Brief Announcement: Hardware Transactional Storage Class Memory

Ellis Giles
Rice University
Houston, TX 77005, USA
erg@rice.edu

Kshitij Doshi
Intel Corporation
Chandler, AZ 85226, USA
kshitij.a.doshi@intel.com

Peter Varman
Rice University
Houston, TX 77005, USA
pjv@rice.edu

ABSTRACT

Emerging persistent memory technologies (generically referred to as Storage Class Memory or SCM) hold tremendous promise for accelerating popular data-management applications like in-memory databases. However, programmers now need to deal with ensuring the atomicity of transactions on SCM-resident data and maintaining consistency between the persistent and in-memory execution orders of concurrent transactions. The problem is specially challenging when high-performance isolation mechanisms like Hardware Transaction Memory (HTM) are used for concurrency control.

In this work we show how SCM-based HTM transactions can be ordered correctly using existing CPU instructions, without requiring any changes to existing processor cache hardware or HTM protocols. We describe a method that employs HTM for concurrency control and enforces atomic persistence and consistency with a novel software protocol and back-end external memory controller. In contrast, previous approaches require significant hardware changes to existing processor microarchitectures.

1 INTRODUCTION

Emerging storage class memory (SCM) technologies, such as Intel/Micron's 3DXPoint provide direct access to vast amounts of persistent memory through familiar byte/word grained LOAD and STORE instructions, hitherto employed to access the volatile memory hierarchy. This provides performance and programmability benefits by eliminating the I/O operations and bottlenecks common to many data management applications. It also raises problems of durability (power failures leaving pending updates in volatile buffers), atomicity, and consistency (power failures while SCM is partially updated with autonomous out-of-order cache evictions).

Existing SCM programming frameworks separate into categories based on the degree of change they require in the processor architecture for such problems. Pure software approaches (Mnemosyne [14], SoftWrap [7], ATLAS [3], REWIND [4], DUDETM [12]) work with existing processor capabilities. Approaches like Kiln [16], ATOM [10] and SPMS [11] require significant change to existing cache hardware and protocols in the processor microarchitecture, while solutions like WrAP [6], and NVM-Controller [5] only require external hardware controllers not affecting the processor core.

SPAA '17, July 24-26, 2017, Washington DC, USA
 978-1-4503-4593-4/17/07
DOI: http://dx.doi.org/10.1145/3087556.3087589

The solutions mentioned above either use two-phase locking protocols to isolate concurrent transactions or handle concurrency within the rubric of an STM [14]. PTM [15] proposes changes to processor caches while adding an on-chip scoreboard and global transaction id register to couple HTM with SCM. Recent work [1, 2, 12] has attempted to provide inter-transactional isolation by employing processor based Hardware Transactional Memory (HTM) [9, 13] mechanisms. However, these solutions all require 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 an HTM transaction to SCM without causing transaction aborts. DUDETM [12] proposes allowing concurrent access to designated memory variables within an HTM without causing an abort. Selective and incremental changes to the clean isolation semantics of HTM are not to be undertaken lightly; understanding their impact on global system correctness and performance typically requires long gestation periods before processor manufacturers will embrace them.

In this paper we provide a new solution to obtain persistence consistency in NVM while using HTM for concurrency control. The solution does not alter the processor micro architecture, but leverages an external NVM memory controller along with a persistence protocol to supplement the existing HTM transaction semantics. The solution aims to achieve the concurrency benefits of HTM by allowing transactions to operate at the speed of in-memory volatile transactions, while using backend operations to guard against and recover to a consistent volatile state in the event of failure.

2 OVERVIEW

Consider transactions A and B of Listings 1 & 2 that update variables x, y and y, z respectively. The directives **HTMBegin** and **HTMEnd** demarcate code sections serialized by the HTM. Each transaction maintains the invariant $x + y + z = 1$ when executed by itself. When executed concurrently, the HTM aborts conflicting transactions to maintain the abstraction of isolated execution.

When x, y, z are persistent variables (*i.e.*, homed in SCM), additional correctness constraints arise. Suppose transaction A executes the HTM section; at **HTMEnd**, the new values of x and y become visible in the cache hierarchy but may or may not have been written to SCM. The subsequent code may use a sequence of CLWB instructions to force their write back from cache to SCM; however a crash that occurs before all the updates have been flushed can leave the SCM inconsistent. Furthermore, the cache itself may autonomously evict some of the updated variables due to cache pressure, resulting in loss of control over the precise state of the SCM at any time.

In [1, 2] the HTM semantics were changed to allow log records to be persisted to SCM from *within* an HTM transaction using a newly

Listing 1: Transaction A

```
A() {
  AtomicBegin();
  HTMBegin;
  x = 2;
  y = -1-z;
  HTMEnd;
  AtomicEnd();
}
```

Listing 2: Transaction B

```
B() {
  AtomicBegin();
  HTMBegin;
  y = -1- x;
  z =  2;
  HTMEnd;
  AtomicEnd();
}
```

proposed instruction *TransparentFlush*. With this mechanism, when a transaction completes *HTMEnd*, a log of all its writes would be persistent in SCM. Since the HTM mechanism isolates all writes and prevents the cached values from being evicted till *HTMEnd*, this ensures that there will be no spurious cache write backs till all the log records are persisted; should a subsequent crash occur, recovery proceeds from the persisted log records. There are two drawbacks to this solution: first, transactions are slowed down and complete only after its log records are written to the slower memory tier; and second, it is difficult to predict whether and when such intrusive hardware changes, which change established memory operation semantics, will be adopted by processor manufacturers.

Our solution to this problem is presented in the next section. The HTM transaction is enclosed within an atomic section demarcated by the primitives **AtomicBegin** and **AtomicEnd**. A transaction commits at **AtomicEnd** at which time its writes become durable (at least in a recoverable log). Logs are written to SCM *outside* the HTM transaction, and thus no change is required to HTM semantics or mechanisms. Non-conflicting transactions execute concurrently without waiting for a previous transaction to commit and their speed and ordering is determined solely by the HTM; logging operations continue in the background without slowing down the front-end HTM transactions.

3 OUR APPROACH

To couple persistence with HTM concurrency we need to solve three challenges: (a) Spurious Cache Evictions (b) Transaction Ordering and (c) Persisting the updates.

Cache Evictions: The HTM mechanism ensures that no updates will be evicted from the cache to SCM between **HTMBegin** and **HTMEnd**. However, cache evictions after **HTMEnd** and before the log is persisted can cause inconsistency in the SCM in case of a crash. We use the non-intrusive NVM controller proposed in [5] to address this problem. The controller implements a victim cache that holds evictions of SCM variables from the LLC until it is safe to update the SCM home locations of updated variables. A protocol for safely reclaiming entries from the victim cache while spilling the contents of the saved logs to the home locations was described in [5]. Here also we use the NVM controller to guard against spurious updates to SCM by cache evictions. However we need to modify the persistence protocol significantly: specifically, the conditions under which log retirements and victim cache deletions are triggered.

Transaction Ordering: To order transactions we use the platform-wide clock available on Intel architectures. Specifically, we use the platform timer read instruction RDTSCP to timestamp transactions. These timestamps will determine when and in what order the transactions are retired to SCM. Each transaction maintains two timestamps: a *start timestamp* (STS) that is set before the transaction executes **HTMBegin** and an *end timestamp* (ETS) that is set immediately before executing **HTMEnd**. The end timestamps will determine the transaction retirement order, so it is necessary that they correctly reflect transaction execution ordering in volatile memory. Specifically, if the HTM orders transaction A before a conflicting transaction B, then the end timestamp of A should be smaller than that of B. Since instruction delays are unpredictable, the instant that ETS is set must be correctly chosen. If two transactions have disjoint variable sets or are non-overlapping (the last invocation of one begins after the other's end), then any choice of the time stamping instant within their HTM sections will order them consistently. The tricky situation is when two transactions share variables and overlap their execution, but neither is aborted because the actual writes (or reads) to the shared variables in one transaction occur only after the second has completed its **HTMEnd**. In this case, if reading the timer is made the last instruction before the end of the HTM section, then we are guaranteed that the order of end timestamps will correctly reflect the consistency order [8].

We obtain the global system time stamp counter using the new Intel instruction RDTSCP or "read time stamp counter and processor ID". This instruction does not need to be preceded by a serializing CPUID instruction. However, later instructions may get reordered before RDTSCP; to prevent the reordering of an XEND before reading the timestamp counter into registers, we save the resulting time stamp into volatile memory, which, at the end of the HTM transaction after XEND, is guaranteed to instantly reflect all memory operations contained between XBEGIN and XEND.

Persisting Updates: After **HTMEnd**, a transaction writes its log records to SCM along with its end timestamp. The controller will retire the transactions in order of their ETS values. In retiring a transaction, the controller must persist the updates to the home locations of the variables, delete the log of the transaction, and reclaim any of these cache lines that may have spilled into the victim cache. To implement the retirement phase we employ two logical structures: a *blue* list and a *red* priority queue ordered by end timestamps. The structure can be implemented as a single lock-free data structure [8]. After setting its start timestamp, a new transaction places itself at the tail of the blue list of open transactions and executes **HTMBegin**. Just prior to completing its HTM section it sets the end timestamp. Following **HTMEnd**, the transaction persists its logs to SCM, inserts itself into the red priority queue in order of its end timestamp, and deletes itself from the blue list. A retirement thread retires the transaction at the head of the priority queue when it is safe: this occurs when all transactions in the blue list have start timestamps that are greater than its end timestamp. Since the head of the blue list has the earliest start timestamp, the retirement thread only needs to examine the head element.

A transaction can commit any time after being inserted into the priority queue. If it waits till retirement to commit then it is guaranteed durability of its updates. Otherwise, if a transaction with smaller ETS has not persisted its logs before a crash, its log may not be replayed on recovery. The application can choose between early commit versus guaranteed durability. In any case, SCM will be restored to a consistent previous state on recovery.

Algorithm 1: Transaction Wrapper Library and Controller

Wrap Library:

`OpenWrapC` *()*
- *wrapId* = OpenWrap(); *// Triggers HandleOpenWrap*
- Initialize write set *ws*; // For logging the updates
- *memory fence*;
- HTMBegin(); *// HTM Begin call XBEGIN*
- *On abort call AbortWrap*;

`wrapStore` *(***addrVar**, **Value***)*
- Add {*addrVar, Value*} to *ws*;
- Normal store of *Value* to *addrVar*;

`CloseWrapC` *()*
- *t* = RDTSCP(); *// Set end time stamp in volatile memory*
- HTMEnd(); *// XEND*
- PublishRTC(*wrapId*, *t*); *// Triggers HandlePublishedRTC*
- **for** ({*addrVar, Value*} in *ws*)
  - Add ({*addrVar, Value*} to SCM log;
- *memory fence*;
- CloseWrap(*wrapId*); *// Triggers HandleCloseWrap*

Controller:

`HandleOpenWrap` *(***wrapId** *w)*
- Mark *w* with start timestamp = RDTSCP();
- Add *w* to tail of Blue List;

`HandlePublishedRTC` *(***wrapId** *w*, **time** *endTime)*
- Add *w* to Red Priority Queue with *endTime* value;
- Remove *w* from Blue List;

`HandleCloseWrap` *(***wrapId** *w)*
- Mark *w* in Red Priority Queue as complete;

3.1 Algorithm

The controller catches cache evictions of SCM variables and holds them in a volatile victim cache till their home locations have been updated by transaction retirements. It tracks the set of open transactions in a bit vector and uses it to tag evicted cache lines with a dependency set. When a transaction retires it is removed from all dependency sets. When the dependency set of a cache line is empty it can be deleted from the victim cache. For details see [5].

Algorithm 1 shows the controller actions and the software wrapper to invoke its functions. We refer to an HTM transaction and the associated logging and retirement operations as a *wrap*.

OpenWrapC opens a wrap, which triggers HandleOpenWrap. It then opens an HTM transaction with HTMBegin (instruction XBEGIN). The HTM transaction detects concurrency conflicts in the transaction. On an HTM abort, the HTM transaction can be reattempted with exponential back-off or revert to a global fallback lock after several attempts. Additionally, if an HTM transaction is aborted, the associated wrap is also aborted.

wrapStore is a normal write, but also records the address and value of the store to an in-cache memory log to be saved to SCM on successful HTM end. Not every write within the HTM transaction needs to be wrapped. Only the final write to a variable needs to create a log record. However, creating multiple log records for a variable will not affect correctness.

CloseWrapC sets the end timestamp of the transaction using RDTSCP. Following a successful end of the HTM section it moves the wrap to the appropriate place in the priority queue based on its end timestamp, persists the log, and then marks the entry in the priority queue as ready for retirement.

The SCM Controller retires logs in order of end timestamps provided all prior opened transactions have completed. **HandlePublishedRTC** moves a wrap from the Blue list to the Red Priority Queue with priority of the end timestamp. **HandleCloseWrap** simply marks the wrap, which is in the Red Priority Queue, as complete.

For retirement, the head of the Red Priority Queue is examined. If the head element is marked as complete and its end timestamp is less than the start timestamp of all transactions in the Blue list then the head element in the Red Queue may be removed and variables in the wrap retired. Retiring the wrap simply reads the log, copies values safely to home locations, and after all variables are safely committed, removes the log.

3.2 Recovery

The Red Priority Queue and Blue list are not saved in persistent SCM. However, the transaction start timestamp, end timestamp, and log completion flag are stored in the log in persistent memory. This allows for easy reconstruction of the state of the queues during recovery following a failure.

For recovery, we read all logs and rebuild the Blue list and Red Queue. We then process the completed transactions starting from the head of the Red queue that have end timestamps that are less than the smallest start timestamp in the Blue list. This brings the system back into the closest possible point for in-memory consistency.

3.3 NVM Controller Variant

The NVM controller presented in [5] retires transactions directly from transaction logs and removes elements from the victim cache when the dependency set becomes empty. In a variant on the NVM controller, we can instead propagate elements directly from the victim cache to the backend SCM when the dependency set of a cache line in the victim cache becomes empty and use logs only for recovery. A number of alternative implementations are possible in this approach *e.g.* explicitly evict transaction values from the processor caches using CLWB instructions after a transaction ends its concurrency section and periodic fence instructions (which guarantee durability of the flushed values in recent Intel architectures) to allow saved logs to be deleted. The details are deferred to the full paper.

3.4 Example

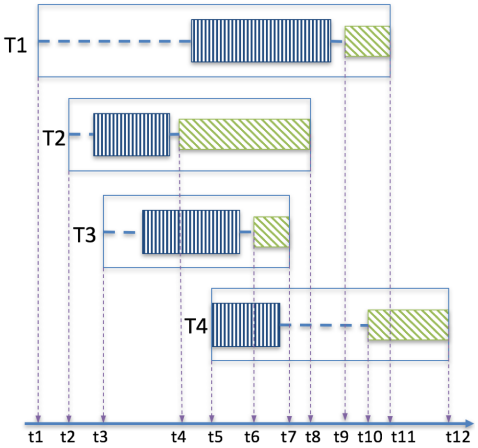
Figure 1a shows example transactions A and B from Listings 1 and 2 using the Wrap Library functions from Algorithm 1. Variables *x,y* and *y,z* are updated using *wrapStore* within *OpenWrapC()* and *CloseWrapC()* bounds.

Figure 1b shows an example set of four transactions, T1-T4, happening concurrently. The Table shown in Figure 1c illustrates the contents of the Blue list and Red priority queue at each time step indicated in Figure 1b.

```
A() {
    OpenWrapC();
    wrapStore(x, 2);
    wrapStore(y, -1-z);
    CloseWrapC();
}

B() {
    OpenWrapC();
    wrapStore(y, -1- x);
    wrapStore(z, 2);
    CloseWrapC();
}
```

(a) Example Wrapped Transactions A & B



(b) Example Transactions T1-T4

Time	Blue List	Red Priority
t1	T1	
t2	T1, T2	
t3	T1, T2, T3	
t4	T1, T3	T2
t5	T1, T3, T4	T2
t6	T1, T4	T2, T3
t7	T1, T4	T2, **(T3)**
t8	T1, T4	**(T2)**, **(T3)**
t9	T4	**(T2)**, **(T3)**, T1
t9+	T4	**(T3)**, T1
t10		**(T3)**, T4, T1
t11		T4, **(T1)**
t12		**(T4)**, **(T1)**

(c) Contents of Blue List and Red Queue

Figure 1: Example Transactions and Contents of Blue List and Red Priority Queue

First, at time $t1$, T1 opens and places itself with its start timestamp at the end of the Blue list. At times $t2$ and $t3$, transactions T2 and T3 are added to the end of the Blue list with their start timestamps. T2 then completes its concurrency section and publishes its end timestamp at time $t4$ and begins writing its logs, moving from Blue to Red. Transaction T4 starts at time $t5$ and is added to the Blue list. T3 completes its concurrency section and publishes its end timestamp at $t6$, moving from the Blue to Red queue.

At time $t7$, transaction T3 has completed writing its logs and is marked completed in the Red queue, denoted by a bold and circled entry. However, T3 cannot be retired since it is not the head of the Red queue and T1, the head of the Blue list, has a smaller start timestamp, which could potentially place it near the front of the Red queue. When T2 completes writing of logs at time $t8$, it is marked as complete in the Red queue, but also has a later completion time than T1, the head of the Blue list.

When T1 finally completes its concurrency section and publishes its end timestamp at $t9$, T1 is moved from the Blue list to the Red queue and T2 is retired. At time $t9+$, T3, the new head of the Red priority queue, is complete with a published end timestamp and log. However, at this time, T3 cannot be retired since it is waiting on T4, which hasn't yet published its end time and had started before T3 completed.

At time $t10$, T4 publishes its end timestamp and moves from the Blue List to Red Queue. Note that T4 has an ending time before T1, so it is placed **before** T1 in the Red Priority Queue. T1 completes log writing at $t11$ and is marked ready for retirement, but is behind T4, which hasn't yet finished writing its logs. At time $t12$, T4 completes writing its logs and both T4 and T1 can be retired in order.

ACKNOWLEDGMENTS

Supported by NSF Grant CCF 1439075 and Intel SSG.

REFERENCES

[1] Hillel Avni and Trevor Brown. 2016. PHyTM: Persistent Hybrid Transactional Memory. *Proceedings of the VLDB Endowment* 10, 4 (2016), 409–420.

[2] Hillel Avni, Eliezer Levy, and Avi Mendelson. 2015. Hardware transactions in nonvolatile memory. In *International Symposium on Distributed Computing*. Springer, 617–630.

[3] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[4] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* 8, 5 (2015), 497–508.

[5] Kshitij Doshi, Ellis Giles, and Peter Varman. 2016. Atomic Persistence for SCM with a Non-intrusive Backend Controller. In *The 22nd International Symposium on High-Performance Computer Architecture*. IEEE. Early version in Comp. Arch. Letters, Vol. 15 Issue 1.

[6] Ellis Giles, Kshitij Doshi, and Peter Varman. 2013. Bridging the programming gap between persistent and volatile memory using WrAP. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 30. Early version in MeAOW 2012.

[7] Ellis Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. 1–14. https://doi.org/10.1109/MSST.2015.7208276 Early version in MeAOW 2013.

[8] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Continuous Checkpointing of HTM Transactions in NVM. In *Proceedings of the 2017 International Symposium on Memory Management (ISMM '17)*. ACM, New York, NY, USA.

[9] Maurice Herlihy and J Eliot B Moss. 1993. *Transactional memory: Architectural support for lock-free data structures*. Vol. 21,2. ACM.

[10] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture*.

[11] Shuo Li, Peng Wang, Nong Xiao, Guangyu Sun, and Fang Liu. 2017. SPMS: Strand Based Persistent Memory System. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 622–625.

[12] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. 2017. DUDETM: Building Durable Transactions with Decoupling for Persistent Memory. In *To Appear: ASPLOS 2017*.

[13] Ravi Rajwar and James R Goodman. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 294–305.

[14] H. Volos, A. J. Tack, and M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 91–104.

[15] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. 2015. Persistent transactional memory. *IEEE Computer Architecture Letters* 14, 1 (2015), 58–61.

[16] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744