Cost of Concurrency in Hybrid Transactional Memory

Trevor Brown*1 and Srivatsan Ravi2

1 Technion, Israel Institute of Technology, Haifa, Israel
me@tbrown.pro
2 University of Southern California, Los Angeles, California, USA
srivatsr@usc.edu

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 (HyTMs) 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 trade-offs for these algorithms. Our experiments with Intel and IBM POWER8 HTMs seem to suggest that (i) the cost of concurrency also exists in practice, (ii) it is important to implement HyTMs that provide progressiveness for a maximal set of transactions without incurring high hardware instrumentation overhead or using global contending bottlenecks and (iii) there is no easy way to derive more efficient HyTMs by taking advantage of non-speculative accesses within hardware.

1998 ACM Subject Classification D.1.3 Concurrent Programming – parallel programming

Keywords and phrases Transactional memory, Lower bounds, Opacity

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.9

1 Introduction

The Transactional Memory (TM) abstraction is a synchronization mechanism that allows the programmer to optimistically execute sequences of shared-memory operations as atomic transactions. Several software TM designs [8, 24, 13, 11] have been introduced subsequent to the original TM proposal based in hardware [14]. The original dynamic STM implementation DSTM [13] ensures that a transaction aborts only if there is a read-write data conflict with a concurrent transaction (à la progressiveness [12]). However, to satisfy opacity [12], read operations in DSTM must incrementally validate the responses of all previous read operations to avoid inconsistent executions. This results in quadratic (in the size of the transaction’s read set) step-complexity for transactions. Subsequent STM implementations like NOrec [8]

* Trevor Brown received funding for this work from the Natural Sciences and Engineering Research Council of Canada.
and TL2 [10] 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 been 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 which do not modify 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, 17], 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, hardware transactions must be instrumented to perform additional metadata accesses, which introduces overhead. Hardware transactions typically provide automatic conflict detection at cacheline granularity, thus ensuring that a transaction will be aborted if it experiences memory contention on a cacheline. This is at least the case with Intel’s Transactional Synchronization Extensions [25]. The IBM POWER8 architecture 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 [3], it was shown that hardware transactions in opaque progressive HyTMs must perform at least one metadata access per transactional read and write. In this paper, we show that in opaque 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 in hardware transactions, there is a quadratic step complexity cost in 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.

Extensive experiments were performed to characterize the cost of concurrency in practice. We studied the instrumentation-optimal algorithms, as well as TL2, Transactional Lock Elision (TLE) [22] and Hybrid NOrec [23] on both Intel and IBM POWER architectures. Each of the algorithms we studied contributes to an improved understanding of the concurrency vs. hardware instrumentation vs. software validation trade-offs for HyTMs. Comparing results between the very different Intel and IBM POWER architectures also led to new insights. Collectively, our results suggest the following.

(i) The cost of concurrency is significant in practice; high hardware instrumentation impacts performance negatively on Intel and much more so on POWER8 due to its limited transactional cache capacity.
(ii) It is important to implement HyTMs that provide progressiveness for a maximal set of transactions without incurring high hardware instrumentation overhead or using global contending bottlenecks.

(iii) There is no easy way to derive more efficient HyTMs by taking advantage of non-speculative accesses supported within the fast-path in POWER8 processors.

2 Hybrid transactional memory (HyTM)

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 \( \mathcal{M} \).

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 \( E \). Let \( \text{txns}(E) \) 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 \( T_k \in \mathcal{T} \), \( E|k = E'|k \). A transaction \( T_k \in \text{txns}(E) \) is complete in \( E \) if \( E|k \) ends with a response event. The execution \( E \) is complete if all transactions in \( \text{txns}(E) \) are complete in \( E \). A transaction \( T_k \in \text{txns}(E) \) is t-complete if \( E|k \) ends with \( A_k \) or \( C_k \); otherwise, \( T_k \) is t-incomplete. We consider the dynamic programming model: the read set (resp., the write set) of a transaction \( T_k \) in an execution \( E \), denoted \( Rset_E(T_k) \) (resp., \( Wset_E(T_k) \)), is the set of t-objects that \( T_k \) attempts to read (resp., write) by issuing a t-read (resp., t-write) invocation in \( E \) (for brevity, we sometimes omit the subscript \( E \)).

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 model, shared memory configurations may be modified by accessing base objects via two kinds of primitives: direct and cached.

(i) In a direct (also called non-speculative) 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 \( \tau_i \).
More precisely, \( \tau_i \) is a set of triples \((b, v, m)\) where \( b \) is a base object identifier, \( v \) is a value, and \( m \in \{ \text{shared, exclusive} \} \) is an access mode. The triple \((b, v, m)\) is added to the tracking set when \( i \) performs a cached \text{rmw} access of \( b \), where \( m \) is set to \text{exclusive} if the access is nontrivial, and to \text{shared} otherwise. We assume that there exists some constant \( TS \) 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 \( \tau_i \) 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 \((b, v, \text{exclusive}) \in \tau_i \) (resp., \((b, v, \text{shared}) \in \tau_i \)), a process \( j \neq i \) applies any primitive (resp., any nontrivial primitive) to \( b \), then \( \tau_i \) becomes invalid and any subsequent event invoked by \( i \) sets \( \tau_i \) to \( \emptyset \) and returns \( \bot \). We refer to this event as a tracking set abort.

Any transaction executed by a correct process that performs at least one cached access must necessarily perform a \text{cache-commit} primitive that determines the terminal response of the transaction. A cache-commit primitive issued by process \( i \) with a valid \( \tau_i \) does the following: for each base object \( b \) such that \((b, v, \text{exclusive}) \in \tau_i \), the value of \( b \) in \( C \) is updated to \( v \). Finally, \( \tau_i \) is set to \( \emptyset \) and the operation returns commit. We assume that a fast-path transaction \( T_k \) returns \( A_k \) as soon as a cached primitive or \text{cache-commit} returns \( \bot \).

**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 \text{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 \( T_k \in \text{txns}(E) \) is a fast-path transaction if \( E|k \) contains at least one cached event. An event of a hardware transaction is either an invocation or response of a t-operation, or a direct trivial access or a cached access, or a cache-commit primitive.

> **Remark (Tracking set aborts).** Let \( T_k \in \text{txns}(E) \) be any t-incomplete fast-path transaction executed by process \( i \), where \((b, v, \text{exclusive}) \in \tau_i \) (resp., \((b, v, \text{shared}) \in \tau_i \)) after execution \( E \), and \( e \) be any event (resp., nontrivial event) that some process \( j \neq i \) is poised to apply after \( E \). The next event of \( T_k \) in any extension of \( E \cdot e \) is \( A_k \).

> **Remark (Capacity aborts).** Any cached access performed by a process \( i \) executing a fast-path transaction \( T_k \); \(|Dset(T_k)| > 1\) first checks the condition \(|\tau_i| = TS\), where \( TS \) is a pre-defined constant, and if so, it sets \( \tau_i = \emptyset \) and immediately returns \( \bot \).

**Direct reads within fast-path.** Note that we specifically allow hardware transactions to perform reads without adding the corresponding base object to the process’ tracking set, thus modeling the \text{suspend/resume} instructions supported by IBM POWER8 architectures. Note that Intel’s HTM does not support this feature: an event of a fast-path transaction does not include any direct access to base objects.

**HyTM properties.** We consider the TM-correctness property of opacity [12]: an execution \( E \) is opaque if there exists a legal (every t-read of a t-object returns the value of its latest committed t-write) 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: every t-operation returns a matching response within a finite number of its own steps if running step-contention free from a configuration in which every other transaction is
t-complete. Moreover, we focus on 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 $E$ [12].

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.

**Theorem 1.** 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 $\Omega(m^2)$; $m = |Rset(T_k)|$.

**Proof sketch.** We construct an execution of a read-only slow-path transaction $T_\phi$ 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.

For each $i \in \{1, \ldots, m\}$, $\mathcal{M}$ has an execution of the form depicted in Figure 1b. Start with the complete step contention-free execution of slow-path read-only transaction $T_\phi$ that performs $(i - 1)$ t-reads: $read_\phi(X_1) \cdots read_\phi(X_{i-1})$, followed by the t-complete step contention-free execution of a fast-path transaction $T_1$ that writes $nv_i \neq v_i$ to $X_i$ and commits and then the complete step contention-free execution fragment of $T_\phi$ that performs its $i^{th}$ t-read: $read_\phi(X_i) \rightarrow nv_i$. Indeed, by progressiveness, $T_i$ cannot incur tracking set aborts and since it accesses only a single t-object, it cannot incur capacity aborts. Moreover, in this execution, the t-read of $X_i$ by slow-path transaction $T_\phi$ must return the value $nv_i$ written by fast-path transaction $T_1$ since this execution is indistinguishable to $T_\phi$ from the execution in Figure 1a.

We now construct $(i - 1)$ different executions of the form depicted in Figure 1c: for each $\ell \leq (i - 1)$, a fast-path transaction $T_\ell$ (preceding $T_i$ in real-time ordering, but invoked following the $(i - 1)$ t-reads by $T_\phi$) writes $nv_\ell \neq v$ to $X_\ell$ and commits, followed by the t-read of $X_i$ by $T_\phi$. Observe that, $T_i$ and $T_\ell$ which access mutually disjoint data sets cannot contend on each other since if they did, they would concurrently contend on some base object and incur a tracking set abort, thus violating progressiveness. Indeed, by the TM-liveness property we assumed (cf. Section 2) and invisible reads for $T_\phi$, each of these $(i - 1)$ executions exist.

In each of these $(i - 1)$ executions, the final t-read of $X_i$ cannot return the new value $nv$: the only possible serialization for transactions is $T_\ell, T_i, T_\phi$; but the $read_\phi(X_\ell)$ performed by $T_\ell$ that returns the initial value $v$ is not legal in this serialization—contradiction to the assumption of opacity. In other words, slow-path transaction $T_\phi$ is forced to verify the validity of t-objects in $Rset(T_\phi)$. Finally, we note that, for all $\ell, \ell' \leq (i - 1); \ell' \neq \ell$, fast-path transactions $T_\ell$ and $T_\ell$ access mutually disjoint sets of base objects thus forcing the t-read of $X_i$ to access least $i - 1$ different base objects in the worst case. Consequently, for all
Table 1 Table summarizing complexities of HyTM implementations.

<table>
<thead>
<tr>
<th>Instrumentation in fast-path reads</th>
<th>Algorithm 1</th>
<th>Algorithm 2</th>
<th>TLE</th>
<th>HybridNOrec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrumentation in fast-path writes</td>
<td>per-read</td>
<td>constant</td>
<td>constant</td>
<td>constant</td>
</tr>
<tr>
<td>Instrumentation in fast-path writes</td>
<td>per-write</td>
<td>per-write</td>
<td>constant</td>
<td>constant</td>
</tr>
<tr>
<td>Validation in slow-path reads</td>
<td>(\Theta(</td>
<td>Rset</td>
<td>))</td>
<td>(O(</td>
</tr>
<tr>
<td>h/w/s/f concurrency</td>
<td>prog.</td>
<td>prog. for slow-path readers</td>
<td>zero</td>
<td>not prog., but small contention window</td>
</tr>
<tr>
<td>Direct accesses inside fast-path opacity</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

\[ i \in \{2, \ldots, m\}, \text{ slow-path transaction } T_\phi \text{ must perform at least } i - 1 \text{ steps while executing the } i^{th} \text{ t-read in such an execution.} \]

3.1 How STM implementations mitigate the quadratic lower bound step complexity

NOrec [8] 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 \(\text{iff} \) the global version has changed. TL2 [10] improves over NOrec by circumventing the lower bound of Theorem 1. 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.

3.2 Implications for disjoint-access parallelism in HyTM

The property of disjoint-access parallelism (DAP), in its weakest form, ensures that two transactions concurrently contend on the same base object only if their data sets are connected in the conflict graph, capturing data-set overlaps among all concurrent transactions [4]. It is well known that weak DAP STMs with invisible reads must perform incremental validation even if the required TM-progress condition requires transactions to commit only in the absence of any concurrent transaction [12, 16]. For example, DSTM [13] is a weak DAP STM that is progressive and consequently incurs the validation complexity. On the other hand, TL2 and NOrec are not weak DAP since they employ a global versioned lock that mitigates the cost of incremental validation, but this allows two transactions accessing disjoint data sets to concurrently contend on the same memory location. Indeed, this inspires the proof of Theorem 1.

4 Hybrid transactional memory algorithms

4.1 Instrumentation-optimal progressive HyTM

We describe a HyTM algorithm that is a tight bound for Theorem 1 and the instrumentation cost on the fast-path transactions established in [3]. Pseudocode appears in Algorithm 1. For each t-object \(X_j\), our implementation maintains a base object \(v_j\) that stores \(X_j\)’s value and a sequence lock \(r_j\).
(a) Slow-path transaction $T_\phi$ performs $i - 1$ distinct t-reads (each returning the initial value) followed by the t-read of $X_i$ that returns value $nv$ written 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 $T_\phi$ and must be committed in this execution since it cannot incur a tracking set or capacity abort. The t-read of $X_i$ must return $nv$ because this execution is indistinguishable to $T_\phi$ from 1a.

(c) In each of these each $i - 1$ executions, fast-path transactions cannot incur a tracking set or capacity abort. By opacity, the t-read of $X_i$ by $T_\phi$ cannot return new value $nv$. Therefore, to distinguish the $i - 1$ different executions, t-read of $X_i$ by slow-path transaction $T_\phi$ is forced to access $i - 1$ different base objects.

**Figure 1** Proof steps for Theorem 1.

**Fast-path transactions:** For a fast-path transaction $T_k$ executed by process $p_i$, the $read_k(X_j)$ implementation first reads $r_j$ (direct) and returns $A_k$ if some other process $p_j$ holds a lock on $X_j$. Otherwise, it returns the value of $X_j$. 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 $X_j$; otherwise process $p_i$ increments the sequence lock $r_j$. 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 t-object from $v_j$, adds $r_j$ to $Rset(T_k)$ if its not held by a concurrent transaction.
transaction 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 entries 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. If all the locks on the write set were acquired successfully, $T_k$ performs validation of the read set and if successful, updates the values of the t-objects in shared memory, releases the locks and returns $C_k$; else $p_i$ aborts the transaction.

**Direct accesses inside fast-path:** Note that opacity is not violated even if the sequence lock accesses during t-read may be performed directly without incurring tracking set aborts.

### 4.2 Instrumentation-optimal HyTM that is progressive only for slow-path reading transactions

Algorithm 2 does not incur the linear instrumentation cost on the fast-path reading transactions (inherent to Algorithm 1), but provides progressiveness only for slow-path reading transactions. The instrumentation cost on fast-path t-reads is avoided by using a global lock that serializes all updating slow-path transactions during the $tryC_k$ procedure. Fast-path transactions simply check if this lock is held without acquiring it (similar to TLE [22]). While per-read instrumentation is avoided, Algorithm 2 still has per-write instrumentation.

### 4.3 Sacrificing progressiveness and minimizing contention window

Observe that the lower bound in Theorem 1 assumes progressiveness for both slow-path and fast-path transactions along with opacity and invisible reads. Note that Algorithm 2 retains the validation step complexity cost since it provides progressiveness for slow-path readers.

Hybrid NOrec [7] 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 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 (the second one called $esl$). 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). Although the window for fast path transactions to contend on $gsl$ is small, our experiments have shown that contention on $gsl$ has a significant impact on performance.
Algorithm 1 Progressive fast-path and slow-path opaque HyTM implementation; code for transaction $T_k$

1. Shared objects
2. $v_j$, value of each t-object $X_j$
3. $r_j$, a sequence lock of each t-object $X_j$

4. Code for fast-path transactions

7. $\text{read}_k(X_j)$
8. $\text{ov}_j := v_j$
9. $\text{or}_j := r_j$
10. $\text{if} \ or_j.\text{isLocked}() \text{then return } A_k$
11. $\text{return } \text{ov}_j$

13. $\text{write}_k(X_j,v)$
14. $\text{or}_j := r_j$
15. $\text{if} \ or_j.\text{isLocked}() \text{then return } A_k$
16. $r_j := or_j.\text{IncSequence}()$
17. $v_j := v$
18. $\text{return } \text{OK}$

20. $\text{tryC}_k()$
21. $\text{commit-cache}_i$

23. Function: $\text{release}(Q)$
24. $\text{for each } X_j \in Q \text{ do } r_j := \text{or}_j.\text{unlock}()$

26. Function: $\text{acquire}(Q)$
27. $\text{for each } X_j \in Q$
28. $\text{if } r_j.\text{tryLock}() \text{ then CAS/LLSC}$
29. $\text{Lset}(T_k) := Lset(T_k) \cup \{X_j\}$
30. $\text{else}$
31. $\text{release}(\text{Lset}(T_k))$
32. $\text{return } \text{false}$
33. $\text{return } \text{true}$
34. 35. Code for slow-path transactions

37. $\text{Read}_k(X_j)$
38. $\text{if } X_j \in \text{Wset}(T_k) \text{ then return } \text{Wset}(T_k)\.\text{locate}(X_j)$
39. $\text{or}_j := r_j$
40. $\text{ov}_j := v_j$
41. $\text{Rset}(T_k) := \text{Rset}(T_k) \cup \{X_j, \text{or}_j\}$
42. $\text{if} \ or_j.\text{isLocked}() \text{then return } A_k$
43. $\text{if not validate()} \text{ then return } A_k$
44. $\text{return } \text{ov}_j$

46. $\text{write}_k(X_j,v)$
47. $\text{or}_j := r_j$
48. $\text{ov}_j := v$
49. $\text{if} \ or_j.\text{isLocked}() \text{then return } A_k$
50. $\text{Wset}(T_k) := \text{Wset}(T_k) \cup \{X_j, \text{ov}_j, \text{or}_j\}$
51. $\text{return } \text{OK}$

53. $\text{tryC}_k()$
54. $\text{if } \text{Wset}(T_k) = \emptyset \text{ then return } C_k$
55. $\text{if not acquire}(\text{Wset}(T_k)) \text{ then return } A_k$
56. $\text{if not validate()} \text{ then return } A_k$
57. $\text{release}(\text{Wset}(T_k))$
58. $\text{return } A_k$
59. $\text{for each } X_j \in \text{Wset}(T_k) \text{ do } v_j := \text{nv}_j$
60. $\text{release}(\text{Wset}(T_k))$
61. $\text{return } C_k$

63. Function: $\text{validate}()$
64. $\text{if } \exists X_j \in \text{Rset}(T_k): \text{or}_j.\text{getSequence}() \neq r_j.\text{getSequence}() \text{ then return } \text{false}$
65. $\text{return } \text{true}$
Algorithm 2 Opaque HyTM implementation that is progressive only for slow-path reading transactions; code for $T_k$ by process $p_i$

```
1 Shared objects
2 $L$, global lock
3
4 Code for fast-path transactions
5 $\text{start}_k()$
6 if $L$.isLocked() then return $A_k$
7
8 $\text{read}_k(X_j)$
9 $ov_j := v_j$
10 return $ov_j$
11
12 $\text{write}_k(X_j, v)$
13 $or_j := r_j$
14 $r_j := or_j.IncSequence()$
15 $v_j := v$
16 return OK

18 $\text{try}_C_k()$
19 return commit - cache

```

```
23 Code for slow-path transactions
24 $\text{try}_C_k()$
25 if $Wset(T_k) = \emptyset$ then return $C_k$
26
27 $L$.Lock()$
28$\text{if not acquire}(Wset(T_k))$ then return $A_k$
29
30$\text{if not validate}()$ then
31 return $A_k$
32
33 for each $X_j \in Wset(T_k)$ do $v_j := nv_j$
34
35 return $C_k$

36 Function: $\text{release}(Q)$
37 for each $X_j \in Q$ do $r_j := nr_j.unlock()$
38 $L$.unlock(); return OK
```

5 Evaluation

We now 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 HyTM algorithm design affects performance with Intel and IBM POWER8 HTMs, and

(G3) to determine whether direct accesses can be used to obtain performance improvements on IBM POWER8 using suspend/resume instructions to escape from a hardware transaction.

5.1 Experimental system

The experimental Intel system is a 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 (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. 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`.

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 experimental POWER8 system is a IBM S822L with 2x 12-core 3.02GHz processor cards, 128GB of RAM, running Ubuntu 16.04 LTS. All code was compiled using G++ 5.3.1. This is a dual socket machine, and each socket has two NUMA zones. It is expensive to access memory on a different NUMA zone, and even more expensive if the NUMA zone is on a different socket. POWER8 uses the L2 cache for detecting tracking set aborts, and limits
the size of a transaction’s read- and write-set to 8KB each \[20\]. This is in contrast to Intel
which tracks conflicts on the entire L3 cache, and limits a transaction’s read-set to the L3
cache size, and its write-set to the L1 cache size.

We pin one thread on each core within a NUMA zone before moving to the next zone. We
remark that unlike the thread pinning policy for Intel which saturated the first socket before
moving to the next, this proved to be the best policy for POWER8 which experiences severe
negative scaling when threads are saturated on a single 8-way hardware multi-threaded core.
This is because all threads on a core share resources, including the L1 and L2 cache, a single
branch execution pipeline, and only two load-store pipelines.

5.2 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. We implemented
Algorithm 1 on POWER8 where each read of a sequence lock during a transactional read
operation was enclosed within a pair of suspend/resume instructions to access them without
incurring tracking set aborts (Algorithm 1\(^*\)). We remark that this does not affect the
opacity of the implementation. We also implemented the variant of Hybrid NOrec (Hybrid
NOrec\(^*\)) in which the update to gsl is performed using a fetch-increment primitive between
suspend/resume instructions, as is recommended in \[23\].

In each algorithm, instead of placing a lock next to each address in memory, we allocated
a global array of one million locks, and used a simple hash function to map each address to
one of these locks. This avoids the problem of having to change a program’s memory layout
to incorporate locks, and greatly reduces the amount of memory needed to store locks, at
the cost of some possible false conflicts since many addresses map to each lock. Note that
the exact same approach was taken by the authors of TL2.

We chose not to compile the hybrid TMs as separate libraries, since invoking library
functions for each read and write can cause algorithms to incur enormous overhead. Instead,
we compiled each hybrid TM directly into the code that uses it.

5.3 Experimental 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 log-
arithmetic 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 slow-path, we introduced an operation called a RangeIncrement
that often fails in hardware and must run on the slow-path. A $\text{RangeIncrement}(\text{low,hi})$ atomically increments the values associated with each key in the range $[\text{low,hi}]$ present in the tree. Note that a $\text{RangeIncrement}$ 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 $\text{Insert}$, $\text{Delete}$ and $\text{Search}$, and (W2) $n - 1$ threads perform $\text{Insert}$, $\text{Delete}$ and $\text{Search}$ and one thread performs only $\text{RangeIncrement}$ operations. Figure 2 shows the results for both types of workloads.

5.4 Results

We first discuss the results for the Intel machine. 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 similar amounts of instrumentation: 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 $\text{RangeIncrement}$ 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 W1 workload graph. For Algorithm 1 (and TL2), this is because of progressiveness. 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 \textit{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, \textit{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 [5]. 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 lengthens the transaction’s window of contention, making it more likely to abort. (In the 0% updates graph in the W2 workload, we still see this effect, because there is a thread performing \textit{RangeIncrement} operations.)

We now discuss the results for the IBM POWER8 machine. Algorithm 1 performs poorly on POWER8: POWER8 transactions can only load 64 cache lines before they will abort [21]. Transactions read locks and tree nodes, which are in different cache lines: together, they often exceed 64 cache lines loaded in a tree operation, so most transactions cannot succeed in hardware. Consequently, on POWER8, it is incredibly important either to have minimal instrumentation in transactions, or for metadata to be located in the same cache lines as program data. Of course, the latter is not possible for HYTMs, which do not have control over the layout of program data. Consequently, Algorithm 2 outperforms Algorithm 1 in POWER8 quite easily by avoiding the per-read instrumentation.

Algorithm 1 is improved slightly by the expensive (on POWER8) suspend/resume on sequence locks during transactional reads, but it still performs relatively poorly. To make suspend/resume a practical tool, one could imagine attempting to collect several metadata accesses and perform them together to amortize the cost of a suspend/resume pair. For instance, in Algorithm 1, one might try to update the locks for all of the transactional writes at once, when the transaction commits. Typically one would accomplish this by logging all writes so that a process can remember which addresses it must lock at commit time. However, logging the writes inside the transaction would be at least as costly as just performing them.

Observe that Hybrid NOrec does far worse with updates in POWER8 than on the Intel machine. This is due to the fact that fetch-increment on a single location experiences severe negative scaling on the POWER8 processor: e.g., in one second, a single thread can perform 37 fetch-add operations while 6 threads perform a total of 9 million and 24 threads perform only 4 million fetch-add operations. In contrast, the Intel machine performs 32 million operations with 6 threads and 45 million with 24 threads. This is likely because this Intel processor provides fetch-add instructions while it must be emulated on POWER8.
In Hybrid NOrec\*, the non-speculative increment of gsl actually makes performance worse. Recall that in Hybrid NOrec, if a fast-path transaction $T_1$ increments gsl, and then a software transaction $T_2$ reads gsl (as part of validation) before $T_1$ commits, then $T_1$ will abort, and $T_2$ will not see $T_1$’s change to gsl. So, $T_2$ will have a higher chance of avoiding incremental validation (and, hence, will likely take less time to run, and have a smaller contention window). However, in Hybrid NOrec\*, once $T_1$ increments gsl, $T_2$ will see the change to gsl, regardless of whether $T_1$ commits or aborts. Thus, $T_2$ will be forced to perform incremental validation. In our experiments, we observed that a much larger number of transactions ran on the fallback path in Hybrid NOrec\* than in Hybrid NOrec (often several orders of magnitude more).

6 Related work and discussion

HyTM implementations and complexity. Early HyTMs like the ones described in [9, 15] provided progressiveness, but subsequent HyTM proposals like PhTM [18] and Hybrid-NOrec [7] sacrificed progressiveness for lesser instrumentation overheads. However, the clear trade-off in terms of concurrency vs. instrumentation for these HyTMs have not been studied in the context of currently available HTM architectures. This instrumentation cost on the fast-path was precisely characterized in [3]. In this paper, we proved the inherent cost of concurrency on the slow-path thus establishing a surprising, but intuitive complexity separation between progressive STMs and HyTMs. Moreover, to the best of our knowledge, this is the first work to consider the theoretical foundations of the cost of concurrency in HyTMs in theory and practice (on currently available HTM architectures). Proof of Theorem 1 is based on the analogous proof for step complexity of STMs that are disjoint-access parallel [16, 12].

Our implementation of Hybrid NOrec follows [23], which additionally proposed the use of direct accesses in fast-path transactions to reduce instrumentation overhead in the AMD Advanced Synchronization Facility (ASF) architecture.

Beyond the two path HyTM approach. Employing an uninstrumented fast fast-path. We now 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 transform an opaque HyTM $\mathcal{M}$ to an opaque HyTM $\mathcal{M'}$ using a shared fetch-and-add metadata base object $F$ that slow-path updating transactions increment (and resp. decrement) at the start (and resp. end). In $\mathcal{M'}$, a “fast” fast-path transaction checks first if $F$ is 0 and if not, aborts the transaction; otherwise the transaction is continued as an uninstrumented hardware transaction. The code for the fast-path and the slow-path is identical to $\mathcal{M}$.

Recent work has investigated fallback to reduced hardware transactions [19] in which an all-software slow-path is augmented using a slightly faster slow-path that is optimistically used to avoid running some transactions on the true software-only slow-path. Amalgamated lock elision (ALE) was proposed in [2] which improves over TLE by executing the slow-path as a series of segments, each of which is a dynamic length hardware transaction. Invyswell [6] is a HyTM design with multiple hardware and software modes of execution that gives flexibility to avoid instrumentation overhead in uncontended executions. We remark that such multi-path approaches may be easily applied to each of the Algorithms proposed in this paper. However, in the search for an efficient HyTM, it is important to strike the fine balance between concurrency, hardware instrumentation and software validation cost. Our lower bound, experimental methodology and evaluation of HyTMs provides the first clear characterization of these trade-offs in both Intel and POWER8 architectures.
Acknowledgements. Computations were performed on the SOSCIP Consortium’s Agile computing platform. SOSCIP is funded by the Federal Economic Development Agency of Southern Ontario, the Province of Ontario, IBM Canada Ltd., Ontario Centres of Excellence, Mitacs and 15 Ontario academic member institutions. This work was performed while Trevor Brown was a student at the University of Toronto.

References


9:16 Cost of Concurrency in Hybrid Transactional Memory


