Brief Announcement: Hardware Transactional Persistent Memory

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

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

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

ABSTRACT

This paper addresses 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 combined with a Persistent Memory Controller, without requiring changes to processor cache hardware or HTM protocols. In contrast, previous approaches require significant changes to existing processor microarchitectures. Our approach compares well with standard (volatile) HTM transactions and yields significant gains in latency and throughput over other persistence methods.

ACM Reference Format:

Ellis Giles, Kshitij Doshi, and Peter Varman. 2018. Brief Announcement: Hardware Transactional Persistent Memory. In *SPAA '18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, July 16–18, 2018, Vienna, Austria*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3210377.3210656>

1 INTRODUCTION

This paper provides a solution to the problems of adding *durability* to concurrent in-memory transactions that use Hardware Transactional Memory (HTM) for concurrency control while operating on data in byte-addressable Non-Volatile Memory or Persistent Memory (PM). In existing PM programming frameworks, transaction isolation is achieved in software within the rubric of an STM [1] or by using two-phase locking [2]; a log of updates is saved in persistent memory before the transaction completes, to permit failure recovery. However, logging based software approaches are problematic for HTM transactions (e.g., Intel TSX) which cannot bypass the caches in order to flush the log records synchronously into PM ahead of transaction closings. For these, PTM [3] proposes changes to processor caches while adding an on-chip scoreboard and global transaction id register to couple HTM with PM. Recent work [4–6] has attempted to provide inter-transactional isolation by employing processor based (HTM) [7, 8] mechanisms. However, these solutions require changes to the existing HTM semantics and implementations – [4] and [5] propose a new instruction to perform a non-aborting cacheline flush from within an HTM, while [6]

proposes allowing non-aborting concurrent access to designated memory variables within an HTM. A software-based solution for this problem without any hardware changes or an external controller was presented in [9].

In this paper we provide a new solution to obtain durable consistency in Persistent Memory while using HTM for concurrency control. The solution does not alter the processor microarchitecture, but leverages a simple, external Persistent Memory Controller along with a persistence protocol, to supplement the existing HTM semantics and allowing HTM transactions to operate at the speed of in-memory volatile transactions. The solution, while it achieves the concurrency benefits of HTM for PM-based data, also applies to non-HTM transactions in a straightforward way.

2 PERSISTENT HTM TRANSACTIONS

The code section between instructions **XBegin** and **XEnd** constitute an *HTM transaction* or simply a transaction. HTM ensures the *atomicity* and *isolation* of transaction execution. Within an HTM transaction, all updates are made to private locations in the cache, and the hardware guarantees that the updates are *not allowed* to propagate to their home locations in PM until **XEnd** completes.

Atomic Persistence: *Following crash recovery, ensure that all or none of the updates of an HTM transaction are stored in their PM home locations.* Following **XEnd**, transaction variables are subject to normal cache operations like evictions and write-back instructions. If the machine crashes before all the transaction updates are written back to PM, then on reboot, the values of these variables in PM will be inconsistent with the pre-crash transaction values. A common solution is to first log transaction updates in persistent storage before permitting updates to the PM home locations. However, with HTM the log can only be written to PM after **XEnd**, at which time the updates are also susceptible to uncontrolled cache evictions into PM. Hence there is no guarantee that the log has been persisted before transaction updates have percolated into PM.

Persistence Ordering: *Following crash recovery, ensure that the persistent state of any sequence of dependent transactions is consistent with their execution order.* For instance, if three dependent transactions A, B, C complete their **XEnd** in that order, we require that following crash recovery the PM variables reflect execution of either no transactions, {A}, {A, B}, or {A, B, C}. With HTM the logs can only be persisted after the transaction **XEnd** and the order in which log writes complete can differ from the transaction completion order. Approaches like ordering using a shared atomic counter within transactions are not practical due to the resulting HTM-induced aborts and serialization of transactions. Other papers have advocated changes to processor HTM semantics and implementation to allow selective writes to PM from within an HTM [4–6].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SPAA '18, July 16–18, 2018, Vienna, Austria

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5799-9/18/07.

<https://doi.org/10.1145/3210377.3210656>

Strict and Relaxed Durability: *Following crash recovery, strict durability requires that the updates of a committed transaction are persistent in PM.* Strict durability is achieved in ACID databases by persisting transaction logs in order before commit. This is difficult to do for HTM transactions since transaction order and log completion order proceed independently. It is possible that an earlier transaction X (on which Y depends) to have completed but not yet persisted its log. During crash recovery the log of Y should not be replayed since X cannot be replayed.

We define a new property known as *relaxed durability* that allows an individual transaction to opt for an early commit immediately after it persists its log. The choice between relaxed or strict durability is made individually by a transaction based on its application requirements. Relaxed-durability transactions face a window of vulnerability after they commit, during which a crash may result in their transaction updates not being reflected in PM after recovery. The gain is reduced transaction latency. However, irrespective of individual durability choices, PM will always recover to a consistent state *i.e.* a prefix of every sequence of dependent transactions.

3 HTM+PM SOLUTION

We obtain durability of HTM transactions in PM using a coordinated protocol involving a back-end Persistent Memory Controller, transaction timestamps and logging, and the failure recovery procedure. The PM controller intercepts dirty cache lines evicted from the last-level processor cache (LLC) and holds them temporarily in a (volatile) FIFO queue within the controller. It delays the evicted cache lines from writing to PM until it can guarantee safety (atomicity and ordering) of the update. The controller is pre-programmed with the address range of a region of PM that is reserved for holding transaction logs. Addresses in the log region simply pass through the controller without intervention.

HTM+PM transactions execute independently. Within an outer-envelope that enforces consistency between updates to the volatile cache hierarchy and the durable state in PM, transactions use an unmodified HTM to serialize the computational portions of conflicting transactions. A transaction (1) notifies the controller when it opens and closes, (2) saves start and end timestamps in PM to enable consistent recovery after a failure, (3) performs its HTM operation, and (4) persists a log of its updates in PM before closing. A transaction requiring strict durability informs the controller during its closing step, and then waits for the go-ahead from the controller before committing. A relaxed-durability transaction can commit immediately after it closes.

The recovery routine is invoked after a system crash to restore the PM variables to a valid state *i.e.* a state that is consistent with the actual execution order of dependent transactions. It uses the saved logs to recover the values of the updated variables, and the saved start and end timestamps to determine which logs are eligible for replay and their replay order.

3.1 Transaction Lifecycle

A transaction can be viewed as progressing through five states: **OPEN**, **COMPUTE**, **LOG**, **CLOSE** and **COMMIT** as shown in Listing 1. When a transaction begins, it notifies the Persistent Memory Controller with a small unique integer (*wrapId*) that identifies the

transaction. The controller adds *wrapId* to a set of Currently Open Transactions (referred to as COT) that it maintains. The transaction then allocates and initializes space in PM for its log and updates the *startTime* record of the log. The *startTime* is obtained by reading a system wide platform timer using the *RDTSCLP* instruction implemented in the *ReadPlatformCounter* function. In addition to *startTime*, a log includes a second timestamp *persistTime* that will be set just prior to completing the HTM transaction. The *writeSet* is a sequence of (*address, value*) pairs in the log that will be filled with the locations and values of the updates made by the HTM transaction. The log with its recorded *startTime* is then persisted using cache line write-back instructions (*clwb*) and *sfence*.

The transaction then enters the **COMPUTE** state by executing **XBegin** and entering the HTM code section. Within the HTM section, the transaction updates *writeSet* with the persistent variables that it writes. Note that the records in *writeSet* will be held in cache during the **COMPUTE** state since it occurs within an HTM and cannot be propagated to PM until **XEnd** completes. Immediately before **XEnd** the transaction obtains a second timestamp *persistTime* that will be used to order the transactions correctly. This timestamp is also obtained using the same *RDTSCLP* instruction.

After executing **XEnd**, a transaction next enters the **LOG** state. It flushes its log records from cache hierarchy into PM using cache line write-back instructions (*CLWB* or *CLFLUSHOPT*), following the last of them with an *SFENCE*. This ensures that all the log records have been persisted. After the *SFENCE* following the log record flushes, the transaction enters the **CLOSE** state.

In the **CLOSE** state the transaction signals the Persistent Memory Controller that its log has been persisted in PM. The controller removes the transaction from its set, COT, of currently open transactions. It also reflects the closing in the state of evicted cache lines in its FIFO buffer as described below in Section 3.2. A transaction requiring strict durability informs the controller at this time; the controller will signal the transaction in due course when it is safe to commit *i.e.* its updates are guaranteed to be durable.

The transaction is then complete and enters the **COMMIT** state. If it requires strict durability it waits till it is signaled by the controller. Otherwise, it immediately commits and leaves the system.

3.2 Persistent Memory Controller

The Persistent Memory Controller is shown in Figure 1. While superficially similar to an earlier design [2, 10, 11], this controller handles races arising from the use of HTM rather than locking for concurrency control, and makes significant simplifications by delegating some PM state maintenance functions to the recovery protocol.

A primary function of the controller is to prevent a transaction's updates from reaching PM until it is safe for it to do so, without knowing which transactions generated a specific update. It does this by enforcing two requirements before allowing an evicted dirty cache line (representing a transaction's update) from proceeding to PM: (i) ensuring that the log of the transaction has been made persistent, and (ii) guaranteeing that the saved log will be replayed during recovery. The first condition is needed (but not sufficient) for atomic persistence by guarding against a failure that occurs after a partial subset of a transaction's updates have been persisted

Listing 1: Transaction Structure

```

Transaction Begin
----- State: OPEN -----
1 NotifyPMController(open);
2 Allocate a Log structure in PM;
3 nowTime = ReadPlatformCounter();
4 Log.startTime = nowTime;
5 Log.writeSet = {};
6 Persist Log in PM;
----- State: COMPUTE -----
XBegin
// Transaction Body of HTM
// All reads and writes to transaction
// variables are performed and writes
// appended to Log.writeSet.
7 endTime = ReadPlatformCounter();
XEnd

----- State: LOG -----
8 Log.persistTime = endTime;
9 Persist Log in PM;

----- State: CLOSE -----
10 NotifyPMController(close);

----- State: COMMIT -----
11 If (strict durability requested)
    Wait for notification by controller;
12 Transaction End

```

Figure 1: PM Controller with Volatile Delay Buffer, Current Open Transactions Set, and Dependency Wait Queue

in PM. The second requirement arises because transaction logs are persisted outside the HTM, and there is no relation between transaction dependency order and the order in which their logs are persisted. To maintain correct persistence ordering, the recovery routine may not be able to replay a saved log. We illustrate the issue in the example below.

Example: Consider two dependent transactions A and B, A: $\{w = 3; x = 1\}$ and B: $\{y = w + 1; z = 1\}$. Assume that HTM transaction A

executes before B, but that B persists its logs and closes before A. Suppose there is a crash just after B closes. The recovery routine will correctly refuse to replay either A and B, since the log of the earlier transaction A is not available.

Now consider the situation where y (value 4) is evicted from the cache and then written to PM after B persists its log, just prior to a crash. Once again neither log will be replayed. However, atomic persistence is now violated since B's updates are partially reflected in PM. Note that this violation occurred even though the write back of y to PM happened after B's log was persisted. The Persistent Memory Controller protocol prevents a write back to PM unless it can also guarantee that the log of the transaction creating the update will be played back on recovery (see Lemma C2 in Section 3.4).

The second function of the controller is to track when it is safe for a transaction requiring strict durability to commit. It is not sufficient to commit when a transaction's logs are persistent on PM since, as seen in the example above, the recovery routine may not replay the log if that would violate persistency ordering. The controller protocol effectively delays a strict-durability transaction τ from committing until the earliest open transaction has a *startTime* greater than the *persistTime* of τ . This is because the recovery protocol will replay a log (see Section 3.3) if and only if all transactions with *startTime* less than its *persistTime* have closed.

Implementation Overview:

The controller tracks transactions by maintaining a set S of currently open transactions (COT). When a transaction opens, its identifier is added to COT and when the transaction closes it is removed. The write to PM of a cache line C evicted into the Persistent Memory Controller is deferred by placing it at the tail of a FIFO queue maintained by the controller. When C enters the Persistent Memory Controller it is assigned a tag called its *dependency set* initialized to the current value of COT.

The controller holds the evicted instance of C in the FIFO until all transactions that are in its dependency set (i.e. S) have closed. When a transaction closes it is removed from both the COT and from the dependency sets of all the FIFO entries. When the dependency set of a cache line in the FIFO becomes empty, it is eligible to be flushed to PM. One can see that the dependency sets will become empty in the order in which the cache lines were evicted, since a transaction that is still in the dependency set of C when a new eviction enters the FIFO will also be in the dependency set of the new entry. This simple protocol guarantees that all transactions that opened before cache line C was evicted into the controller (which must also include the transaction that last wrote C) must have closed and persisted their logs when C becomes eligible to be written to PM. This also implies that all transactions with *startTime* less than the *persistTime* of the transaction that last wrote C would have closed, satisfying the condition for log replay. Hence the cache line can be safely written to PM without violating atomic persistence.

Note that the evicted cache lines intercepted by the controller do not hold any identifying transaction information and can occur at arbitrary times after the transaction leaves the COMPUTE state. The cache line could hold the update of a currently open transaction or could be from a transaction that has completed or even committed and left the system. To guarantee safety, the controller must perforce assume that the first situation holds.

3.3 Recovery

Each completed transaction saves its log in PM. The log holds records *startTime* and *persistTime* obtained by reading the platform timer using the RDTSCP instruction. We refer to these as the start and end timestamps of the transaction. The start timestamp is persisted *before* a transaction enters its HTM. This allows the recovery routine to identify a transaction that started but had not completed at the time of a failure. Note even though such a transaction has not completed, it could still have finished its HTM and fed values to a later dependent transaction which has completed and persisted its log. The end timestamp and the write set of the transaction are persisted *after* the transaction completes its HTM section, followed by an end of log marker. There can be an arbitrary delay between the end of the HTM and the time that its log is flushed from caches into PM and persisted.

The recovery procedure is invoked on reboot following a machine failure. The routine will restore PM values to a consistent state that satisfies persistence ordering by copying values from the *writeSets* of the logs of qualifying transactions to the specified addresses. A transaction qualifies for log replay if and only if all earlier transactions on which it depends (both directly and transitively) are also replayed.

Implementation Overview:

The recovery procedure first identifies the set of incomplete transactions $\mathcal{I}$, which have started (as indicated by the presence of a *startTime* record in their log) but have not completed (indicated by the lack of a valid end-of-record marker). The remaining complete transactions (set $\mathcal{C}$) are potential candidates for replay. Denote the smallest start timestamp of transactions in $\mathcal{I}$ by T_m . A transaction in $\mathcal{C}$ is valid (qualifies for replay) if its end timestamp (*persistTime*) is no more than T_m . All valid transactions are replayed in *increasing order* of their end timestamps *persistTime*.

3.4 Protocol Properties

We now summarize the invariants maintained by our protocol.

Definition: The precedence set of a transaction T , denoted by $\text{prec}(T)$, is the set of all dependent transactions that were ordered by the HTM before T .

Lemma C1: Consider a transaction X with a precedence set $\text{prec}(X)$. For all transactions Y in $\text{prec}(X)$, $\text{startTime}(Y) < \text{persistTime}(X)$.

Lemma C2: Consider transactions X and Y with $\text{startTime}(Y) < \text{persistTime}(X)$. If a cache line C that is updated by X is written to PM by the controller at time t , then Y must have closed and persisted its log before t .

Proof Sketch: Suppose C was evicted to the controller at time $t' \leq t$. Now t' must be later than the time X completed HTM execution and set $\text{persistTime}(X)$; by assumption this is after Y set its *startTime* at which time Y must have been registered as an open transaction by the controller. Now, either Y has closed before t' or is still open at that time. In the latter case, Y is included in the dependence set of C at t' . Since C can only be written to PM after its dependence set is empty, it follows that Y must have closed and removed itself from the dependence set of C .

Lemma C3 Any transaction X whose update is written to PM at time t will be replayed by the recovery routine if there is a crash any time after t .

Proof The recovery routine will replay a transaction X if the only incomplete transactions (started but not closed) at the time of the crash started after X completed; that is, there is no incomplete transaction Y that has a $\text{startTime}(Y) \leq \text{persistTime}(X)$. By Lemma C2 such an incomplete transaction cannot exist.

4 IMPLEMENTATION AND EVALUATION

In our full paper, we detail the implementation of the controller and user software library. We evaluate our approach using both micro-benchmarks and the STAMP suite. Our approach compares well with standard (volatile) HTM transactions. In comparison with persistent transactional locking, our approach performs 3x faster on standard benchmarks and almost 9x faster on a persistent Red-Black Tree data structure.

5 SUMMARY

In this paper we presented an approach that unifies HTM and PM to create durable, concurrent transactions. Our approach works with existing HTM and cache coherency mechanisms, does not require changes to existing processor caches or store instructions, and only utilizes logs for recovery. The solution correctly orders HTM transactions and atomically commits them to Persistent Memory by the use of a novel software protocol combined with a back-end Persistent Memory Controller.

ACKNOWLEDGMENTS

Supported by NSF Grant CCF 1439075 and Intel SSG.

REFERENCES

- [1] H. Volos, A. J. Tack, and M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 2011, pp. 91–104.
- [2] K. Doshi, E. Giles, and P. Varman, "Atomic Persistence for SCM with a Non-intrusive Backend Controller," in *The 22nd International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, March 2016.
- [3] 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.
- [4] H. Avni, E. Levy, and A. Mendelson, "Hardware transactions in nonvolatile memory," in *International Symposium on Distributed Computing*. Springer, 2015, pp. 617–630.
- [5] H. Avni and T. Brown, "PHyTM: Persistent hybrid transactional memory," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 409–420, 2016.
- [6] 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*. ACM, 2017, pp. 329–343.
- [7] M. Herlihy and J. E. B. Moss, *Transactional Memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21.2.
- [8] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001, pp. 294–305.
- [9] 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>
- [10] —, "Brief announcement: Hardware transactional storage class memory," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2017, pp. 375–378.
- [11] L. Pu, K. Doshi, E. Giles, and P. Varman, "Non-Intrusive Persistence with a Backend NVM Controller," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 29–32, Jan 2016.