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^{TM} 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 store through the cache aborts the transaction. Logging outside of the transaction may be subject to arbitrary processor delays and thus complicate both the persistence method and recovery. See Figure 1.

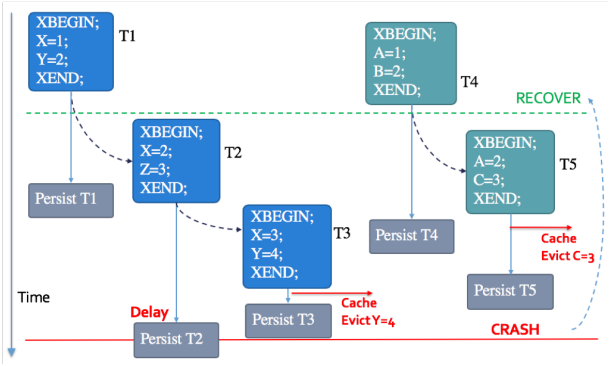

Figure 1: Problems Persisting HTM Transactions to PM

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.

```
AtomicBegin
----------  State: START  ----------
 1 myId= FetchAndIncrement(TxCounter)
 2 myIndex= myId % Qsize
 3 TXQueue[myIndex].startTS= myId

----------  State: EXECUTE  ----------
 XBegin
 // Transaction Body of HTM
 //   All transaction reads and writes are aliased
 //   in shadow DRAM and recorded in a log
 4 myPersistTS= RDTSCP()
 XEnd

----------  State: FINISH  ----------
 5 TXQueue[myIndex].endTS= Read(TxCounter)
 6 TXQueue[myIndex].persistTS= myPersistTS
 7 Persist the transaction log in PM
 8 TXQueue[myIndex].logAddress= PM Address of log
 9 Place TXQueue[myIndex] into a priority
    queue PQ ordered by their persistTS fields

----------  State: WAIT  ----------
 // Wait for transaction to be retired.
 // Retirement thread updates PM home
 // locations and notifies waiting thread.
AtomicEnd
```

Figure 2: Transaction Structure

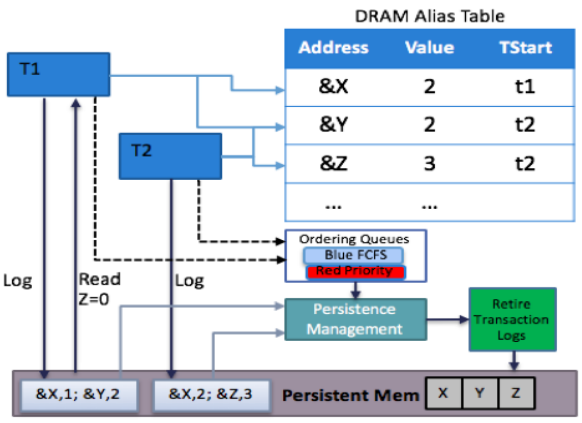

Figure 3: Software Using DRAM Aliasing and Ordering Queues

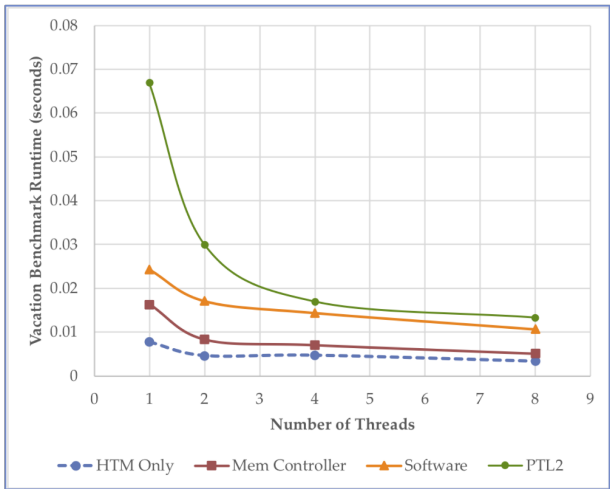

Figure 4: Vacation Benchmark Execution Time as a Function of the Number of Parallel Execution Threads

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) and 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 (through streaming stores) of transaction open and closes so that the controller can selectively track transaction boundaries and allow values to flow to PM safely.

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.

Using micro-benchmarks and benchmarks from the STAMP benchmark suite, we compared our solution to transactions running on HTM cache only with no persistence and a persistent STM solution based on TL2. Figure 4 shows results from the *Vacation* benchmark. Our solution tracks closely with HTM Only with no persistence and has a significant improvement over PTL2.

4 SUMMARY

Our solution is a novel technique for all-or-nothing updates to PM locations in the course of HTM transactions, so that the use of HTM can be extended seamlessly from volatile to durable memories. The solution does not require any changes to existing Intel HTM instructions or semantics and uses only the announced Intel instructions for PM. 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

[1] H. Avni, E. Levy, and A. Mendelson, "Hardware transactions in nonvolatile memory," in *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, ser. DISC 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 617–630.

[2] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent transactional memory," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 58–61, Jan 2015.

[3] H. Avni and T. Brown, "PHyTM: Persistent hybrid transactional memory," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 409–420, 2016.

[4] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building Durable Transactions with Decoupling for Persistent Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 329–343. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037714

[5] E. Giles, K. Doshi, and P. Varman, "Continuous Checkpointing of HTM Transactions in NVM," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2017. New York, NY, USA: ACM, 2017, pp. 70–81. [Online]. Available: http://doi.acm.org/10.1145/3092255.3092270

[6] ——, "Hardware Transactional Persistent Memory," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. New York, NY, USA: ACM, 2018, pp. 192–207.