_k as soon an underlying cached primitive or *cache-commit* returns $\perp$. This implies the following observation:

Observation 1. *For any t-incomplete transaction $T_k \in txns(E)$ executed by process i and $(b, v, exclusive) \in \tau_i$ (and resp. $(b, v, shared) \in \tau_i$) after execution E and let e be any event (and resp. nontrivial event) that some process $j \neq i$ is poised to apply after E, then the next event of T_k in any extension of $E \cdot e$ is A_k.*

Non-cached reads inside fast-path. Note that we specifically allow hardware transactions to perform reads without adding the corresponding base object to the process's tracking set, thus modelling the non-speculative accesses inside hardware allowed by IBM Power8 architectures. We remark that Intel's HTM does not support this feature: an event of a hardware transaction does not include any direct access.

HyTM properties. Throughout this paper, we consider the TM-correctness property of *opacity* [10]: an execution E is opaque if there exists a *legal* sequential execution S equivalent to some t-completion of E that respects the *real-time ordering* of transactions in E.

We also assume a weak TM-liveness property for t-operations in this paper: every t-operation returns a matching response within a finite number of its own steps if running step-contention free from a *quiescent* configuration, *i.e.*, a configuration in which every transaction is t-complete.

Algorithms and lower bounds presented in this paper concern HyTMs that provide *invisible reads*: t-read operations do not perform nontrivial primitives in any execution.

3. Progressive HyTM must perform incremental validation

In this section, we show that it is impossible to implement opaque *progressive* HyTMs with *invisible reads* with $O(1)$ step-complexity read operations for slow-path transactions. This result holds even if fast-path transactions may perform direct trivial accesses.

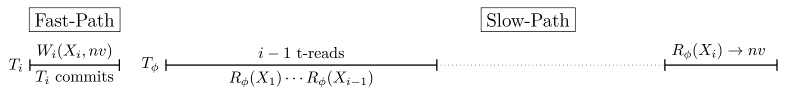
Formally, we say that a HyTM implementation $\mathcal{M}$ is progressive for a set $\mathcal{T}$ of transactions if in any execution E of $\mathcal{M}$; $\mathcal{T} \subseteq txns(E)$, if any transaction $T_k \in \mathcal{T}$ returns A_k in E, there exists another concurrent transaction T_m that *conflicts* (both access the same t-object and at least one writes) with T_k in E.

Theorem 2. *Let $\mathcal{M}$ be any progressive opaque HyTM implementation providing invisible reads. There exists an execution E of $\mathcal{M}$ and some slow-path read-only transaction $T_k \in txns(E)$ that incurs a time complexity of $\Omega(m^2)$; $m = |Rset(T_k)|$.*

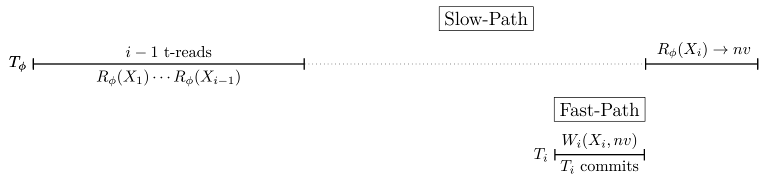
For the proof, we construct an execution of a read-only slow-path transaction T_ϕ that performs $m \in \mathbb{N}$ distinct t-reads of t-objects $X_1, \ldots, X_m$. We show inductively that for each $i \in \{1, \ldots, m\}$; $m \in \mathbb{N}$, the i^{th} t-read must access $i - 1$ distinct base objects during its execution. The (partial) steps in our execution are depicted in Figure 1.

How STM implementations avoid the lower bound. NOrec [6] is a progressive opaque STM that minimizes the average step-complexity resulting from incremental validation of t-reads. Transactions read a global versioned lock at the start, and perform value-based validation during t-read operations *iff* the global version has changed. TL2 [8] improves over NOrec by circumventing the lower bound of Theorem 2. Concretely, TL2 associates a global version with each t-object updated during a transaction and performs validation with $O(1)$ complexity during t-reads by simply verifying if the version of the t-object is greater than the global version read at the start of the transaction. Technically, NOrec and algorithms in this paper provide a stronger definition of progressiveness: a transaction may abort only if there is a prefix in which it conflicts with another transaction and both are t-incomplete. TL2 on the other hand allows a transaction to abort due to a concurrent conflicting transaction.

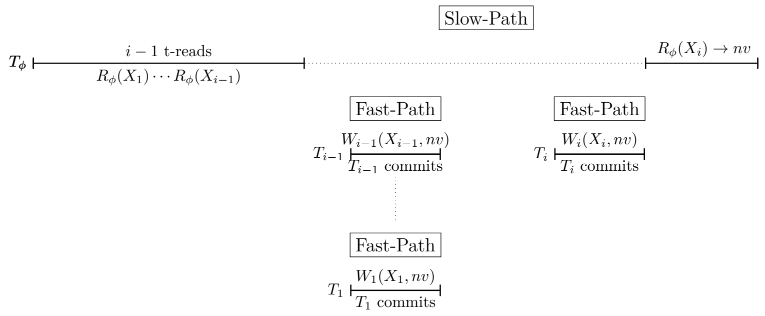
The proof of the lemma below is a simple extension of the analogous lemmas from [3] allowing direct trivial



(a) Slow-path transaction T_ϕ performs $i-1$ distinct t-reads followed by the t-read of X_i that returns value nv writtten by fast-path transaction T_i



(b) Fast-path transaction T_i does not contend with any of the $i-1$ t-reads performed by T_ϕ; this execution is indistinguishable to T_ϕ from 1a and t-read of X_i must return nv



(c) To distinguish the $i-1$ different executions, t-read of X_i by T_ϕ is forced to access $i-1$ different base objects

Figure 1: Proof steps for Theorem 2

accesses inside fast-path transactions. This intuively follows from the fact that the tracking set of a process executing a fast-path transaction is invalidated due to contention on a base object with another transaction (cf. Observation 1).

Lemma 3. *Let $\mathcal{M}$ be any progressive HyTM implementation in which fast-path transactions may perform trivial direct accesses. Let $E_1 \cdot E_2$ be an execution of $\mathcal{M}$ where E_1 (and resp. E_2) is the step contention-free execution fragment of fast-path transaction T_1 (and resp. T_2), T_1 and T_2 do not conflict in $E_1 \cdot E_2$, 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 $E_1 \cdot E_2$.*

The formal proof of the theorem follows.

Proof. We construct an execution of a progressive opaque HyTM in which every t-read performed by a read-only slow-path transaction must access linear (in the size of the read set) number of distinct base objects.

For all $i \in \{1, \ldots, m\}$; $m \in \mathbb{N}$, let v be the initial value of t-object X_i. Let π^m denote the complete step contention-free execution of a slow-path transaction T_ϕ that performs m t-reads: $read_\phi(X_1)\cdots read_\phi(X_m)$ such that for all $i \in \{1, \ldots, m\}$, $read_\phi(X_i) \rightarrow v$.

Claim 4. *For all $i \in \mathbb{N}$, $\mathcal{M}$ has an execution of the form $\pi^{i-1} \cdot \rho^i \cdot \alpha^i$ where,*

- *π^{i-1} is the complete step contention-free execution of slow-path read-only transaction T_ϕ that performs $(i-1)$ t-reads: $read_\phi(X_1)\cdots read_\phi(X_{i-1})$,*
- *ρ^i is the t-complete step contention-free execution of a fast-path transaction T_i that writes $nv_i \neq v_i$ to X_i and commits,*
- *α^i is the complete step contention-free execution fragment of T_ϕ that performs its i^{th} t-read: $read_\phi(X_i) \rightarrow nv_i$.*

For each $i \in \{2, \ldots, m\}$, $j \in \{1, 2\}$ and $\ell \leq (i-1)$, we now define an execution of the form $\mathbb{E}^i_{j\ell} = \pi^{i-1} \cdot \beta^\ell \cdot \rho^i \cdot \alpha^i_j$ as follows:

- β^ℓ is the t-complete step contention-free execution fragment of a fast-path transaction T_ℓ that writes $nv_\ell \neq v$ to X_ℓ and commits
- α^i_1 (and resp. α^i_2) is the complete step contention-free execution fragment of $read_\phi(X_i) \rightarrow v$ (and resp. $read_\phi(X_i) \rightarrow A_\phi$).

Claim 5. *For all $i \in \{2, \ldots, m\}$ and $\ell \leq (i-1)$, $\mathcal{M}$ has an execution of the form $\mathbb{E}^i_{1\ell}$ or $\mathbb{E}^i_{2\ell}$.*

The proof of the above claim is immediate. We now show that for all $i \in \{2, \ldots, m\}$, $j \in \{1, 2\}$ and $\ell \leq (i-1)$, slow-path transaction T_ϕ must access $(i-1)$ different base objects during the execution of $read_\phi(X_i)$ in the execution $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i \cdot \alpha^i_j$.

Consider the $(i-1)$ different executions: $\pi^{i-1} \cdot \beta^1 \cdot \rho^i$, …, $\pi^{i-1} \cdot \beta^{i-1} \cdot \rho^i$ (cf. Figure 1c). For all $\ell, \ell' \leq (i-1); \ell' \neq \ell$, $\mathcal{M}$ has an execution of the form $\pi^{i-1} \cdot \beta^\ell \cdot \rho^i \cdot \beta^{\ell'}$ in which fast-path transactions T_ℓ and $T_{\ell'}$ access mutually disjoint data sets. By invisible reads and Lemma 3, the pairs of transactions $T_{\ell'}$, T_i and $T_{\ell'}$, T_ℓ do not contend on any base object in this execution. This implies that $\pi^{i-1} \cdot \beta^\ell \cdot \beta^{\ell'} \cdot \rho^i$ is an execution of $\mathcal{M}$ in which transactions T_ℓ and $T_{\ell'}$ each apply nontrivial primitives to mutually disjoint sets of base objects in the execution fragments β^ℓ and $\beta^{\ell'}$ respectively.

This implies that for any $j \in \{1, 2\}$, $\ell \leq (i-1)$, the configuration C^i after E^i differs from the configurations after $\mathbb{E}^i_{j\ell}$ only in the states of the base objects that are accessed in the fragment β^ℓ. Consequently, slow-path transaction T_ϕ must access at least $i-1$ different base objects in the execution fragment π^i_j to distinguish configuration C^i from the configurations that result after the $(i-1)$ different executions $\pi^{i-1} \cdot \beta^1 \cdot \rho^i$, …, $\pi^{i-1} \cdot \beta^{i-1} \cdot \rho^i$ respectively.

Thus, for all $i \in \{2, \ldots, m\}$, slow-path transaction T_ϕ must perform at least $i-1$ steps while executing the i^{th} t-read in π^i_j. □

4. Hybrid transactional memory algorithms

In this section, we present progressive opaque HyTM algorithms that subject to the lower bound of Theorem 2 and then discuss some techniques to circumvent the lower bound.

4.1 Instrumentation-optimal progressive HyTMs

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 *sequence lock* r_j. The sequence lock is an unsigned integer whose LSB bit stores the *locked* state. Specifically, we say that process p_i *holds a lock on* X_j *after an execution* E if or_j & $1 = 1$ after E, where or_j is the value of r_j after E.

Fast-path transactions: For a fast-path transaction T_k executed by process p_i, the $read_k(X_j)$ implementation first reads r_j (uncached) and returns A_k if some other process p_j holds a lock on X_j. Otherwise, it returns the value of X_j. Updating fast-path transactions As with $read_k(X_j)$, the $write(X_j, v)$ implementation returns A_k if some other process p_j holds a lock on X_j. Process p_i then increments the value of r_j by 2 via a direct access and stores the cached state of X_j along with its value v. If the cache has not been invalidated, p_i updates the shared memory during $tryC_k$ by invoking the *commit-cache* primitive.

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 se, adds r_j to $Rset(T_k)$ and then performs *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. Validation of the read set is performed by re-reading the values of the sequence lock entires stored in $Rset(T_k)$.

Slow-path updating transactions: 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 unchanged since last reading it during $write_k(X.v)$ If all the locks on the write set were acquired successfully, T_k performs validation of the read set and returns C_k if successful, else p_i aborts the transaction.

Non-cached accesses inside fast-path: As indicated in the pseudocode of Algorithm 1, some accesses may be performed uncached (as allowed in IBM Power 8) and the resulting implementation would still be opaque.

Instrumentation-optimal HyTMs that are progressive only for a subset of transactions. Algorithm 2 does not incur the linear instrumentation cost on the fast-path reading transactions (as in Algorithm 1, but provides progressiveness only for slow-path reading transactions.

4.2 Minimizing the cost for incremental validation in opaque HyTMs

Observe that the lower bound in Theorem 2 assumes progressiveness for both slow-path and fast-path transactions along with opacity and invisible reads. In this section, we suggest algorithmic ideas for cirvumventing the lower bound or minimizing the cost incurred by implementations due to incremental validation. Figure 2 summarizes the complexity costs associated with the HyTM algorithms considered in this paper.

Sacrificing progressiveness and minimizing contention window. *Hybrid NOrec* [5] is a HyTM implementation that does not satisfy progressiveness (unlike its STM counterpart NOrec), but mitigates the step-complexity cost on slow-path transactions by performing incremental validation during a transactional read *iff* the shared memory has changed since the start of the transaction. Conceptually, hybrid NOrec uses a global sequence lock gsl that is incremented at the start and end of each transaction's commit procedure. Readers

Algorithm 1 Progressive fast-path and slow-path opaque HyTM implementation; code for transaction T_k

```
Shared objects
    $v_j$, value of each t-object $X_j$
    $r_j$, a sequence lock for each t-object $X_j$

Code for fast-path transactions
read$_k$($X_j$)
    ov$_j$ := v$_j$                          ▷ cached read
    or$_j$ := r$_j$                          ▷ uncached read
    if or$_j$ & 1 then return $A_k$
    return ov$_j$
write$_k$($X_j$, $v$)
    or$_j$ := read($r_j$)                    ▷ cached read
    if or$_j$ & 1 then return $A_k$
    r$_j$ := or$_j$+2
    v$_j$ := v                              ▷ uncached write
    return OK
tryC$_k$()
    commit-cache$_i$
Function: release(Q)
    for each X$_j$ ∈ $Q$
        r$_j$ := or$_j$+1
Function: acquire(Q)
    for each X$_j$ ∈ $Q$
        if r$_j$.setV()
            Lset(T$_k$) := Lset(T$_k$) ∪ {X$_j$}
            return true
        release(Lset(T$_k$))
        return false
Code for slow-path transactions
Read$_k$(X$_j$)
    if X$_j$ ∈ Wset(T$_k$) then return Wset(T$_k$).locate(X$_j$)
    or$_j$ := r$_j$
    ov$_j$ := v$_j$
    Rset(T$_k$) := Rset(T$_k$)∪ {X$_j$,or$_j$}
    if or$_j$ & 1 then return $A_k$
    if not validate() then return $A_k$
    return ov$_j$
write$_k$($X_j$, $v$)
    or$_j$ := r$_j$
    nv$_j$ := v
    if or$_j$ & 1 then return $A_k$
    Wset(T$_k$) := Wset(T$_k$) ∪ {X$_j$, nv$_j$, or$_j$}
    return OK
tryC$_k$()
    if Wset(T$_k$) = ∅ then return $C_k$
    if not acquire(Wset(T$_k$)) then return $A_k$
    if not validate()
            release(Wset(T$_k$))
            return $A_k$
    for each X$_j$ ∈ Wset(T$_k$)
            v_j:= nv$_j$
            release(Wset(T$_k$))
            return $C_k$
Function: validate()
    if ∃ X$_j$ ∈ Rset(T$_k$):or$_j$ ≠ r$_j$ then return false
    return true
```

	Algorithm 1	Algorithm 2	TLE	HybridNorec
Instrumentation in fast-path reads	per-read	none	none	none
Instrumentation in fast-path writes	per-write	per-write	constant	none
Validation in slow-path reads	$\Omega(\lvert Rset\rvert)$	$\Omega(\lvert Rset\rvert)$	None	$\Omega(\lvert Rset\rvert)$ only if concurrency
h/w-s/f concurrency	prog.	prog. for slow-path readers	zero	not prog., but small contention window
Uncached accesses inside fast-path	yes	yes	no	yes
opacity	yes	yes	Yes	Yes

Figure 2: Table summarizing complexities of HyTM implementations

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

```
Shared objects
    L, global single-bit lock

Code for fast-path transactions
start$_k$()
    l := L                                  ▷ cached read
    if l & 1 ≠ 0 then return $A_k$
read$_k$(X$_j$)
    ov$_j$ := v$_j$                          ▷ cached read
    return ov$_j$
write$_k$(X$_j$, v)
    or$_j$ := r$_j$                          ▷ uncached read
    r$_j$ := or$_j$ + 2                      ▷ uncached write
    v$_j$ := v                              ▷ cached write
    return OK
try$C_k$()
    return commitCache$_i$()                ▷ returns $C_k$ or $A_k$
Code for slow-path transactions
tryC$_k$()
    if Wset(T$_k$) = ∅ then return $C_k$
    while not flag do flag := CAS(L, 0, 1)
    for each X$_j$ ∈ Wset(T$_k$)
        nr$_j$ := r$_j$
        if or$_j$ ≠ nr$_j$ then
            release(Wset(T$_k$))
            return $A_k$
    for each X$_j$ ∈ Wset(T$_k$) do r$_j$ := nr$_j$ + 1
    if validate() then
        release(Wset(T$_k$))
        return $A_k$
    for each X$_j$ ∈ Wset(T$_k$) do v$_j$ := nv$_j$
    release(Wset(T$_k$)); return $C_k$
Function: release(Q)
    for each X$_j$ ∈ Q do r$_j$ := nr$_j$ + 1
    L := 0; return OK
```

can use the value of gsl to determine whether shared memory has changed between two configurations. Unfortunately, with this approach, two fast path transactions will always conflict on the gsl if their commit procedures are concurrent. To reduce the contention window for fast path transactions, the gsl is actually implemented as two separate locks. A slow path transaction locks both esl and gsl while it is committing. Instead of incrementing gsl, a fast path transaction checks if esl is locked and aborts if it is. Then, at the end of the fast path transaction's commit procedure, it increments gsl twice (quickly locking and releasing it and immediately commits in hardware), thus, the window for fast path transactions to content on gsl is very small.

Employing an uninstrumented fast fast-path. We describe how every transaction may first be executed in a "fast" fast-path with almost no instrumentation and if unsuccessful, may be re-attempted in the fast-path and subsequently in slow-path. Specifically, we describe a generic transformation for any opaque HyTM $\mathcal{M}$ to an opaque HyTM $\mathcal{M}'$ by employing a shared *fetch-and-add* metadata F that slow-path updating transactions increment (and resp. decrement) at the start (and resp. end). The fast fast-path checks first checks if F is 0 and if not, aborts the transaction; otherwise the transaction is continued as an uninstrumented hardware transaction. Since there is no concurrency between fast fast-path and slow-path, opacity is immediate.

5. Evaluation

In this section, we study the performance characteristics of Algorithms 1 and 2, Hybrid NOrec, TLE and TL2. Our experimental goals are (G1) to study the performance impact of instrumentation on the fast-path and validation on the slow path, (G2) to understand how Algorithms 1 and 2 perform relative to the other algorithms, and (G3) to determine whether non-speculative accesses can be used to obtain significant performance improvements. We discuss (G1) and (G2), here. We are currently in the process of investigating (G3).

Experimental system. The experimental system is a large-scale 2-socket Intel E7-4830 v3 with 12 cores per socket and 2 hyperthreads (HTs) per core, for a total of 48 threads. Each core has a private 32KB L1 cache and 256KB L2 cache (which is shared between HTs on a core). All cores on a socket share a 30MB L3 cache. This system has a non-uniform memory architecture (NUMA) in which threads have significantly different access costs to different parts of memory depending on which processor they are currently executing on.

We pin threads so that the first socket is saturated before we place any threads on the second socket. Thus, thread counts 1-24 run on a single socket. Furthermore, hyperthreading is engaged on the first socket for thread counts 13-24, and on the second socket for thread counts 37-48. Consequently, our graphs clearly show the effects of NUMA and hyperthreading.

The machine has 128GB of RAM, and runs Ubuntu 14.04 LTS. All code was compiled with the GNU C++ compiler (G++) 4.8.4 with build target x86_64-linux-gnu and compilation options `-std=c++0x -O3 -mx32`.

Hybrid TM implementations. For TL2, we used the implementation published by its authors. We implemented the other algorithms in C++. Each hybrid TM algorithm first attempts to execute a transaction on the fast path, and will continue to execute on the fast path until the transaction has experienced 20 aborts, at which point it will fall back to the slow path.

Methodology. We used a simple unbalanced binary search tree (BST) microbenchmark as a vehicle to study the performance of our implementations. The BST implements a dictionary, which contains a set of keys, each with an associated value. For each TM algorithm and update rate $U \in \{40, 10, 0\}$, we run six timed *trials* for several thread counts n. Each trial proceeds in two phases: *prefilling* and *measuring*. In the prefilling phase, n concurrent threads perform 50% *Insert* and 50% *Delete* operations on keys drawn uniformly randomly from $[0, 10^5)$ until the size of the tree converges to a steady state (containing approximately $10^5/2$ keys). Next, the trial enters the measuring phase, during which threads begin counting how many operations they perform. In this phase, each thread performs $(U/2)$% *Insert*, $(U/2)$% *Delete* and $(100-U)$% *Search* operations, on keys/values drawn uniformly from $[0, 10^5)$, for one second.

Uniformly random updates to an unbalanced BST have been proven to yield trees of logarithmic height with high probability. Thus, in this type of workload, almost all transactions succeed in hardware, and the slow path is almost never used. To study performance when transactions regularly run on the slow path, we introduced another operation called a *RangeIncrement* that often fails in hardware and must run on the slow path. A *RangeIncrement*(low, hi) atomically increments the values associated with each key in the range $[low, hi]$ present in the tree. Note that a *RangeIncrement* is more likely to experience data conflicts and capacity aborts than BST updates, which only modify a single node.

We consider two types of workloads: (W1) all n threads perform *Insert*, *Delete* and *Search*, and (W2) $n-1$ threads perform *Insert*, *Delete* and *Search* and one thread performs only *RangeIncrement* operations. Figure 3 shows the results for both types of workloads.

Results. We first discuss the 0% updates graph for workload type W1. In this graph, essentially all operations committed in hardware. In fact, in each trial, a small fraction of 1% of operations ran on the slow-path. Thus, any performance differences shown in the graph are essentially differences in the performance of the algorithms' respective fast-paths (with the exception of TL2). Algorithm 1, which has instrumentation in its fast-path read operations, has significantly lower performance than Algorithm 2, which does not. Since this is a read-only workload, this instrumentation is responsible for the performance difference.

In the W1 workloads, TLE, Algorithm 2 and Hybrid NOrec perform similarly (with a small performance advantage for Hybrid NOrec at high thread counts). This is because the fast paths for these three algorithms have

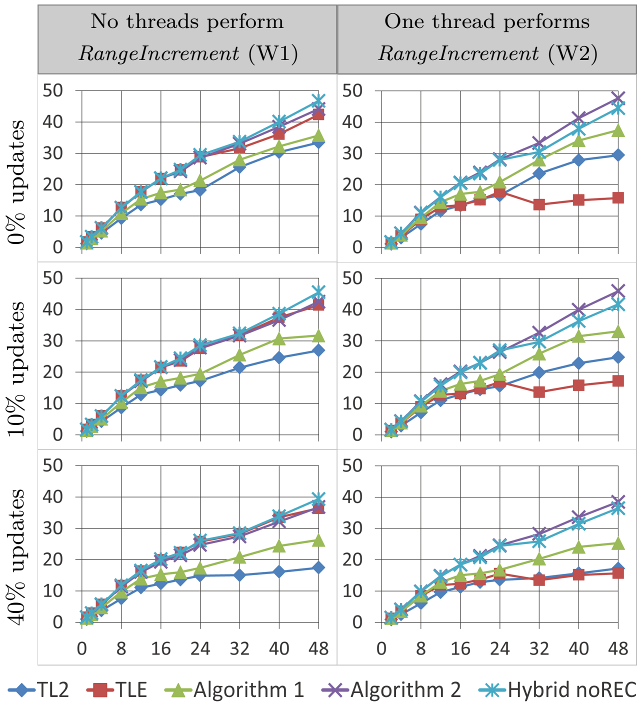


Figure 3: Results for a **BST microbenchmark**. The x-axis represents the number of concurrent threads. The y-axis represents operations per microsecond.

similar amounts of instrumentation. In each algorithm, there is no instrumentation for reads or writes, and the transaction itself incurs one or two metadata accesses.

In contrast, in the W2 workloads, TLE performs quite poorly, compared to the HyTM algorithms. In these workloads, transactions must periodically run on the slow-path, and in TLE, this entails acquiring a global lock that restricts progress for all other threads. At high thread counts this significantly impacts performance. Its performance decreases as the sizes of the ranges passed to *RangeIncrement* increase. Its performance is also negatively impacted by NUMA effects at thread counts higher than 24. (This is because, when a thread p reads the lock and incurs a cache miss, if the lock was last held by another thread on the same socket, then p can fill the cache miss by loading it from the shared L3 cache. However, if the lock was last held by a thread on a different socket, then p must read the lock state from main memory, which is significantly more expensive.)

On the other hand, in each graph in the W2 workloads, the performance of each HyTM (and TL2) is similar to its performance in the corresponding graph in the W1 workloads. For Algorithm 1 (and TL2), this is because it is progressive. Although Algorithm 2 is not truly progressive, fast-path transactions will abort only if they are concurrent with the commit procedure of a slow-path transaction. In *RangeIncrement* operations, there is a long read-only prefix (which is exceptionally long because of Algorithm 2's quadratic validation) followed by a relatively small set of writes. Thus, *RangeIncrement* operations have relatively little impact on the fast-path. The explanation is similar for Hybrid NOrec (except that it performs less validation than Algorithm 2).

Observe that the performance of Hybrid NOrec decreases slightly, relative to Algorithm 2, after 24 threads. Recall that, in Hybrid NOrec, the global sequence number is a single point of contention on the fast-path. (In Algorithm 2, the global lock is only modified by slow-path transactions, so fast-path transactions do not have a single point of contention.) We believe this is due to NUMA effects, similar to those described in [4]. Specifically, whenever a threads on the first socket performs a fast-path transaction that commits and modifies the global lock, it causes cache invalidations for all other threads. Threads on socket two must then load the lock state from main memory, which takes much longer than loading it from the shared L3 cache. This causes the transaction to run for a longer time, which lengthens its window of contention, making it more likely to abort. (Note that, in the 0% updates graph in the W2 workload, we still see this effect, because there is a thread performing *RangeIncrement* operations.)

6. Related work and discussion

The proof of Theorem 2 is based on the analogous proof for step complexity of STMs that are *disjoint-access parallel* [14]. Early HyTMs like the ones described in [7, 13] provided progressiveness, but subsequent HyTM proposals sacrificed progressiveness for lesser instrumentation overheads. Recent work has investigated fallback to *reduced* hardware transactions [17] in which an all-software slow-path is replaced by a mix of hardware and software transactions. Our implementation of Hybrid NOrec follows [19], which additionally proposed the use of non-speculative accesses in fast-path transactions to reduce instrumentation overhead.

In ongoing work, we are implementing our algorithms on the IBM POWER8 HTM implementation which supports non-cached accesses in hardware transactions. We hope to understand whether the instrumentation overheads we observed on Intel's HTM are also inherent to POWER8's HTM implementation. To our knowledge, ours is the first work to consider the theoretical foundations of the cost of concurrency in HyTMs. In order to achieve high performance in practice, one must either identify a new progress condition to replace progressiveness or develop a new HyTM algorithm that effectively uses non-speculative writes. Both directions are promising, and little work has been done in either in the context of today's HTMs.

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

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

[4] T. Brown, A. Kogan, Y. Lev, and V. Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 121–132, 2016.

[5] 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.

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

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

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

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

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

[11] 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.

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

[13] 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.

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

[15] H. Q. Le, G. L. Guthrie, D. Williams, M. M. Michael, B. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1), 2015.

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

[17] 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.

[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.

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

[21] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel&reg; transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.