# Persistent Transactional Memory

Zhaoguo Wang†, Han Yi†‡, Ran Liu†‡, Mingkai Dong†, Haibo Chen†
†Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
‡School of Computer Science, Fudan University

**Abstract**—This paper proposes persistent transactional memory (PTM), a new design that adds durability to transactional memory (TM) by incorporating with the emerging non-volatile memory (NVM). PTM dynamically tracks transactional updates to cache lines to ensure the ACI (atomicity, consistency and isolation) properties during cache flushes and leverages an undo log in NVM to ensure PTM can always consistently recover transactional data structures from a machine crash. This paper describes the PTM design based on Intel's restricted transactional memory. A preliminary evaluation using a concurrent key/value store and a database with a cache-based simulator shows that the additional cache line flushes are small.

**Index Terms**—Hardware Transactional Memory, Non-volatile Random Access Memory

## 1 Introduction

Transactional memory (TM) [6], as a concept borrowed from database transactions, essentially ensures atomicity, isolation and consistency (ACI) of a group of memory operations. However, TM lacks an important property from database transactions, namely durability, which ensures a transaction can persist permanently once committed.

On the other hand, recent hardware advances in non-volatile memory (NVM) [1], [9] have made durability of key data structures in a program a necessity to preserve key program invariants during a machine crash. Though NVM has already enabled persistence of in-memory data structures, non-volatile on-chip structures may cause such data structures to violate the ACI properties during crashes, due to unordered cache line flushes.

However, adding durability to TM is hard due to potentially high hardware and/or performance overhead. One approach would be adding a battery to CPU chip and flushing all cache lines to memory before a power failure. However, such an approach is not reliable due to issues with diminishing battery volume and limited battery lifecycle. Further, it is hard to identify the dependency among cache lines and blind cache flushes may still violate program invariants. Another approach is using a write-through cache to flush all cache lines of a transaction to NVM once committed. However, this not only incurs prohibitive performance overhead, but also cannot ensure atomicity due to non-atomic cache flushes.

This paper proposes persistent transactional memory (PTM), a new design that adds durability to TM by combining TM with NVM. Strict durability in database usually requires making the working set of a transaction durable before the acknowledgement of a transaction commit. This, however, requires flushing related cache lines to NVM during a transaction commit, and thus will cause significant performance overhead due to no reuse of cached data. Hence, PTM trades freshness for performance by ensuring *eventual persistence:* a committed transaction either are persistent as a whole or not persistent at all, while still respecting consistency and isolation among committed transactions. Hence, PTM preserves the ACI properties of committed transactions even under power outage, machine crashes or system errors.

PTM introduces only small hardware cost to CPU and NVM. It adds a small-sized scoreboard to CPU that tracks dependency of committed transactions to respect their orders. Each cache line is extended with 8-bit transaction ID to identify which transaction this cache line belongs to, such that all cache lines of a transaction are flushed as a whole. A transaction ID register is added to a CPU chip to uniquely identify a transaction. To ensure all-or-nothing during flushing multiple cache lines to memory, PTM uses an undo log in NVM during flushing a group of cache lines, by first checkpointing related data to a log space before writing cache lines to their home locations.

We implement PTM on a cache simulator, by modeling Intel's restricted transactional memory (RTM) design. PTM assumes a three-level cache, using the general MESI [8] cache coherence protocol[1], and uses L1 cache to detect transactional conflicts. Evaluation using a concurrent key/value store [7] and a concurrent database (LevelDB [5]) shows that PTM only incurs only 10% more cache line flushes on average.

## 2 Challenges and Related Work

### 2.1 Violation of Transaction Semantics

Simply integrating transactional memory (referred to as SITM) with NVM may violate atomicity, consistency and isolation during a machine crash.

**Atomicity:** A transactional region may contain updates to multiple cache lines, while SITM can only flush a cache line into NVM at a time. Hence, as shown in figure 1-(a), in the case of a machine crash, only parts of the committed cache lines (e.g., cache line B) may be persisted. As a result, the transaction can only recover parts of its data (e.g., cache line B) after a machine restarts, which violates the atomicity property of transactional memory.

*Manuscript submitted: 24-Dec-2013. Manuscript accept: 01-Apr-2014. Final manuscript received: 07-Apr-2014. This research is supported in part by a Foundation for the Author of National Excellent Doctoral Dissertation of PR China and Singapore NRF (CREATE E2S2).* *Corresponding author: Haibo Chen, haibochen@sjtu.edu.cn.*

1. Intel uses MESIF protocol that adds a "forward" state. It should be straightforward to apply PTM to that one.

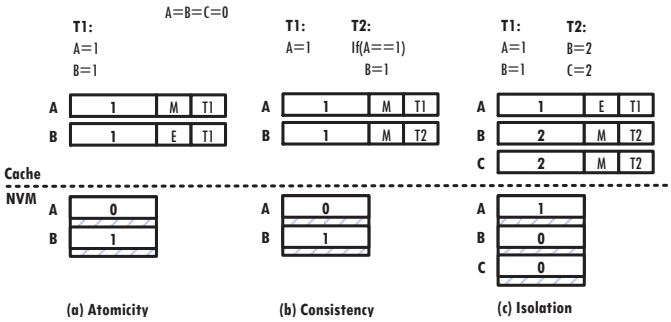


Fig. 1. Example violation of transaction semantics: each cache line has a cache status (e.g., M for modified, E for exclusive) and transactional ID (e.g., T1 and T2).

**Consistency:** The order of cache line eviction may also be different from the transaction commit order, which may break isolation of multiple transactions. As shown in figure 1-(b), if transaction 1 (T1) updates A from 0 to 1 and successfully commits. Transaction 2 (T2) checks A and update B from 0 to 1. However, only the cache line with B is evicted to NVM. As a result, the program states may be inconsistent (A = 0, B = 1) after recovery from a crash.

**Isolation:** Committed data can be updated before being persistent. When a transaction commits, the updated data is still buffered in the CPU cache and not persistent. However, the dirty cache line can be modified by a transactional update. As shown in figure 1-(C), if T1 updates two variables A and B, then a subsequent transaction T2 may update variable B and C. During a crash, as the value of B updated by T1 is lost, this violates the isolation property of transactional memory.

### 2.2 Related Work

Providing proper hardware/software interface between persistent memory and volatile CPU structures has been intensively studied recently [2], [3], [4], [10]. For example, Kiln [10] uses a non-volatile last-level cache (LLC) and NVM-aware cache replacement policy to ensure atomicity and ordering of memory updates. WRaP [4] uses a victim persistence cache to coalesce updates to NVM and a redo log in memory. In contrast, PTM mostly retains existing cache hierarchy structures (e.g., no non-volatile or specialized caches). BPFS [3] uses epochs delimited by write barriers to ensure ordering among epochs. In contrast, PTM relies on a scoreboard for dependence tracking among transactions, which absorbs updates from different transactions in cache. Hence, BPFS may incur more cache line flushes as updates to cache lines in an old epoch will flush all cache lines of older epochs. More importantly, PTM is designed to couple with TM to add persistence support, while other proposals use decoupled designs that require additional hardware or software mechanism to preserve consistency and/or isolation.

## 3 PTM Design

PTM is designed based on CMP architecture. It currently assumes a three-level on-chip cache and the MESI protocol.

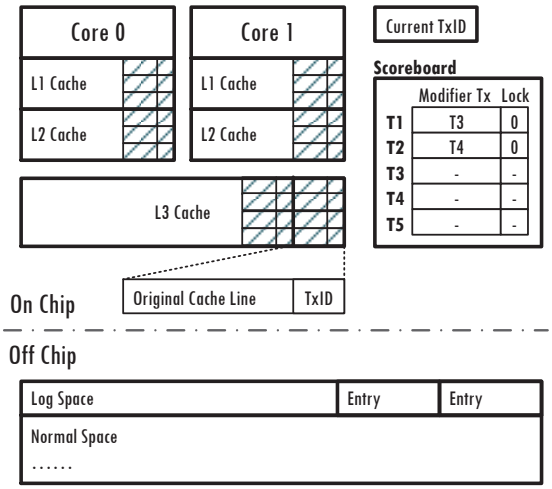


Fig. 2. Design of PTM

### 3.1 Overall Architecture

PTM guarantees a consistent state after recovery from a machine crash. Inspired from designs in file system and database, PTM uses logging to provide atomic writes of multiple cache lines to non-volatile memory. To track non-persistent transactions in cache, PTM uses a scoreboard to track dependency among them, as shown in Figure 2. To uniquely identify a transaction, PTM uses a global transaction ID register to generate transaction ID for each transaction. The following details the components of PTM:

**Transaction ID (TxID).** PTM uses a 8-bits register to assign the next assigned TxID. When a transaction commits successfully, it atomically gets a TxID from the register and increases the register by one. The TxID can be used to identify the timeline of transaction execution.

**Cache Line.** Like Intel's HTM implementation, the working set of a transaction is tracked at the cache line granularity. PTM adds a TxID field to each cache line to track the committed transaction that modified the cache line.

**Scoreboard.** PTM uses a scoreboard of 264 entries to track the dependency among transactions. It essentially records that a committed cache line of a transaction is dirtied by a subsequent transaction (called *modifier TX*). The $k$th entry of this board records the TxID of last *modifier TX* that dirties a cache line belonged to transaction $k$. Currently, it is implemented in a centralized location of CMP for simplicity. As it will only be updated when evicting dirty cache lines to L2/L3 caches, the access latency can be overlapped with the eviction process.

**Undo Log.** All cache lines evicted to NVM will be recoded in the log space in NVM at first. PTM uses "undo log" instead of "redo log" to avoid interfering with normal memory reads. This because the newest data may locate at the log space under "redo log" and hence other concurrent memory reads also need to first search the log space before directly read the NVM. In contrast, the newest data always locates at the NVM and thus other concurrent reads can directly operate on NVM when using "undo log".

Before flushing multiple cache lines to NVM, PTM first logs the original data in such cache lines from NVM to the log space. Then, PTM can write these cache lines to their home locations. The log space only buffers the most recent flushed cache lines and thus a small size is enough.

It can either be the normal NVM or a special-purpose static memory for the sake of life cycle.

### 3.2 Maintaining PTM States

PTM mainly interacts with cache-related operations. In the followings, we describe how PTM maintains corresponding states during cache read, update and eviction.

#### 3.2.1 Cache Read

A cache line read by a transaction may be with or without a transaction ID (TxID). Here we discuss how to handle these two cases for a transaction read.

**Read a cache line with a TxID.** This means that the transaction depends on a predecessor transaction, because it reads a cache line modified by the former transaction. Thus, if this transaction's update becomes persistent, all committed transactions whose states have been observed by this transaction should be persistent as well. To provide this guarantee, when a transaction's update is flushed into NVM, any updates of all earlier transactions (with smaller TxIDs) are also flushed into NVM.

**Read a cache line without a TxID.** This happens when the cache line was modified by a normal instruction or loaded into cache on a cache miss. In PTM, if a transaction reads a cache line without a TxID, when it commits, it will tag the cache line with its own TxID. If the cache line is in shared state, PTM will broadcast an invalidation message and set its state to "exclusive". Any concurrent or successor transactions reading this cache line will inherit this TxID for this cache line. When any cache line of this transaction is evicted to NVM, all cache lines tagged with the same TxID in "modified" state will be flushed to NVM. Thus, dirty cache lines depended by this transaction will also be persistent.

#### 3.2.2 Cache Update

A cache line can be updated by either a transactional write or a normal write. PTM only needs to maintain cache states related to transactional states. If a normal write tries to update a cache line which is already tagged with a TxID, PTM will assign a TxID to this instruction by treating this instruction as a simple transaction that never aborts. In the followings, we will show how PTM updates the cache line states and the scoreboard during a transactional write.

**Update the TxID field**. If a transaction issues a write, the corresponding cache line is changed to "modified" and the TxID field is set to be a special value (TX-PENDING). If the transaction commits, the TxID fields of all modified cache lines are changed from TX-PENDING to a newly assigned transaction ID. If a transaction tries to update a cache line that is already in "modified" state, this cache line needs to be evicted to L2 cache before being written. As a result, L2 cache always maintain the old version of the modified cache lines. In case of transaction aborts, PTM can simply roll back by invalidating the modified cache lines in L1 cache and clearing the TxID fields.

**Update scoreboard**. When a cache line is to be evicted from L1 cache to L2/L3 caches, PTM will check the state of the cache line to be evicted (evicting cache line) and the cache line to replaced (victim cache line) in L2/L3 cache. If both of them are "modified" and tagged with TxIDs, this means that both of them belong to non-persistent commits of some transactions. Hence, there may be a dependency between the two transactions as they update the same data. In this case, PTM will read the entry in the scoreboard for the *owner TX* of the victim cache line, and check if the TxID of the evicting cache line is larger than the modifier TX's TxID of the victim cache line. If so, PTM will replace the entry with the TxID of the evicting cache line. Otherwise the original modifier TX's TxID will remain in the entry.

#### 3.2.3 Cache Eviction to NVM

Cache lines may be evicted due to cache set conflict or transition from "modified" to other states. If the evicted cache lines are modified by a transaction, to guarantee transactional semantics in NVM, cache lines are modified by the following three kinds of transactions should also be flushed into NVM: 1). *owner TX* of the evicted cache lines; 2). transactions committed before the *owner TX*; 3). *modifier TXs* of the transaction to be persistent. PTM calculate a closure that includes all three kinds of transaction by scanning the scoreboard. A top pointer is initialized to the ID of the transaction that owns the cache line to be evicted. Then, from the beginning of the scoreboard, PTM scans the scoreboard to check whether TxID stored in each entry is larger than the top pointer. If so, the top pointer will be updated to that TxID in that entry. The scan stops when the index of the entry to be checked equals to the top pointer. After scanning, the top pointer contains the largest TxID of the minimal closure. PTM will then flush all the cache lines tagged with TxIDs less than the largest TxID. As the underlying cache flushing can be streamed, the latency of this bulk flush should not be very long. Moreover, CPU would not be stalled during flushing if it did not access the cache bank that is in the middle of a flush operation.

Note that this scheme will likely cause more cache lines being flushed. Another approach would be precisely tracking the dependency of transactions, instead of flushing transactions earlier than largest TxID. This, however, would require more complex hardware, which will be our future work.

**Atomic flushes:** During scanning the score board, new TXs may also require updating the scoreboard, which may lead to races and thus imprecise tracking. To this end, PTM adds a lock bit to each scoreboard entry and each TX needs to acquire the lock bit before updating the scoreboard. When scanning scoreboard for cache flushing, PTM will lock all scanned entries in the scoreboard and stalled TXs that requires updating the scoreboard. After cache flushes, PTM then unlocks the lock bits originally locked for scanning.

#### 3.2.4 A Running Example

Here we use a simple example to illustrate the process of cache operations. As shown in figure 3, transactions update variables in different cache lines and execute on a single core sequentially. The figure shows the executed transactions and the corresponding cache line state after transaction execution. After the first transaction has executed, one cache line (A) in L1 cache is dirtied and tagged with T3 as the TxID. Then T4 updates both A and B, the cache line (A) dirtied by previous transaction (T3) needs to be evicted to L2 cache. Meanwhile both cache lines (A and B) in L1 cache are set to be "modified" and tagged with T4. Then T5 will update all A, B and C. Before updating operations, cache lines holding A and B updated by T4 should be evicted to

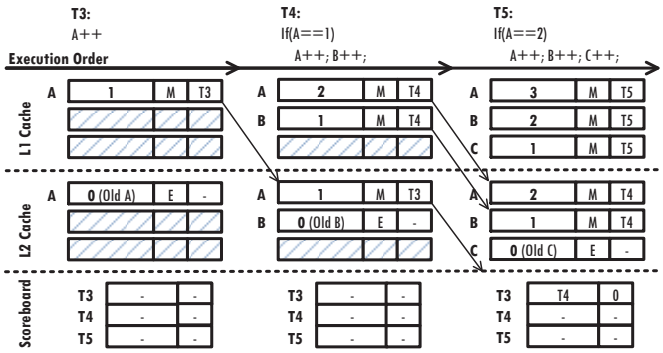


Fig. 3. A simple example

L2 cache. Due to a cache line tagged with T3 is already in in L2 cache and will replaced by the evicted cache line tagged with T4. We need to record T4 in T3's scoreboard entry.

### 3.3 Other Issues

**TxID overflow:** As PTM only uses 8-bit to identify a transaction, the TxID will overflow after 254 transactions' execution ( 0 for normal cache lines and TX-PENDING for pending TXs). To guarantee correctness, all cache lines in "modified" state and tagged with a transaction ID will be flushed into NVM during an overflow. Then the state is changed to "exclusive" and the transaction tag is cleared. It is straightforward to use more bits to identify transactions as well, which results in different tradeoffs between hardware overhead and cache lines flushed due to TxID overflow.

**Durability:** By default, PTM seamlessly follows the programming convention of TM by preserving ACI properties even during machine crashes. Like other designs in database and file systems, PTM also needs to trade off between freshness and performance: buffering more data in CPU cache may result in more data loss during machine crashes, while frequently evicting data may hurt performance. PTM further provides a special instruction called "dsync" to force all transactions before "dsync" has been durable, similarly to its "sync" counterpart in file systems. This enables controlling when transactions should be persistent, instead of relying on PTM to eventually persist them.

**Hardware Cost:** The major costs of PTM to CPU are a 8-bit TxID register, 8-bit extension to each cache line and a 254-entry scoreboard, which sums up as (1 + #cache lines + 254 (scoreboard entry) + 32 (lock bits)) bytes. Other cost include the hardware logics, whose cost is mostly fixed. The off chip cost include the log space (currently 16 Mbytes) as well the hardware logics to support "undo log".

## 4 Preliminary Evaluation

We built a multi-level cache simulator with the MESI protocol based on Pin to implement PTM . The processor is configured with 4 cores, each core has a 32 KB L1 cache and a 256 KB L2 cache. All cores share a 8 MB L3 cache, which are the typical configuration of the current Intel Haswell processor that supports HTM. As we currently have no cycle accurate simulator with HTM support on hand, we only collect the number of cache lines evicted to the NVM.

We compare PTM with the write back policy without any consistency guarantee, as well as a commit-through policy, in which all cache lines updated by a transaction will be flushed to the NVM on commit. For all three cases, we assume that NVM is used as the main memory. For evaluation, we test a concurrent $B^+$-tree (i.e., Masstree [7]) wrapped using HTM and the concurrent skip list using HTM from the LevelDB [5]. They are key data structures in database and key/value stores and thus demand consistency semantics during a crash. We evaluate their performance using Yahoo's YCSB benchmark with 20% put and 80% get.

Figure 4 and 5 shows the evaluation result, y-axis is the number of cache lines flushed to NVM during the execution, x-axis is the number of cores (worker threads). PTM adds around 5% and 12% more cache lines being flushed for Masstree and LevelDB accordingly. By contrast, flushing cache lines during transaction commit adds 3X more cache lines being flushed.

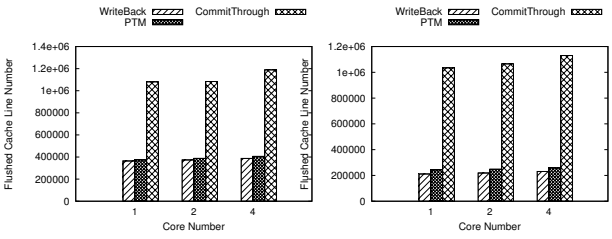


Fig. 4. *LevelDB (skiplist)*




Fig. 5. *Masstree (B+tree)*

## 5 Summary and Future Work

This paper presented PTM, a new design that adds durability support to transaction memory by combining it with NVM. By extending HTM design with simple hardware component, PTM ensures that the ACI properties are preserved even under machine crashes. Preliminary evaluation shows that PTM incurs small amount of additional cache line flushes. We plan to further explore, extend and evaluate different design choices of PTM in future.

## References

[1] L. Chua, "Memristor-the missing circuit element," *Circuit Theory, IEEE Transactions on*, vol. 18, no. 5, pp. 507–519, 1971.

[2] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS*, 2011, pp. 105–118.

[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.

[4] E. Giles, K. Doshi, and P. Varman, "Bridging the programming gap between persistent and volatile memory using wrap," in *Conf. on Computing Frontiers*, 2013.

[5] Google Inc., "Leveldb: A fast and lightweight key/value database library by google," http://code.google.com/p/leveldb/, 2013.

[6] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993.

[7] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys*, 2012.

[8] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *ISCA*, 1984.

[9] H. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proc. of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[10] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: closing the performance gap between systems with and without persistence support," in *MICRO*, 2013.

## APPENDIX

Again, we thank the editor and reviewers for their insightful comments and suggestions, which are very helpful to improve this paper. The followings list how we address the main issues as well as responses to address the concerns of each reviewer:

### .1 Main issues

- How your approach compares to an epoch-based implementation: addressed this in response 1 of review 1.
- How do you ensure atomicity of flush operations (or why it does not matter): addressed this in response 1 of review 2.
- What are the baseline latencies and overheads assumed in the comparison: addressed this in response 2 of review 3.

### .2 Responses

#### *.2.1 Review 1*

1. The major concern at this point is the similarity to the epoch mechanism previously proposed by the BPFS work from MSR. ?

In section 2.2, we further clarified that the issues with epoch-based mechanism of BPFS. In epoch-based mechanism of BPFS, any update to a single cache line dirtied from a prior epoch will cause flushes of all cache lines owned by old epoch. This may not be an issue for a file system, as it is relatively infrequent that accesses overlaps for file updates. However, this will likely be an issue as transactional regions will like access the same memory location. Applying such mechanism will frequently flush cache lines to NVM and may even degenerate to the "commit-through" case.

2.At the least, mentioning how often the undo log is used for the workloads would give a sense of whether this is likely to be an issue?

XXX

3. A few writing nits

Thanks for pointing it out. We have changed them accordingly and have done a through proof-reading.

#### *.2.2 Review 2*

1. How to ensure the atomicity in a particular *flush* operation?

Thanks for clarifying the question and sorry for the misunderstanding. Yes, there could be races between flushing caches and transactions updating scoreboard. To address this issue, we add a lock bit for each entry of the scoreboard and requires locking the bit before updating. During flushing, all scanned scoreboard entries are also locked. We added a paragraph in section 3.2.3 to illustrate this.

#### *.2.3 Review 3*

1. I am still not comfortable with the bulk updates in the cache. For instance, haswell does not use things like Txid in the cache. It is a best-effort Tx system. It appears to me at first glance, that committing of lines is predicated on Txid bit. How does one go about this? It seems like the associated logic of if (tx-id == id) then commit cache line for the entire cache is hard? Please add a description.

2. What are the latencies you use for cache and memory and NVM. Does the baseline assume DRAM? this has critical effect on what slowdown you may see by requiring flushes to the NVM.

The baseline assume NVM since we assume that future computer may likely have NVM by default. We added a sentence in section 4 to clarify this.

3. Seems like the benefit of PTM is dependent on the overheads of dsync? Can you please quantify this?

Dsync gives programmers a tradeoff between freshness and performance. Due to space constraint, we didn't include the sensitivity evaluation of dsync calls and will do this in a full-fledged version.