Hardware Transactional Persistent Memory

Ellis Giles
Rice University
ACM Student Member

Peter Varman
Rice University
Advisor

ABSTRACT
This research solves the problem of creating durable transactions in byte-addressable Non-Volatile Memory or Persistent Memory (PM) when using Hardware Transactional Memory (HTM)-based concurrency control. It shows how HTM transactions can be ordered correctly and atomically into PM by the use of a novel software protocol. We design a new persistence method that decouples HTM concurrency from back-end PM operations. Using efficient lock-free mechanisms, failure atomicity is achieved using redo logging coupled with aliasing to guard against mistimed cache evictions. A back-end distributed memory controller alternative provides a hardware implementation choice for catching PM cache evictions. Our approach compares well with standard (volatile) HTM transactions and yields significant gains in latency and throughput over other persistence methods.

1 INTRODUCTION AND OVERVIEW
Two emerging hardware developments 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 or Persistent Memory (PM), such as Intel’s 3D XPoint™ technology, and (b) the availability of CPU-based transaction support with Hardware Transactional Memory or HTM.

HTM, originally designed for volatile memory, makes it straightforward for threads to work concurrently in shared memory spaces with hardware supported isolation control. Once an HTM transaction closes, its updates become visible en masse through the cache hierarchy and can travel in any order to memory DIMMs.

This works well for DRAM, however, instantly visible transactional values in the cache followed by a random cache eviction can corrupt PM backed transactions on a system failure. Furthermore, logging based software approaches are problematic for HTM since 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. Transaction ordering uses a fine-grained monotonic persistence-timestamp from within the HTM execution phase, without the danger of causing inter-thread memory collisions. A start counter is used for tracking open transactions and tagging stored values. The structure of a transaction is shown in Figure 2.

Figure 1: Problems Persisting HTM Transactions to PM

Figure 2: Transaction Structure

Recent work [1–4] 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 or the front-end cache. Our approach creates durable HTM transactions to PM that operate on existing commodity hardware using a novel software protocol [5]. A back-end memory controller alternative provides an option for performance gains [6].

2 OUR APPROACH
Our approach persists an HTM transaction onto durable PM 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. Transaction ordering uses a fine-grained monotonic persistence-timestamp from within the HTM execution phase, without the danger of causing inter-thread memory collisions. A start counter is used for tracking open transactions and tagging stored values. The structure of a transaction is shown in Figure 2.
We evaluated our method using benchmarks directly running on hardware using an Intel(R) Xeon(R) E5-2650 v4 series processor with 12 cores, running at 2.20 GHz, with Red Hat Enterprise Linux 7.2. HTM transactions were implemented with Intel TSX using a global fallback lock. We built our software using g++ 4.8.5.

The transaction can choose between strict durability, where, on completion, values are fully-recoverable to PM, or relaxed durability, where values will be persisted safely at some point in the future, allowing for increased performance.

There are two major components of the software solution: (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, while the Red queue holds those that are in the WAIT state. The full protocol is described in [5].

The Alias Table (AT), see Figure 3, is implemented as a DRAM-resident, set associative key-value store that holds the values of transaction updates. Most recent values of variables are found in either the AT or the home location (if retired and reclaimed from the AT). Transactional loads first consult the AT, and if the address is not found loads the value from PM. Similarly, stores update an existing entry in the AT or create a new entry, while reclaiming space from retired entries. If no entry is available, the transaction explicitly aborts. A back-end thread manages retirement of transactions.

**Memory Controller Alternative:**

In an alternative implementation choice, support can be added to the back-end memory controller. The protocol is modified so that cache evictions are held in the controller until they are safe to flow to PM. No software aliasing is required, thereby executing with little overhead. Protocol steps 1 and 5 instead notify the controller to PM. No software aliasing is required, thereby executing with relaxed durability, where, on completion, values are fully-recoverable to PM, or strict durability, where values will be persisted safely at some point in the future, allowing for increased performance.

**3 EVALUATION**

We evaluated our method using benchmarks directly running on hardware using an Intel(R) Xeon(R) E5-2650 v4 series processor with 12 cores, running at 2.20 GHz, with Red Hat Enterprise Linux 7.2. HTM transactions were implemented with Intel TSX using a global fallback lock. We built our software using g++ 4.8.5.

**REFERENCES**