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. We exploit the ordering primitives in 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.

1. Introduction and Overview
Two emerging hardware developments provide raise the potential for transformative gains in speed and scalability of massively parallelized in-memory data processing: (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) makes it straightforward for threads to work spontaneously in shared memory spaces without having to synchronize explicitly.

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.

Current solutions for transaction durability and atomicity in NVM vary in the degree of processor support they require: (a) pure software approaches craft solutions using software algorithms and data structures, with a common characteristic that a log is created and made durable on NVM whilst changes are made; (b) others propose significant changes to existing cache hardware and protocols or use external hardware controllers without disrupting the processor core. Concurrency control can use traditional locking to isolate transactions or be handled within an STM [1]. Recent work [2][3] 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.

The technique described in this paper achieves lightweight, fine-grained checkpointing for durable state in NVM without the need for frequent and expensive snapshots of a machine’s memory state and without requiring any hardware changes.

2. 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 which updates get committed to NVM home locations is established using a fine-grained monotonic persistence-timestamp from within the HTM execution phase, 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, which provides the required temporal reasoning about a consistent order for committing potentially overlapping updates from HTM transactions. 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 after XEnd from corrupting the NVM with partial transactional updates, all transaction variables are aliased to shadow volatile locations within the HTM.

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.

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. Writes are also logged in a private log that will be written to NVM after XEnd. To create a consistent persist order, just before executing XEnd, the transaction reads the platform-wide timestamp counter by executing RDTSWC. Note that reading the timestamp counter (step 4) can be done within an HTM transaction without causing an abort, and it is made the last instruction to be completed before XEnd.

This is achieved by inserting the finished transactions in a priority queue. Thus, the end timestamp of the head of the priority queue 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. Thus, the end timestamp of the head of PQ must be less than or equal to the start timestamp of all in-flight transactions. A separate persistence thread is responsible for safely retiring transactions and signaling the waiting transaction.

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. Implementation

There are two major components of cc-HTM: (1) an Alias Table used as shadow memory for transactions to prevent corruption due to cache spillage; 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).

The Alias Table (AT), see Figure 1, 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 AT. 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 AT or the home location (if retired and reclaimed from the Alias Table).

A transactional Read first checks the AT for the requested address and finds the entry in the AT that matches the passed address or, if no entries match, it performs a normal LOAD from the NVM address of the variable and returns the value. A transactional Write similarly checks the AT, and if the variable is found, the entry is updated with the new value; else if there is a stale entry that can be reclaimed, the address, new value, and the start timestamp of the transaction doing the write are entered in the claimed entry. If no entry is available, the transaction explicitly aborts.

4. Summary

Continuous Checkpointing HTM 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. cc-HTM 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.

References