 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 protocol1, 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.

1. Intel uses MESIF protocol that adds a “forward” state. It should be straightforward to apply PTM to that one.
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. WRP [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.
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 transactions 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
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.

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

APPENDIX

Again, we thank the editor and reviewers for their insightful comments and suggestions, which are very helpful to improve this paper. The following lists 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.