Continuous Checkpointing of HTM Transactions in NVM

Ellis Giles  
Rice University, USA  
erg@rice.edu

Kshitij Doshi  
Intel Corporation, USA  
kshitij.a.doshi@intel.com

Peter Varman  
Rice University, USA  
pjv@rice.edu

Abstract

This paper addresses the challenges of coupling byte addressable non-volatile memory (NVM) and hardware transaction memory (HTM) in high-performance transaction processing. We first show that HTM transactions can be ordered using existing processor instructions without any hardware changes. In contrast, existing solutions posit changes to HTM mechanisms in the form of special instructions or modified functionality. We exploit the ordering mechanism to design a novel persistence method that decouples HTM concurrency from back-end NVM operations. Failure atomicity is achieved using redo logging coupled with aliasing to guard against mistimed cache evictions. Our algorithm uses efficient lock-free mechanisms with bounded static memory requirements. We evaluate our approach using both micro-benchmarks, and, benchmarks in the STAMP suite, and showed that it compares well with standard (volatile) HTM transactions. We also showed that it yields significant gains in throughput and latency in comparison with persistent transactional locking.

CCS Concepts  • Information systems → Storage class memory; • Computing methodologies → Concurrent algorithms

Keywords  Storage Class Memory; Checkpointing; Persistence; Lock-Free; Hardware Transactional Memory; HTM; TSX

1. Introduction

Recent years have witnessed a sharp shift towards real time data-driven and high-throughput applications. This has spurred a broad adoption of in-memory and massively parallelized data processing [9, 22, 26] across business, scientific, and industrial application domains [18, 28]. Two emerging hardware developments provide further impetus to this approach and raise the potential for transformative gains in speed and scalability: (a) the arrival of byte-addressable and large non-volatile memories (NVM), such as Intel’s 3D XPoint™ technology, simplifies management of durability, and (b) the availability of CPU-based transaction support (with Hardware Transactional Memory or HTM) [13, 25] makes it straightforward for threads to work spontaneously in shared memory spaces without having to synchronize explicitly. This paper describes a novel technique for all-or-nothing updates to NVM locations in the course of HTM transactions, so that the use of HTM can be extended seamlessly from volatile to durable memories.

To achieve consistency of persistent states, a sequence of store operations to NVM in a transactional section of code must be durable and atomic (all or none). Software approaches for ensuring atomic durable updates share some characteristics with HTM techniques in commercial processors – both checkpoint state at some level of granularity, and both guard against communication of partial updates. However, different mechanisms are at play: while stable stores to persistent media are usually obtained by covering updates with replication, logging, or versioning, partial updates are prevented from being propagated between threads in HTM transactions by CPUs sheltering them until the transaction closes. Once an HTM transaction closes, its updates become visible en masse through the cache hierarchy and can travel in any order to memory DIMMs. However stable storage of the updates into NVM by the transaction further requires an ability to reliably delineate its updates from those by other overlapping transactions, and to use that delineation to recover from an unanticipated machine restart. The technique described in this paper achieves this delineation without requiring hardware changes, and achieves lightweight, fine-grained checkpointing for durable state in NVM without the need for frequent (and expensive) snapshots of a machine’s memory state.

Current solutions for transaction durability and atomicity in NVM vary in the degree of processor support they require: (a) pure software approaches [4, 5, 11, 17, 30] craft solutions using software algorithms and data structures, with a common characteristic that a log (either a write-ahead log or an undo log) is created and made durable on NVM whilst changes are made to the volatile cache hierarchy. The need to persist log records into NVM makes these approaches problematic for existing HTMs (e.g., Intel TSX) which cannot bypass the caches in order to flush the log records synchronously into NVM ahead of transaction closings; (b) others propose significant changes to existing cache hardware and protocols [16, 31] or use external hardware controllers without disrupting the processor core [8, 10, 24], and work with traditional methods of concurrency control – using either two-phase locking protocols to isolate transactions, or handle concurrency within the rubric of an STM [30]. Recent work [1, 2, 17] aims to exploit processor-supported HTM mechanisms extended in various ways to perform write-through updates of metadata before, or coincident with, closing an HTM envelope. While exploring such hardware support is an interesting research path, it is also true that selective and piecemeal changes to the clean isolation semantics of HTM cannot be undertaken lightly. Understanding the full impact of these changes on system correctness and performance can require long gestation periods before processor manufacturers will embrace the proposals.

The solution proposed in this paper does not require any changes to existing Intel HTM instructions or semantics and uses only the announced Intel instructions for NVM. It aims to achieve the concurrency benefits of HTM by allowing transactions to continue to operate at the speed of volatile memory transactions, while using backend operations to create a consistent persistent log for recovery to a consistent state in the event of failure. We introduce a
persistence lag parameter $\ell$ that controls the lag between transaction commit and transaction persistence in NVM. When $\ell = 0$ we have strict persistence consistency, wherein a transaction commits only when all its updates have been persisted. Larger values of $\ell$ allow transaction executions to run ahead of persistence, similar to a checkpointing mechanism. However, we do not checkpoint at fixed intervals as is traditional; instead our persistence mechanisms provide a continuous and consistent waveform of persistence, and applications can decide how much ahead of the waveform a transaction can be when it commits. We therefore call our scheme cc-HTM for continuous checkpointing. Using cc-HTM, a transaction may choose between full transactional persistence (i.e., ACID) or opt for looser coupling between concurrency and durability.

In this paper we make the following contributions: (a) show that HTM transactions can be ordered using existing instructions without any hardware changes; (b) create a method for persistent NVM transactions that leverages HTM for transaction isolation and uses aliasing and redo-logging for durability and atomicity; (c) develop an efficient, lock-free implementation using bounded, statically allocated memory space; and (d) empirically validate our solution on existing Intel hardware. We evaluated our cc-HTM solution using both micro-benchmarks and benchmarks from the Stanford Transactional Applications for Multi-Processing [20], or STAMP, benchmark suite, and compared it to volatile DRAM based HTM concurrency and a persistent transactional locking implementation (by extending the technique of [7] for persistence, as described in section 5). We show that cc-HTM approach compares well with standard (volatile) HTM transactions, and that it yields significant gains in throughput and latency in comparison with persistent transactional locking.

Section 2 discusses the problem and checkpointing model, followed by a high-level description of our solution in Section 3. Implementation is discussed in Section 4, followed by evaluation results in Section 5. Previous work is summarized in Section 6.

2. Overview

A transaction refers to a program unit that is executed atomically without interference from other transactions that may access the same data. Two transactions are said to conflict if they both access a common variable and at least one of the accesses is a write. An HTM transaction is demarcated by the instructions \texttt{XBegin} and \texttt{XEnd}. The execution of concurrent conflicting HTM transactions is serialized by the HTM. The HTM hardware ensures that transactions appear to execute atomically (all-or-nothing execution) and are isolated from each other (some serial execution order).

Consider two conflicting HTM transactions $T_1$: $\{x = 1, y = z\}$ and $T_2$: $\{y = 10, z = x\}$. HTM guarantees that the execution will preserve atomicity and strict isolation. Suppose initially $x = y = z = 0$. Possible visible states of $(x, y, z)$ when $T_1$ and $T_2$ execute concurrently are: $S_0 = (0, 0, 0)$ (no transaction completed), $S_1 = (1, 0, 0)$ ($T_1$ only), $S_2 = (0, 10, 0)$ ($T_2$ only), $S_3 = (1, 10, 1)$ ($T_1$ followed by $T_2$), $S_4 = (1, 0, 1)$ ($T_2$ followed by $T_1$).

With persistent memory there is an additional execution constraint, namely that of durability. Following recovery after a system crash, the values of the variables must reflect one of the the atomic states $S_i$ arising from the actual execution. However this is complicated due to two constraints imposed by the hardware: (i) within the HTM all writes are restricted to protected cached copies and cannot update memory; for instance, using a cache bypass instruction to directly stream an update to memory or using a CLWB (cache line writeback) instruction to flush a cache line to memory, will cause the transaction to abort [15]; (ii) outside the HTM all transaction variables are visible in the cache hierarchy, and are subject to normal cache eviction and write back mechanisms; if the machine crashes before all the transaction updates are reflected in NVM, the system will recover to an inconsistent persistent state.

A possible solution to the problem is to alter HTM semantics and hardware to allow writes to selectively escape the HTM protection domain and update memory from within the HTM. This is the approach followed in [1, 2], where a new instruction called Transparent Flush (TF) is proposed for this purpose. With TF, a transaction can write a log of its updates to NVM from within the transaction before its \texttt{XEnd}. In this case a machine crash can be tolerated since the HTM prevents dribbling of updates made to transaction variables from within the transaction, while once outside the transaction the updates are protected by the logged values from which they can be recovered. The drawback to the scheme is that it violates existing HTM semantics and requires significant changes to HTM hardware. In contrast cc-HTM uses only existing processor mechanisms and writes its logs outside the HTM transaction.

A second problem which arises is that of ordering transaction logs written concurrently to NVM, so that they can be replayed correctly following a failure. Proposed solutions include expanding the instructions of \texttt{XBegin} and \texttt{XEnd} to support the all or nothing semantics and requires significant changes to HTM hardware. Our solution uses a novel timestamp supported in current Intel processors to achieve this in a more lightweight manner and without any hardware changes.

Our solution to this problem is presented in detail in Section 3. It separates concurrency control from the persistence portions of the transaction. The concurrency portions execute in the foreground, serializing themselves and exchanging data as controlled by the HTM mechanism. The persistence operations occur in the background outside the HTM section, safely retiring the updated variables to NVM and ensuring that in the event of a failure the NVM can be restored to a consistent state. Abort (and retry) occur exactly as they would with an HTM-based algorithm in volatile memory, since an abort must leave NVM unchanged.

2.1 Checkpointing and Recovery Model

In this section we describe the checkpointing model and the role of the lag parameter $\ell$ in more detail. Informally, transactions are allowed to complete the concurrency portions of their transactions at the speed of the HTM. The transaction is held in a wait state for its updates to be reflected consistently in NVM, but its updates in volatile memory are available for use by other transactions. In the default policy, a transaction waits to commit until its updates are persistent. In practice, transactions may be allowed to complete early even before their updates are persisted, with the understanding that the updates may be lost in an inopportune machine crash.

The lag parameter $\ell$ reflects how many transactions can be allowed to complete prior to being persisted; i.e. how far ahead of the persistence wavefront we allow the computation wavefront to advance. $\ell = 0$ reflects the default conservative policy. Different applications may choose different thresholds of $\ell$ depending on the perceived benefit of early completion versus guaranteed persistence. Note that in the event of failure the system will always recover to a consistent past state, and system consistency is not affected by the early commit of some transactions.

2.1.1 Formal Checkpoint Model

The execution of concurrent conflicting transactions are serialized by the HTM. For any two conflicting transactions $T_1$ and $T_2$ one can define a relation $\prec$ such that $T_1 \prec T_2$ if $T_1$ precedes $T_2$ in the execution order defined by the HTM. Define a valid order to be a sequence of completed transactions $T_1$, $T_2$, $\cdots$, $T_n$ in which $T_i$ occurs earlier than $T_j$ in the sequence if $T_i \prec T_j$. A consistent
checkpoint is any prefix \( T_1, T_2, \ldots, T_k \), \( k \leq n \) of a valid order. On recovery from a failure our algorithm will restore the state of NVM to a consistent checkpoint by replaying the log of the retired transactions. A consistent checkpoint will reflect a prior, valid transaction state that may differ (in an incommittal way) from any state of volatile memory during transaction execution.

**Example:** Consider conflicting transactions \( T_1, T_2, T_3 \) (with all variables initially 0) performing the following transactional updates \( \{ x = 1, y = 10 \} \), \( \{ y = 11, z = 20 \} \) and \( \{ x = 2, z = 21 \} \) respectively, and suppose they are serialized by the HTM in the ordered sequence \( S_1 = (T_1, T_2, T_3) \). Similarly, assume conflicting transactions \( T_4, T_5 \) with transaction updates \( \{ a = 5, b = 6 \} \), \( \{ b = 7, c = 8 \} \) are serialized as \( S_2 = (T_4, T_5) \). Any interleaving of \( S_1 \) and \( S_2 \) that maintains individual sequence order is a valid order: for instance \((T_1, T_4, T_2, T_5, T_3)\). Note that the definition does not insist on an ordering between two non-conflicting transactions like \( T_2 \) and \( T_4 \) due to the inherent ambiguity in defining their relative order. That is the orderings of volatile memory and persistent memory for non-conflicting transactions may differ. However, this cannot happen for conflicting transactions.

Suppose that the actual execution completed in the order \((T_1, T_2, T_3)\), so that the state of volatile memory is \( x = 1, y = 11, z = 20, a = 5, b = 6 \) when the machine crashes. It is possible that the system recovers to a consistent checkpoint \((T_1, T_2)\), which will restore the state to \( x = 1, y = 10, z = 0, a = 5, b = 6 \), since the non-conflicting transactions \( T_3 \) and \( T_4 \) do not impose any correctness ordering.

### 3. Our Approach

Our approach persists an HTM transaction onto durable NVM consistently by splitting it into two parts: a parallel execution phase that completes under HTM provisions (where its updates are limited to the volatile cache hierarchy) and a decoupled, ordered-durability phase that follows. A volatile log constructed during the HTM execution is first persisted and then used in the ordered-durability phase to cover deferred updates of values transactionally.

The order in which the updates get committed to NVM home locations is established by using a fine-grained monotonic persistence-timestamp from within the HTM execution phase, obtained without the danger of causing inter-thread memory collisions. The persistence-timestamp is bracketed between two other timestamps that are obtained outside the HTM phase - a start-timestamp and an end-timestamp, and as described shortly, this bracketing provides the required temporal reasoning about a consistent order for committing potentially overlapping updates from HTM transactions that may be partially concurrent with one another. The timestamped logs are funneled into a persistence management thread for durability ordering, and the threads that issue them may either wait until updates are durably accepted into NVM in the right order, or proceed to perform other operations without waiting - giving rise to consistent (in NVM) but flexibly delayed and efficiently parallel transactions in the volatile cache hierarchy.

To prevent spontaneous cache evictions that may occur after XEnd from corrupting the NVM with partial transactional updates, all transaction variables are aliased to shadow volatile locations within the HTM so the updates do not directly percolate to NVM. This is done using an Alias Table that implements the shadow locations and is described in detail in Section 4.1.

The structure of a transaction is shown in Listing 1. The original HTM transaction is marked by the XBegin and XEnd instructions and embedded within a transaction wrapper marked by cc-HTM primitives AtomicBegin and AtomicEnd. A transaction has two phases during its execution: a concurrency phase in volatile memory and a persistence phase in NVM. TXQueue is the main data structure that tracks the progress of a transaction.

### Listing 1: Transaction Structure

```
Listing 1: Transaction Structure

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>myId = FetchAndIncrement(TxCounter)</td>
</tr>
<tr>
<td></td>
<td>TXQueue[myIndex].persistTS = myPersistTS</td>
</tr>
<tr>
<td>FINISH</td>
<td>XEnd</td>
</tr>
<tr>
<td></td>
<td>TXQueue[myIndex].logAddress = NVM Address of log</td>
</tr>
<tr>
<td></td>
<td>Place TXQueue[myIndex] into a priority queue ordered by their persistTS fields</td>
</tr>
<tr>
<td>WAIT</td>
<td>AtomicEnd</td>
</tr>
</tbody>
</table>
```

A transaction goes through four states in the course of its execution. The EXECUTE state is where it performs the specified HTM transaction. The WAIT state where it waits for its updates to be persisted to NVM before committing. A transaction maintains three timestamps in the course of its execution: `start timestamp` (startTS) and `end timestamp` (endTS) are values of a monotonically-increasing global transaction counter that assigns a unique id to each transaction; the `persistance timestamp` is obtained by reading the platform-wide timestamp counter provided in Intel machines [14].

Execution begins in the START state. The transaction obtains a unique transaction number myId that serves as the start timestamp of the transaction, which is recorded in the entry (myIndex) of TXQueue (steps 1-3). The transaction then enters the EXECUTE state with an XBegin instruction, which starts the execution of the HTM controlled transaction. To sidestep problems of uncontrolled cache evictions of transaction variables following XEnd, updates are made to shadow DRAM locations implemented as an Alias Table. Writes are also logged in a private log that will be written to NVM after XEnd. To create a consistent persist order we leverage the read platform clock instruction RDTSCP provided in Intel machines [14]. Just before executing XEnd, the transaction reads the platform-wide timestamp counter by executing RDTSCP. Note that reading the timestamp counter (step 4) can be done within an HTM transaction without causing an abort, and it can be made the last instruction to be completed before XEnd.

We do not make any assumptions about the order in which two transactions read the counter and complete their XEnd. It is entirely possible for a transaction to receive a smaller timestamp.
The log record for a transaction consists of its start timestamp, and the RDTSCP timer value that was previously read at its persistence timestamp (steps 5, 6). The transaction then flushes its log records to NVM using cache line writes together with an SFENCE as recommended by Intel (step 7). The transaction log is now persistent, and the transaction could apply its updates to the NVM home locations of the variables (by copying their updated values from the log or the shadow locations) without fear of a machine crash. We refer to the updating of home locations as transaction retirement. However there is an ordering constraint that must first be satisfied. Transaction retirements must occur in the order in which they completed\(^1\), which in our case is the order of persistence timestamps. (See Claims 1 and 2 below.) This is achieved by inserting the finished transactions in a priority queue ordered by their persistence timestamps (steps 8, 9).

A transaction can be retired when it is in the WAIT state and has the smallest persistence timestamp among all transactions in the system. It is not enough to have the smallest timestamp in the priority queue, because there may be (laggard) threads with earlier persistence timestamps that have not yet entered themselves into the priority queue. This condition can be ensured by checking that all transactions not in the priority queue began after the transaction at the head of the priority queue ended.\(^i.e\). by comparing the end timestamp (endTS) of the transaction at the head of the priority queue with the smallest start timestamp (startTS) of in-flight transactions whose endTS is not in the priority queue.

A separate persistence thread is responsible for retiring transactions when it is safe to do so, and signaling the waiting transaction. In Section 4, we describe an efficient concurrent lock-free data structure to implement the priority queue and safety checking operations, along with details of managing of the shadow variables with the Alias Table. We end the section by stating invariants maintained by the algorithm, which justify its correctness. The proofs of the claims below are provided in the Appendix.

Claim 1: If two conflicting transactions \(T_1\) and \(T_2\) satisfy \(T_1 \prec T_2\) (\(i.e\). \(T_1\) completes its XEnd before \(T_2\)) then the persistence timestamp of \(T_1\) must be less than that of \(T_2\).

Claim 2: If the transaction \(T\) at the head of the priority queue has an endTS that is smaller than the minimum startTS of all transactions not in the priority queue (PQ), then \(T\) has the smallest persistence timestamp of all transactions in the system.

3.1 Logging and Recovery

The log record for a transaction consists of its start timestamp (startTS), end timestamp (endTS), and persistence timestamp (persistTS), followed by its write set (address, size, value for each of its updates). At the start of a transaction its startTS is persisted to the log; at the end of the transaction, its endTS, persistTS, write set, and end marker are persisted to the log. Only after its log has been fully persisted may a transaction be committed. Once the write set in the log has been copied to the NVM home locations by the persistence thread (called log retirement), the log may be deleted.

To recover after a failure, completed logs that have not been retired are examined. All log records whose endTS is smaller than the startTS of the earliest non-completed log are selected, and retired one-by-one in increasing order of their persistTS values.

Effectively these are the transactions in the priority queue that satisfy the conditions for retirement, but have not yet been retired by the persistence thread before the machine crashes\(^2\).

3.2 Transactional Persistence Extensions

Some transactions might require the lag parameter \(L = 0\). This ensures that the transaction is retired before subsequent program instructions are executed. In like fashion, some applications might have needs similar to a memory fence, \(e.g\). an application may require a) that all transactions be completed before proceeding or b) all transactions currently open must close before proceeding but new transactions may open. We propose two new software primitives for these handshakes between the persistence thread and the remainder of software:

- **TXFence**: A transaction fence - this returns only after all open transactions have retired but new transactions may start in other threads.
- **TXBarrier**: A transaction barrier - this disallows opening a new transaction until all transactions have retired.

TXFence is useful for reading into registers and using some variables non-transactionally, but ensuring that if they have been potentially updated in preceding transactions, then their updated values have been copied to home locations. TXBarrier allows for multiple threads with concurrent transactions to converge on a single point and read shared values non-transactionally after completing a parallel section.

4. Implementation

In this section we describe our implementation of cc-HTM. There are two major components: (1) an Alias Table used as shadow memory for transactions to prevent corruption due to cache spilling; and (2) a pair of queues designated as Blue and Red queues, to order persistent writes consistently. The Blue queue holds transactions that have not entered the WAIT state (called blue transactions), while the Red queue will hold those that are in the WAIT state (called red transactions).

4.1 Alias Table

The structure of the Alias Table is shown in Figure 1. It is implemented as a DRAM-resident, set associative key-value store that holds the values of transaction updates. Access to NVM transaction variable \(X\) is redirected to a shadow DRAM variable \(X'\) allocated in the Alias Table. The aliased entry holds the address of \(X\), its value, and a timestamp field \(TStart\) that is used to reclaim the space after \(X\) has been persisted in NVM. Most recent values of variables are found in either the Alias Table or the home location (if retired from the log and reclaimed from the Alias Table).

Transaction reads and writes are implemented as calls to a user library shown in Algorithm 1. The **Read** library call first checks the Alias Table for the requested address using the function **getAliasTableEntry**. The function returns the entry in the table that matches the passed address or, if no entries match, returns the entry in the associative set with the smallest timestamp as a possible candidate for reclamation. The **Read** returns the value in the table if found; else it performs a normal LOAD from the NVM address of the variable and returns the value.

The **Write** library call similarly checks the Alias Table. If it is found in the table the entry is updated with the new value; else

\(^1\) Strictly this ordering applies only to conflicting transactions. Transactions which don’t share variables can be ordered arbitrarily.

\(^2\) An alternative to persisting the startTS timestamp at the start of a transaction is to write a “completed bit” to the log in NVM when the transaction in the priority queue is ready for retirement; “completed bits” must be persisted in priority queue order. The recovery procedure only restores these logs in the order of their persistTS.
if there is a stale entry that can be reclaimed, then the address, new value, and the start timestamp of the transaction doing the write are entered in the reclaimed entry. If no entry is available, the transaction explicitly aborts. Note we do not try to evict entries from the Alias Table to make space, but instead use a simple aging mechanism to reclaim stale entries. This reduces contention for the Alias Table that would otherwise cause many spurious aborts due to accesses for table management. To facilitate conflict-free reclamation each entering transaction sets a private variable myObservedMin to the lowest start timestamp of all transactions in the system. If an Alias Table entry has a timestamp smaller than this value, the last transaction that wrote that entry has retired and the space can be safely reclaimed.

### 4.2 cc-HTM Lock-Free Algorithm

We present the details of the transaction lifecycle described in Section 3. Transaction interface routines are shown in Algorithm 1 and Queue management routines are shown in Algorithm 2. In the START state, the transaction invokes the library function TransactionStart of Algorithm 1. The function QueueNotifyStart returns the (unique non-decreasing) transaction id that also serves as its start timestamp startTS. The current minimum start timestamp of all transactions currently in the system is computed by the function QueueGetMin and is saved locally in myObservedMin. The transaction then enters the EXECUTE state by executing XBegin and begins its HTM section. Read and write operations within the HTM section are implemented by accesses to the Alias Table as described in Section 4.1, with writes additionally appending to the log.

After the last instruction of the HTM section the library function TransactionEnd is invoked. This reads the platform timer using RDTSCP, ends the HTM transaction with XEnd, and enters the FINISH state. The transaction obtains the end timestamp, records the persistence timestamp, and writes its log to NVM. The log is persisted by a sequence of CLWB (cacheline writebacks) instructions that write back the values cached during the HTM execution, followed by a persistent memory fence. The function QueueNotifyEnd moves the transaction to its correct location in the priority queue. The transaction then enters the WAIT state by calling function QueueTxCommit. The transaction will be released from the WAIT state and will commit when it is signaled by the persistence management thread. As discussed previously this can be controlled with the choice of lag parameter (higher L allows transactions to commit early).

#### Algorithm 1: cc-HTM Transaction Implementation

```
TransactionStart ()
    myId = QueueNotifyStart();
    myObservedMin = QueueGetMin();
    Open Log and persist startTS=myId;
    XBegin();

TransactionEnd ()
    myPersistentTS = RDTSCP();
    XEnd();
    QueueNotifyEnd(myId, myPersistentTS);
    Persist cached Log and timestamps to NVM;
    QueueTxCommit(myId, Log);

Read (address)
    Entry *e = getAliasTableEntry(address);
    if (e->address = address))
        return e->Value;
    else return +address;

Write (address, value)
    Entry *e = getAliasTableEntry(address);
    // Abort if entry not found and min entry not retired:
    if (e->id > myObservedId) XAbort();
    // Save the write and tag with myId start time:
    e->Address = address;
    e->Value = value;
    e->TStart = myId;
    Append (address, value) to cached Log;

getAliasTableEntry (address)
    line = getLine(address);
    for (entry = each element in line) {
        if (entry->address = address)
            return entry;
        if (entry->TStart < minTStart) {
            minTStart = entry->TStart;
            minEntry = entry;
        }
    }
    return minEntry;
```

When a transaction enters the system its transaction id is used to allocate a slot in the Blue queue that is populated from the tail and freed from its head. When a transaction enters the Red priority queue it marks itself as deleted in the Blue queue. The head of the Blue queue points to the earliest transaction that has not been marked for deletion, and the tail is the last entry that has been allocated. If the transaction at the head of the Blue queue is marked as deleted, the head pointer is advanced sequentially until it points to an entry that has not been deleted or reaches the tail of the queue. The amortized time per deletion is a constant.

The Red queue is organized as a singly-linked list arranged in increasing order of the persistence timestamps, permitting simple concurrent insertion and deletion using Compare-and-Swap instructions [12]. When changing color from Blue to Red in QueueNotifyEnd, the persistence timestamp of the transaction is used to find its position in the Red Queue using function FindInsertNode. Using the head pointers of the Blue and Red queues, entering transactions can update myObservedMin in constant time by comparing the start timestamps of the entries at the head of the two queues.

Transaction retirement is managed by a Persistence Management Thread shown in Figure 1. When it is safe to retire a transaction it reads log records of the entry at the head of the Red priority
Algorithm 2: Persistence Management Queue

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueueNotifyStart()</td>
<td>AtomicFetchAndAdd(&amp;numElements, 1);</td>
</tr>
<tr>
<td></td>
<td>id = AtomicFetchAndAdd(&amp;TxCounter, 1);</td>
</tr>
<tr>
<td></td>
<td>Q[ti].startTS = id; // Save token number.</td>
</tr>
<tr>
<td></td>
<td>sfence();</td>
</tr>
<tr>
<td></td>
<td>return id; Retain the transaction token.</td>
</tr>
<tr>
<td>QueueNotifyEnd(id, persistentTS)</td>
<td>currentTxCounter = TxCounter;</td>
</tr>
<tr>
<td></td>
<td>ti = id &amp; Mask;</td>
</tr>
<tr>
<td></td>
<td>Q[ti].persistentTS = persistentTS;</td>
</tr>
<tr>
<td></td>
<td>while (true) {</td>
</tr>
<tr>
<td></td>
<td>i = FindInsertNode(ti, persistentTS);</td>
</tr>
<tr>
<td></td>
<td>if (i == HEAD) {</td>
</tr>
<tr>
<td></td>
<td>if (CAS(&amp;redHead, Q[ti].nextRed, ti)) break;</td>
</tr>
<tr>
<td></td>
<td>} else if (CAS(&amp;(Q[i].nextRedIndex), Q[i].nextRedIndex, ti)) break;</td>
</tr>
<tr>
<td></td>
<td>Q[token].endTS = currentTxCounter; sfence();</td>
</tr>
<tr>
<td>QueueGetMin()</td>
<td>return TxMin;</td>
</tr>
<tr>
<td>QueueTxCommit(startTS, logPointer)</td>
<td>Q[startTS &amp; Mask].logPointer = logPointer; sfence();</td>
</tr>
<tr>
<td></td>
<td>while (numElements &gt; 0);</td>
</tr>
<tr>
<td>FindInsertNode(node, persistentTS)</td>
<td>// Walk Red Queue find persistentTS insertion point</td>
</tr>
<tr>
<td></td>
<td>// with increasing time. Return parent node index.</td>
</tr>
<tr>
<td>CheckRetirement()</td>
<td>ri = redHead;</td>
</tr>
<tr>
<td></td>
<td>if ((Q[ri].endTS &gt; 0) and (Q[ri].logPointer != 0)) and (Q[ri].endTS &lt;= UpdatedBlueHead()) {</td>
</tr>
<tr>
<td></td>
<td>next = Q[ri].nextRedIndex;</td>
</tr>
<tr>
<td></td>
<td>if ((CAS(&amp;(Q[i].nextRedIndex), next, next+FLAG)) return;</td>
</tr>
<tr>
<td></td>
<td>if (CAS(&amp;redHead, ri, next)) {</td>
</tr>
<tr>
<td></td>
<td>Retire(Q[ri].logPointer);</td>
</tr>
<tr>
<td></td>
<td>if (Q[ri].startTS == TxMin)</td>
</tr>
<tr>
<td></td>
<td>CAS(&amp;TxMin, Q[ri].startTS, UpdatedMin());</td>
</tr>
<tr>
<td></td>
<td>Q[ri].startTS = Q[ri].endTS = 0;</td>
</tr>
<tr>
<td></td>
<td>Q[ri].logPointer = Q[ri].priorityTS = 0;</td>
</tr>
<tr>
<td></td>
<td>Q[ri].nextRedIndex = EMPTY; sfence();</td>
</tr>
<tr>
<td></td>
<td>AtomicAdd(&amp;numElements, -1);</td>
</tr>
<tr>
<td>} else Q[ri].nextRedIndex = next;</td>
<td></td>
</tr>
<tr>
<td>UpdatedMin()</td>
<td>// Walk from TxMin to next non-empty Q[i].startTS</td>
</tr>
<tr>
<td>UpdatedBlueHead()</td>
<td>static blueMin = 0;</td>
</tr>
<tr>
<td></td>
<td>if (blueMin == TxCounter) return blueMin;</td>
</tr>
<tr>
<td></td>
<td>while (Q[blueMin &amp; MASK] != 0)</td>
</tr>
<tr>
<td></td>
<td>blueMin++;</td>
</tr>
<tr>
<td></td>
<td>return blueMin;</td>
</tr>
<tr>
<td>RetirementThread()</td>
<td>while (!done OR (numElements &gt; 0)) CheckRetirement();</td>
</tr>
<tr>
<td>Retire(log)</td>
<td>walk log and write *address=value; sfence();</td>
</tr>
</tbody>
</table>

Figure 2: Lock Free Data Structure Examples

queue and writes them to their home locations. The thread monitors the Red queue, and retires the transaction at the head when its startTS is smaller than the startTS of the entry at the head of the Blue queue. This ensures that it is the transaction with the smallest persistence timestamp in the system (Claim 2; Section 3). After retirement the element at the head of the Red Queue must be deleted from the list. A lock-free implementation in which the element is marked with a flag for removal from the linked list before removal is used to prevent concurrent insertions from swapping into the next pointer simultaneously.

An example is shown in Figure 2. Both Blue and Red queues share the same statically allocated circular buffer of transaction entries. The logical Blue Queue contains elements 11 and 14, while the logical Priority Red Queue has 12, 9 and 13. The minimum of both queues has a TxMin of 9. When an element is added to the Blue Queue, it is simply placed in the tail, the TxCounter. When one of the Blue colored elements is ready to change to Red, the endTS time is saved with the persistTS time. The element is added to the Red queue by Atomic compare-and-swap after finding the index in the Red Queue. When the TxMin index element is retired, e.g. removed from the Red Queue, in this case element 9 at index 1, the new TxMin is found by walking until the next non-empty element, 11 in this example.

5. Evaluation

For evaluation, we employed Intel(R) Xeon(R) E5-2699 v3 series processors, 18 cores per processor, running at 2.30 GHz, with Red Hat Enterprise Linux 7. We used numacll to restrict all threads to a single processor (socket) for repeatable results and reducing interference from other background activities during measurements. HTM transactions were implemented with Intel Transactional Synchronization Extensions (TSX) [15] using a global fallback lock. Processor micro-code was patched to enable TSX support. A set of runtime environment variables allowed for the easy configuration of Alias Table sizes, persistence methods such as cc-HTM or no persistence, and maximum allowable lag. We built our software using g++ 4.8.2. Each measurement reflects an average over twenty repeats with small variation among the repeats.

Using micro-benchmarks and SSCA2 [3] and vacation, from the STAMP [20] benchmark suite, we compared the following methods:
High Productivity Computing Systems [3], The Scalable Synthetic Compact Applications for benchmarking 5.1 Benchmarks

Increasing Alias Table sizes, the benchmark can take longer to the graph. We executed the multiple kernels that construct a graph and perform operations on STAMP Stanford Transactional Applications for Multi-Processing [20], or NVM and is over 1.5 times faster than a persistence method to set associativity. We also set the maximum allowable Lag number of threads from 1 to 16 in powers of two and recorded the generations a graph with over 45 million edges. We increased the having to perform persistent flushes within its concurrent region. yields a significantly weaker scalability due to the inherent costs of the cc-transaction performs its sequence of writes. The undo-log entries are written with write-through stores and SFENCEs, and once the TL2 transaction commits and the new data values are flushed into NVM, the undo-log entries are removed.

5.1 Benchmarks

The Scalable Synthetic Compact Applications for benchmarking High Productivity Computing Systems [3], SSCA2, is part of the Stanford Transactional Applications for Multi-Processing [20], or STAMP, benchmark suite. It uses a large memory area and has multiple kernels that construct a graph and perform operations on the graph. We executed the SSCA2 benchmark with scale 20, which generates a graph with over 45 million edges. We increased the number of threads from 1 to 16 in powers of two and recorded the execution time for the kernel for each method.

For cc-HTM, we use an Alias Table of Size 64 MB with 2-way set associativity. We also set the maximum allowable Lag \( L \) to 512. Later in this section we explore variations in these parameters.

Figure 3 shows the execution time for each method for the Compute Kernel in the SSCA2 benchmark as a function of the number of threads. Each method reduces the execution time with increasing numbers of threads. Our cc-HTM approach has similar execution time to HTM in the cache hierarchy with no persistence and is over 1.5 times faster than a persistence method to NVM with PTL2. Figure 4 shows the speedup for each method as a function of the number of threads. Even though the HTM (cache-only) method does better in absolute terms as we saw in Figure 3, it proceeds from a higher baseline for single-threaded execution. The cc-HTM method matches its speedup. And even though PTL2 has the advantage of a low single thread number for comparison, it yields a significantly weaker scalability due to the inherent costs of having to perform persistent flushes within its concurrent region.

Figure 5 shows the execution time of the SSCA2 Compute Graph Kernel with varying Alias Table sizes and associativity. With increasing Alias Table sizes, the benchmark can take longer to execute, due to having more cache misses for Alias Table entries and having to retrieve entries from main memory. A transaction will abort if it does not find a matching entry in the Alias Table and cannot find a free slot it can reclaim-- that is, as we described earlier, the case when there is an alias table collision with a set containing values that were modified in a transaction that is still pending retirement. Therefore a smaller Alias Table or one with lower associativity, can have more conflicts and can cause more aborts. However, too large an Alias Table will suffer from cache performance, having to reach back to main memory, lengthening the time of transactions and also potentially causing more aborts. We find Alias Tables in the size of 64-256 MB with 2-4 way associativity perform well in general.

Next, we investigated the effect of the the maximum \( L \) (we chose this as 512) on parallel execution time. First, we counted the number of non-retired transactions colored Blue and Red over time in a subset of the Compute Graph Kernel in the SSCA2 Benchmark. We used the default graph scale of 20 and performed evaluations with 8 concurrent threads. Initially, we set the maximum allowable Lag \( L \) to 512 and recorded the results in Figure 6. The total number of non-retired transactions never grows past 150 over the entire execution time of the benchmark. The number of Red transactions depends on long running transactions that have not yet published their completion times. A delayed or long running transaction thread might hold up the persistence of several transactions that have completed in each of several threads. Once the long running transaction is complete, changing color from Blue to Red, then other transactions can be persisted. A fast thread might have created numerous outstanding transactions waiting on persistence but is allowed to continue in the foreground path. Figure 7 shows how the maximum allowable Lag can affect the overall performance. We use 8 threads and vary the maximum allowable Lag \( L \) in the SSCA2

![Figure 3: SSCA2 Benchmark Compute Graph Kernel Execution Time as a Function of the Number of Parallel Execution Threads](image)

![Figure 4: SSCA2 Benchmark Compute Graph Kernel Speedup as a Function of the Number of Parallel Execution Threads](image)

![Figure 5: SSCA2 Benchmark Compute Graph Kernel Time with Varying Alias Table Sizes and Associativity for 8 Threads](image)
benchmark and show that after $L$ is increased to $125^3$ throughput is insensitive to additional increases in Lag and cc-HTM achieves peak transactional throughput.

We also evaluated the vacation benchmark which is part of the STAMP benchmark suite. The vacation benchmark emulates database transactions for a travel reservation system. We executed the benchmark with the low option for lower contention emulation. Figure 8 shows the execution time for each method for the vacation benchmark as a function of the number of threads. Each method reduces the execution time with increasing numbers of threads. Our cc-HTM approach follows the trends similar to HTM in the cache hierarchy with no persistence, with both approaches flattening execution time after 4 threads. As thread contention increases, all experience more aborts, but PTL2 can spin and retry with individual locks to gain extra parallelism under high contention, and thus it continues to improve, albeit slowly, while cc-HTM and HTM-Only stop reducing their runtimes.

Additionally, we examined the effect of increased NVM write times on the benchmark. Byte-addressable, persistent non-volatile memory is characterized by longer write times. To emulate the longer write times for NVM, we insert a delay after non-temporal stores when writing to new cache lines and a delay after cache line flushes. The write delay can be tuned to emulate the effect of longer write times typical of NVM. Figure 9 shows the vacation benchmark execution time for various NVM write times. Our cc-HTM method is less affected by increasing NVM write times than the PTL2 approach due to several factors. First, cc-HTM performs write-combining for log entries on the foreground path for each thread, so writes to several transaction variables may be combined into a fewer writes. Additionally, PTL2 transactionally persists an undo log on writes causing a foreground delay for each write.

5.2 Hash Table

Our next series of experiments show how read / write ratios and transaction sizes affect overall performance. We create a 64 MB Hash Table Array of elements in main memory and transactionally perform a number of element updates. For each transaction, we generate a set of random numbers of a configurable size, compute their hash, and write the value into the Hash Table Array.

First, we create transactions consisting of 10 atomic updates and vary the number of concurrent threads and measure the maximum
throughput. We perform 1 million updates and record the average throughput with maximum allowable Lag set to 256 and plot the results in Figure 10. This experiment shows that throughput increases with the number of threads until transactions start conflicting and affect the overall throughput. For up to 5 concurrent threads cc-HTM achieves roughly 2x throughput over PTL2. Next, we fix the number of concurrent threads to 6 and vary the maximum allowable Lag parameter $L$ over a broad range and record the average throughput. The results in Figure 11, like SSCA2, show cc-HTM achieves peak throughput with Lag $L$ at 125-150 even for write intensive workloads.

The transaction write set was then varied from 2 to 30 elements with 6 concurrent threads. The average throughput was recorded and is shown in Figure 12. With just 2 elements in the write set, cc-HTM performs slightly slower than PTL2, due to overhead in the

5.3 Red-Black Tree

In the final set of experiments, we use the transactional Red-Black tree from STAMP [20] initialized with 1 million elements. We then perform insert operations on the Red-Black tree and record average transaction times and throughput over 200k additional inserts. Figure 14 shows the average response time versus transaction request arrival rate for 4 concurrent processing threads. Each transaction inserts an additional element into the Red-Black tree after it is initialized with 1 million elements. As shown in the Figure, cc-HTM has a much faster response time (by a factor of 4 to 5) over PTL2 and close to non-persistent cache-based HTM Only.

Next, we record the maximum throughput of inserts into the Red-Black tree per second for a varying number of threads. The average throughput over 1 million insertions is shown in Figure
15. As can be seen in the Figure, cc-HTM has a much higher throughput over PTL2, and thread contention on the data structure causes both methods to fall off. cc-HTM can have almost 4-5x improvement over eager persistence locking. Inserting an element into a Red-Black tree first requires finding the insertion point which can take many read operations. In our experiments, we averaged 63 reads and 11 writes per transactional insert of one element into the Red-Black tree. Next, we vary the Lag parameter \( L \) and record the maximum throughput in Figure 16. In this more read-heavy workload, cc-HTM achieves peak throughput with \( L \) at only 80.

Finally, we vary the NVM write time similar to our prior experiment. Figure 17 shows the Red-Black tree throughput as a function of the NVM write time. As the write time increases both our cc-HTM method and the PTL2 method have decreasing throughput.

6. Related Work

Related Persistence Work: Analysis of consistency models for persistent memory was considered in [23]. Changes to the frontend cache for ordering cache evictions were proposed in [6, 16, 29, 31]. BPFS [6] proposed epoch barriers to control eviction order, while [29] proposed a flush software primitive to control of update order. Snapshotting the entire micro architectural state at the point of a failure is proposed in [21]. Memory controller support for transaction atomicity in NVM have been proposed in [8, 10, 24]. FIRM [32] describes techniques to differentiate persistent and non-persistent memory traffic. In comparison to these works, the solution proposed in this paper does not require any hardware changes, and capitalizes on the property that were an HTM transaction to abort, none of its partial updates need to be retracted from the cache hierarchy.

**Related Concurrency Work:** Existing non-HTM solutions [4, 5, 30] tightly couple concurrency control with durable writes of either write-ahead logs or data updates into NVM to maintain persistence consistency. Software that employs these approaches generally means they must extend the duration for which they remain in critical sections, leading to longer times to hold locks, which reduces concurrency and expands transactional duration. Other work [11, 17] decouples concurrency control so that post transactional values may flow through cache hierarchy and reach NVM asynchronously; however, the write ahead log for an updating transaction has to get committed into NVM synchronously before the transaction can close so that the integrity of the foreground value flow is preserved across machine restarts. Another hardware-assisted mechanism proposes hardware changes to allow a dual-scheme checkpointing that writes previous check-pointed values in the background while collecting current transaction writes [27].

Recent work [1, 2, 17] aims to exploit processor-supported HTM mechanisms for concurrency control instead of traditional locking or STM-based approaches. However, all of these solutions require making significant changes to the existing HTM semantics and implementations. For instance, PHTM [2] and PHyTM [1], propose a new instruction called TransparentFlush which can be used to flush a cache line from within a transaction to persistent memory without causing any transaction to abort. They also propose a change to the xend instruction that ends an atomic HTM region, so that it atomically updates a bit in persistent memory as part of its execution. Similarly, for DUDETM [17] to use HTM, it requires that designated memory variables within a transaction be allowed to be updated globally and concurrently without causing an abort. One difficulty with proposals for changing hardware is the uncertainty around whether and when such proposals are embraced by processor manufacturers.

7. Summary

Systems with large core counts coupled with large, byte-addressable non-volatile memory offer the potential of breakthrough performance on problems with large data footprints, by enabling massively parallel in-memory applications. Hardware Transactional Memory creates the possibility that parallel applications can run with minimal entanglement among threads, impeded only by actual data races, instead of by preventive software synchronization. However, since HTM implementations typically make all changes to memory visible instantly, they create a window of vulnerability for transactions whose updates flow out of order across the memory buses and potentially leave NVM based data and structures in a broken state in the event of an untimely machine crash.

In this paper we presented cc-HTM, a continuous checkpointing of Hardware Transactional Memory to persistent memory, which requires no hardware changes and runs on existing Intel hardware. The paper presents light-weight mechanism to propagate the updates from a completed HTM transaction to NVM asynchronously but atomically and serializably across transactions. By introducing a small lag parameter, we create a continuous wave of transactions that are persisted to NVM fronted by transactional HTM concurrency sections. Using micro-benchmarks and benchmarks from the STAMP suite, we showed that the lock-free implementation of our data structures enables fast, parallel atomic persistence to NVM. With small values of the lag parameter, cc-HTM achieves its peak throughput, at which it significantly outperforms persistent software transactions in both throughput and response time, and it compares well with volatile HTM transactions.

Acknowledgments

Supported by NSF Grant CCF 1439075 and Intel SSG.
Appendix

Proof of Claim 1

Let $I_1$ denote the interval between $X\text{Begin}$ and $X\text{End}$ for transaction $T_1$. The claim holds trivially when $T_1$ and $T_2$ are non-overlapping in time i.e. $I_1 \cap I_2 = \emptyset$ (Figure 18(a)). If $T_1$ and $T_2$ overlap in time, then $T_2$ could not have accessed any of the variables it shares with $T_1$ in the interval $I_1 \cap I_2$ (Figure 18(b)); otherwise one of $T_1$ or $T_2$ must have been aborted by the HTM (see Figure 18(c)). Hence the timestamp of $T_2$, which is taken after the last instruction of $T_2$ before $X\text{End}$, must occur after $T_1$ gets its persistent timestamp. Note that a similar claim cannot be made if $T_1$ and $T_2$ are non-conflicting (Figure 18(d)). In this case, the timestamps of $T_1$ and $T_2$ can be in the opposite order of their completion.

Figure 18: Txns $T_1$ and $T_2$ with persistent timestamps $TS_1$ and $TS_2$.

Proof of Claim 2

Partition the transactions be into two sets $R$ and $B$ consisting of transactions in the PQ (i.e. those in the WAIT state) and those not in the PQ respectively. Since the PQ is ordered in increasing order of persistence timestamps, the element at the head of the PQ has the smallest persistence timestamp among all transactions $R$. If the endTS of a transaction $P$ is smaller than the startTS of a transaction $Q$, then the persistence timestamp of $P$ (which is less than its endTS must be less than the persistence timestamp of $Q$ (which will be greater than its endTS). Hence the persistence timestamp of $T$ must be less than the persistence timestamps of all transactions in $B$.

References


