Hardware Transactions in Nonvolatile Memory

Hillel Avni[1], Eliezer Levy[1(✉)], and Avi Mendelson[2]

[1] Huawei Technologies, European Research Center
{hillel.avni,eliezer.levy}@huawei.com
[2] Technion CS & EE Departments, Haifa, Israel
avi.mendelson@technion.ac.il

Abstract. Hardware transactional memory (HTM) implementations already provide a transactional abstraction at HW speed in multi-core systems. The imminent availability of mature byte-addressable, nonvolatile memory (NVM) will provide persistence at the speed of accessing main memory. This paper presents the notion of persistent HTM (PHTM), which combines HTM and NVM and features hardware-assisted, lock-free, full ACID transactions. For atomicity and isolation, PHTM is based on the current implementations of HTM. For durability, PHTM adds the algorithmic and minimal HW enhancements needed due to the incorporation of NVM. The paper compares the performance of an implementation of PHTM (that emulates NVM aspects) with other schemes that are based on HTM and STM. The results clearly indicate the advantage of PHTM in reads, as they are served directly from the cache without locking or versioning. In particular, PHTM is an order of magnitude faster than the best persistent STM on read-dominant workloads.

1 Introduction

In [10], Herlihy and Moss defined hardware transactional memory (HTM), as a way to leverage hardware cache coherency to execute atomic transactions in the cacheable shared memory of multicore chips. The basic idea was that each transaction is isolated in the local L1 cache of the core that executes it. The semantics of atomicity were borrowed from the database research [4]. However, database transactions, unlike HTM transactions, are persistent, i.e. once a transaction committed successfully, it is also backed up to stable storage.

As HTM lingered, much research was done on software transactional memory (STM) in order to obtain low overhead and scalable synchronization among memory transactions without hardware assistance. Intel's first HTM implementation reached the market in 2013. At the same time, STM was incorporated into the GCC compiler [3,12].

Latest developments in memory technology (such as phase change memory, STT-RAM, and memristors) introduce the possibility of NVM devices that are fast and byte-addressable as DRAM, more power-efficient than DRAM, yet non-volatile and cheap as HDD. This paper proposes to provide HTM transactions

Y. Moses (Ed.): DISC 2015, LNCS 9363, pp. 617–630, 2015.
DOI: 10.1007/978-3-662-48653-5_41

with fast persistent storage by using NVM instead of (or in addition to) DRAM while keeping the volatile cache hierarchy intact.

The remainder of this paper is organized as follows. We conclude the Introduction section by reviewing related work. Section 2 introduces the model and terminology of PHTM. Section 3 defines the PHTM implementation, and explains the correctness of PHTM. Section 4 evaluates PHTM performance, and Section 5 concludes and discusses future work.

1.1 Related Work

Coburn et al. [5] suggested NV-Heaps, a software transactional memory (STM) that works correctly with NVM. The basic idea follows DSTM [9], in which transactional objects are stored in NVM. A transactional object can be opened for writing, and then the STM transaction, T, copies it to an undo log, and locks it. T maintains a volatile read log and a non-volatile undo log for each transaction. If a system failure occurs, T is aborted and the undo log, which is persistent, is used to reverse the changes of T.

While NV-Heaps is object-based, the Mnemosyne STM [13], which was published at the same time, is word-based and is derived from TinySTM [8]. However, the ideas behind these algorithms, i.e. nonvolatile undo log and a volatile read log, are identical.

While correct and feasible, the software-based methods exhibit poor performance due to bookkeeping overhead and locking serialization. Thus, some database implementations [11,14] use HTM for synchronization. However, these databases still use HDDs for persistence.

PMFS [7] uses NVM for storage of a file system and [6] uses NVM for persistence in an OLTP database. PMFS does use HTM, but only for very specific purposes of managing file system metadata.

This paper introduces the combination of NVM and HTM for the purpose of transaction processing. Our contribution is the presentation of a concrete implementation that provides full ACID semantics to transactions. We emulate this implementation and compare its performance to other approaches using STM, HTM, and NVM.

2 Persistent HTM

The imminent availability of mature Non-Volatile RAM (NVM) technology is bound to disrupt the way transactional systems are built. NVM devices are fast and byte-addressable as DRAM, more power-efficient than DRAM, yet non-volatile and cheap as HDD. Therefore, NVRAM will obliterate the traditional multi-tier memory hierarchy that is fundamental to the durability guarantees of ACID transactions. In this foreseen situation, maintaining invalid states in main memory can render a system crash unrecoverable. Therefore, a fresh and careful approach is required in the design of persistence and recovery after a crash (Restart) schemes that would harness the benefits of NVM. In the sequel, we use the following terms:

- **HTM:** A synchronization mechanism that commits transactions atomically, and maintains isolation. Once an HTM transaction T commits, all its newly modified data is in volatile cache.
- **An HTM transaction T_k:** A transaction executed by a processing core P_k.
- **NVM:** nonvolatile, byte addressable, writable memory.
- **Restart:** The task of a restart is to bring the data to a consistent state, removing effects of uncommitted transactions and applying the missing effects of the committed ones.

The hardware model investigated here includes unlimited NVM, many cores and no disk. All NVM is cacheable and caches are volatile and coherent. The system includes limited size DRAM. Finally, we use the term *Persistent HTM (PHTM)* to refer to the notion of existing HTM realizations, with the minimal necessary hardware and software adjustments needed for the incorporation of NVM. The PHTM system includes software and hardware.

2.1 Problem Definition

A difficulty arises when NVM persistency meets the growing number of cores on modern hardware. On the one hand, as data is in NVM, it is unnecessary to allocate another persistent storage for it, which reduces persistency overhead. On the other hand, as an address is written, the new value must be exposed atomically with a new consistent and persistent state. One alternative for guaranteeing this atomicity is by means of locks. With the ever growing number of cores, locking will introduce bottlenecks. A method to achieve atomicity without locking is HTM, but HTM cannot access physical memory. The problem is to close the gap between HTM and NVM and allow an application to maintain consistent and persistent states in NVM without locking and without duplicating the data.

2.2 Data Store Flow

The state of a data item x (an addressable word), which is written by an HTM transaction T, is specified as follows (see Figure 1). Note that x may be cached in the volatile cache or reside only in the NVM (just like any other addressable word):

1. **Private / Shared:** *Private* means x is only in the L1 cache of one thread, and is not visible to other threads. When x is *Shared*, the cache coherency makes its new value visible.
2. **Persistent / Volatile:** *Persistent* means that the last write of x is in NVM, otherwise, the new value of x is *Volatile* in cache and will disappear on a power failure.
3. **Logged / Clear:** When x is *Logged*, a restart will recover x from nonvolatile log. If x is *Clear*, the restart will not touch x, since its log record has been finalized or its transaction aborted.

Notice that although Figure 1 illustrates the state machine of a single write in a PHTM transaction, the *Logged* state should be for the entire transaction. That is, turning all writes of a single transaction from clear to logged requires a single persistent write. In a PHTM commit, all writes are exposed by the HTM and are simultaneously logged. Each write must generate a persistent log record, but until a successful commit, the write is not logged in the sense that it will not be replayed by a restart process.

When the HTM transaction T_k writes to a variable x, x is marked as transactional in the L1 cache of P_k and is private, i.e., exclusively in the cache of P_k. It is volatile, as it is only in cache, and clear, i.e. not logged, as the transaction is not yet committed. Upon an abort or a power failure, the volatile private value of x will be discarded and it will revert to its previous shared and persistent value.

In the PHTM commit, the state of x changes twice. It becomes shared, i.e. visible and at the same time it is also logged. Both changes must happen atomically in a successful commit. After a successful commit, the PHTM flushes the new value of x transparently to NVM and clears x. If there is a system failure and restart when x is logged, the recovery process uses the log record of x to write the committed value of x and then clears x.

It is important to observe that the log in NVM is not a typical sequential log. Instead, it holds unordered log records only for transactions that are either in-flight or are committed and their log records have not been recycled yet.

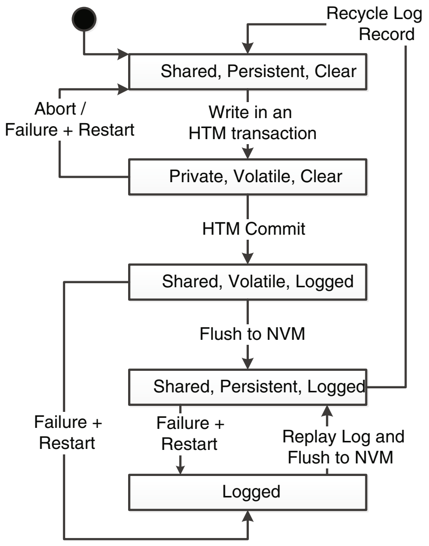

Recycle Log Record
Shared, Persistent, Clear
Abort / Failure + Restart
Write in an HTM transaction
Private, Volatile, Clear
HTM Commit
Shared, Volatile, Logged
Flush to NVM
Shared, Persistent, Logged
Failure + Restart
Failure + Restart
Replay Log and Flush to NVM
Logged

Fig. 1. State machine for a persistent transactional variable

3 PHTM Implementation

If the L1 cache was non-volatile, HTM would be persistent as is without any further modifications. A restart event could abort the in-flight transactions, and a committed HTM transaction, while in the cache, would be instantly persistent and not require any logging. However, due to hardware limitations, e.g. fast wear out and slow writes of NVM, the cache hierarchy will stay in volatile SRAM in the foreseeable future.

3.1 Hardware Ramifications

As shown in Figure 1, PHTM requires that the successful commit of the transaction T_k will atomically set the persistent commit record of T_k. The $tx\_end\_log(T_k)$ instruction is added for this purpose. This instruction performs an HTM commit and sets the commit record of T_k. Figure 2 shows the layout of the persistent footprint of T_k in NVM. It includes the `logged` indication which serves as the commit record of T_k. The $tx\_end\_log(T_k)$ writes the commit record in NVM, and in addition sets the status of the writes of T_k to *Logged*.

PHTM requires that log records are flushed from cache to NVM by a live transaction without aborting itself. We call this process the $Finalization$ of T. In T finalization, after $tx\_end\_log(T)$, flushing of the data written by T from cache to NVM must not abort ongoing concurrent transactions that read this value. Considering the HTM eager conflict resolution, these flushes must not generate any coherency request and for performance, they should not invalidate the data in the cache. Such operations are designated "transparent flushes" (TF) as they have no effect of the cache hierarchy and the HTM subsystem.

In summary, we define a new HW-related primitive called **Transparent flush (TF)** as follows: If α is a cached shared memory address, TF(α) will write α to physical shared memory, but will not invalidate it and will not affect cache coherency in any way. If T_k reads α transactionally, and then T_q, where possibly $k \neq q$ executes TF(α), T_k will not abort because of it. The ARM DC CVAC instruction to clean data cache by virtual address to point of coherency [1], and the cache line write back (CLWB) instruction from Intel future architecture [2] are examples in this direction.

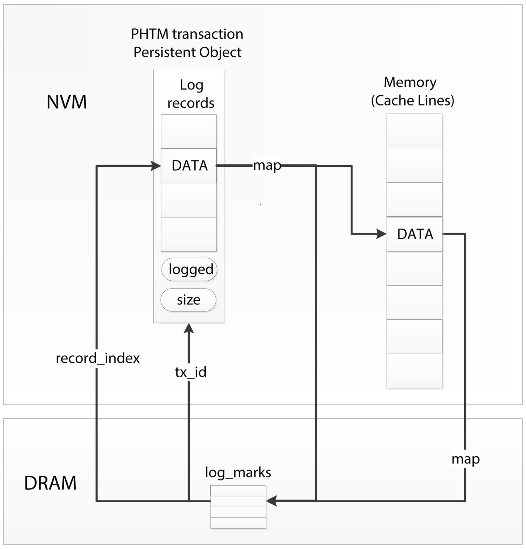


Fig. 2. PHTM System

3.2 Software Details

The API of PHTM is $tx\_start()$ and $tx\_end()$ as in a non-persistent, Intel HTM transaction. $tx\_start()$ is translated to starting an HTM transaction, while $tx\_end()$ is translated to flushing the transaction persistent structure, followed by a $tx\_end\_log(T)$ instruction, followed by flushing of the data itself. The machine store instructions are replaced by the preprocessor with the $tx\_write()$ function.

The log records and the size field that appear in the PHTM transaction persistent object (Figure 2) are flushed as part of the transaction, but **not** as part of the $tx\_end\_log(T_k)$ instruction. However, multiple writes to the same cache line will write to the same log record. Thus, as an optimization, the log records are flushed only once before commit to prevent multiple flushes of the same data.

In a system with NVM, it is assumed that the compiler will automatically replace the $tx\_end()$ with $tx\_finalization(T)$ (Algorithm 2), and the store instructions with tx_write (Algorithm 1). This type of preprocessing already exists for GCC STM support. In case a PHTM transaction gets aborted, all its writes (to volatile memory) are undone automatically, and the commit record is not set, so there is no overhead.

Fallback. The HTM follows a best effort policy, which means it does not supply a progress guarantee. As a result, after a certain number of aborts in the standard volatile HTM, the transaction must take a global lock and commit. However with NVM, a global lock is not enough as the tentative writes may have already contaminated memory. Therefore, an undo log entry must be created for every volatile HTM write, or a full redo log must be created before the first value is written to NVM. The first option was chosen to avoid read after write overhead.

Scalable Logging and Fast Recovery. With volatile cache and committed HTM transactions that accommodate all their writes in cache, it is necessary to log the writes in order to allow recovery in case a restart happened after HTM commit, when the writes were still volatile.

All writes to the log must reach non-volatile memory before an HTM commit, while all transactional writes stay in the cache. This implies that the log to NVM needs to be flushed without aborting the executing transaction. As the log is local, non-transactional stores to write the log records can be used, and later, a flush is used to write them to NVM. The flush should be a TF so the log stays in the cache and no transaction is aborted.

Logging must provide the restart process with the last committed value for each logged variable x. The two ways to do this with concurrent transactions is to attach a version to x or to verify that x is logged only once in the system. If the appearance of x in multiple logs is allowed, then the latest version of the log of x must be kept. Thus, freeing the log safely will require communication among the committed transactions, e.g. barriers. This communication is not scalable. On the other hand, not freeing the log will require unbounded memory and a longer time for recovery in restarts.

Instead, each address is allowed to appear at most in one log. To avoid instances of the same address in multiple logs, a volatile array of log marks is added in which each memory address is mapped to one mark. When a transaction is about to write x, it also marks x as logged. Until x is flushed, no other transaction can write it. The reason marks are used is to prevent a write to a variable that was already written by another transaction, but not yet flushed, so it is still logged. All other conflicts are handled directly by the HTM. The array of marks can be volatile, as in case of restart it is known that the logged addresses are unique, and that the restart and recovery process do not create any new log records. After restart, a new and empty array of marks can be allocated.

The writing of the marks is a part of a transaction, i.e. if T_k writes x, it also marks x and in committing, the writing and the marking will take effect simultaneously as they are both transactional writes, while at abort, they are both canceled. As long as the mark is set, the value of x, which appears in the log of T_k, cannot be changed. Therefore, after x is secured in NVM and cleared, the mark is unset. It is important to emphasize that the transactional load instructions ignore the marks and execute in full speed, which is a key advantage of PHTM as Reads are processed in hardware speed.

Write. It is assumed there is a map function that extracts the index of the mark from the address. The map not only performs mapping of every address to a unique mark, but also maps all the addresses in the same cache line to the same mark. As cache flushing is in cacheline units, the log records and the marks are maintained in cacheline granularity.

Algorithm 1. PHTM Write Instrumentation

```
1:  function TX_WRITE($addr, val, T$)
2:      $id \leftarrow T.self\_id$
3:      $m \leftarrow log\_marks[map(addr)]$
4:      ▷ $C_{addr}$ is the cache line of addr
5:      if $m.tx\_id = T.self\_id$ then
6:          ▷ T already accessed $C_{addr}$
7:          $rec\_index \leftarrow m.rec\_index$
8:      else
9:          if $m.tx\_id =$ null then
10:             ▷ $C_{addr}$ is not marked
11:             $m.tx\_id \leftarrow id$
12:             $rec\_index \leftarrow T.size$
13:             $m.rec\_index \leftarrow rec\_index$
14:             $T.size \leftarrow T.size + 1$
15:             $T.addr[rec\_index] \leftarrow addr$
16:             $T.data[rec\_index] \leftarrow C_{addr}$
17:         else
18:             ▷ $C_{addr}$ is marked
19:             $\_xabort(MARKED)$
20:         end if
21:     end if
22:     ▷ log the writing
23:     $T.data[rec\_index][offset] = val$
24:     ▷ Perform the actual writing
25:     $addr \leftarrow val$
26: end function
```

Algorithm 1 shows the implementation of *tx_write*. It starts by locating the mark of the address (Line 3). If the mark was already marked by T (Line 6), T extracts the *record_index* from the mark (Line 7). Otherwise, if the mark is free (Line 10), T sets the mark to point to itself (Line 11). Next, T allocates a log entry (Lines 12 - 14), which is in the index currently pointed by the size field. T stores the index in the mark so later writes to the same line by T will use this index and increment the size field. Next, T stores the address in the log (Line 15) and since this is the first access to this cacheline, T stores the original content of this cacheline (Line 16). The restart will write full cache lines, and thus the original values of the unwritten parts of the lines need to be kept.

If the address is currently marked by another transaction, T explicitly aborts with the code MARKED (Line 19). When the abort handler sees the reason for the abort was MARKED, it will not count this as a conflict and will not fallback to locking. After acquiring the log mark, the value is written into the private, logged copy of the cache line (Line 24), and then it is written to actual memory (Line 25). Writing the marks is done in transactional mode, so they are added to the transaction size, and may cause size violation and aborts that do not occur in the standard HTM.

Finalization. In Algorithm 2, the code for $tx\_finalize$ is presented. This code replaces the HTM *_xend* instruction to commit persistent HTM transactions. Before committing the HTM transaction, the log, including its size (Line 3), and data (Lines 6 and 5) are flushed to NVM using TF. Then the PHTM commits using the new tx_end_log, and if the commit was successful it simultaneously sets the logged indication of T (Line 11). Next, all the cache lines that include data that was written during the transaction are flushed to NVM (Line 13). After flushing, the data is persistent, so the log is cleared by writing zero to the log indication and flushing it to NVM (Line 17). Only after clearing the log can the marks (line 22) be freed and the size of the log (Line 20) be reset.

Algorithm 2. PHTM Finalization

```
 1: function TX_FINALIZE($T$)
 2:         ▷ Transparently flush the log
 3:     $sz \leftarrow T.size$
 4:     for all $s < sz$ do
 5:         $TF(T.addr[s])$
 6:         $TF(T.data[s])$
 7:     end for
 8:     $TF(T.size)$
 9:         ▷ HTM commit and log
10:     $tx\_end\_log(T.logged)$
11:         ▷ Transparently flush data
12:     for all $s < sz$ do
13:         $TF(T.addr[s])$
14:     end for
15:         ▷ Clear log with regular flushes
16:     $T.logged \leftarrow 0$
17:     $Flush(T.logged)$
18:
19:     $T.size \leftarrow 0$         ▷ Clear the marks
20:     $Flush(T.size)$
21:     for all $s < sz$ do
22:         $marks[map(addr[s])] \leftarrow$ null
23:     end for
24: end function
```

3.3 Correctness and Liveness

The correctness and liveness of PHTM are derived from HTM with the necessary adjustments.

Correctness. When power is not interrupted the PHTM is operating exactly as HTM with the addition of each transaction maintaining private information about its writes and success. It is left to show that after a power failure in any point of the execution, the last committed write of an address is in NVM. If the last committed write is not logged than in Line 13 of Algorithm 2 it was already flushed to NVM. If it is is logged than it was marked in Line 3 of Algorithm 1 and it is the only logged write to the address, so the recovery process will set it in NVM. If the last write is not successfully committed then it is only in the local volatile cache of the transaction executor and it will vanish at power down.

Liveness. As a PHTM transaction includes an HTM transaction and HTM has no progress guarantees, PHTM has no progress guarantees either. However, an HTM transaction cannot delay another concurrent HTM transaction, while if a PHTM transaction stops before clearing its marks in Line 22 of Algorithm 2 it can stop a concurrent transaction from writing the marked address. The mark, which is a writer-writer lock, is set only from commit to the end the data flush. If a transaction T_1 committed but swapped out before clearing its write-set, it can lockout a concurrent writer T_2. To resolve this situation T_2 may clear T_1, as the data is already shared. This help involves communication among transactions so it is technically complicated and breaks the isolation among transactions. It may also require that the synchronization primitives used will be persistent.

4 Evaluation

In this section the performance of PHTM is evaluated using an RB-Tree data structure and a synthetic benchmark. The synthetic benchmark checks the overheads of PHTM and how it interacts with the size limitation of the HTM.

These tests were executed on an Intel Core i7-4770 3.4 GHz Haswell processor with 4 cores, each with 2 hyper threads. Each core has private L1 and L2 caches, whose sizes are 32 KB and 256 KB respectively. There is also an 8 MB L3 cache shared by all cores. Section 4.1 describes how the PHTM hardware was emulated, i.e. the NVM and the *tx_end_log* instruction. Section 4.2 explains how a persistent STM for a fair comparison was emulated and in Section 4.3 some preliminary performance results are presented.

4.1 Hardware Emulation

Intels Haswell processors feature an HTM facility that is used the experiments. However, NVM and PHTM are still not realized in hardware so they are emulated for evaluation. We emulate the effects of power failure in PHTM by leaving the power on and zeroing only the volatile regions, i.e. the log marks. To emulate the *tx_end_log*, the commit record is written during the transaction. As this is part of the transaction, the HTM itself makes the commit record visible simultaneously with a successful commit. In TF emulation, only the NVM access time is emulated by inserting a 100 nanosecond delay according to the expected NVM performance. The interconnect traffic is not emulated, as this traffic is identical to the interconnect traffic in the solution compared to, i.e., the persistent STM.

4.2 Compared Algorithms

PHTM. is compared with standard **HTM**, with an emulation of a persistent software transactional memory (**PSTM**), and with standard **STM** without persistence. The STM is from GCC [3,12], but in the lowest optimization level. In order to provide a fair comparison of all 4 algorithms, compiler optimization, which can reduce the number of accesses, is avoided.

Analagous to PHTM, PSTM implements a redo log in persistent memory. The overhead of writes in PSTM is one flush of the log entry before commit and one flush of the data after commit, so it is comparable to PHTM. PSTM is based on Mnemosyne [13], but with few adjustments. A redo log and in-place writing was chosen to be implemented in order to avoid huge read after write penalties. PHTMs advantage over STM concerns the processor speed loads. For a fair comparison, the PSTM loads should be as fast as possible in order to challenge the PHTM.

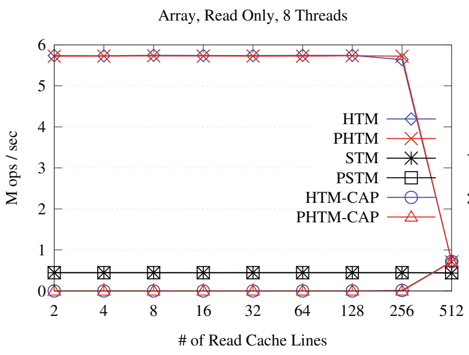
The shortcomings of PSTM concern its STM nature, i.e. the overhead associated with instructions instrumentation, locking and versioning. The problems of PHTM concern its HTM origin, i.e. limited transaction size and sensitivity to contention.

4.3 Benchmarks

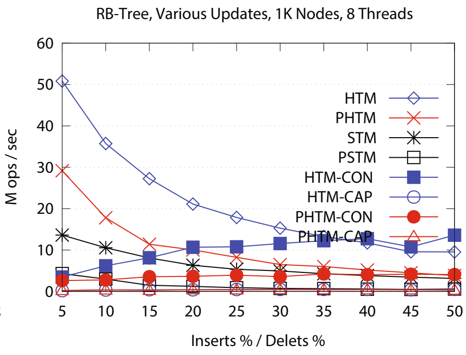
PHTM performance is evaluated without contention on a synthetic array benchmark. It is then tested on an RB-Tree to see its performance under contention. The algorithms checked include **HTM**, **PHTM**, **STM** and **PSTM**. In the graphs, the **HTM-CAP** and **PHTM-CAP** lines are added to count the number of HTM capacity aborts and the **HTM-CON** and **PHTM-CON** lines are added to count the number of conflict aborts in some of the graphs. Aborts are counted in operations per second that where aborted. A conflict abort is retried 20 times before taking a global lock, but a capacity abort is not retried and locks immediately, as a retry has low chance for success.

Array Workloads. In this workload all transactions have the same number of accesses in order to make their execution time comparable. The tests access a set of consecutive memory addresses, so the cache is filled with no fragmentation. Each accessed address is in a separate cache line.

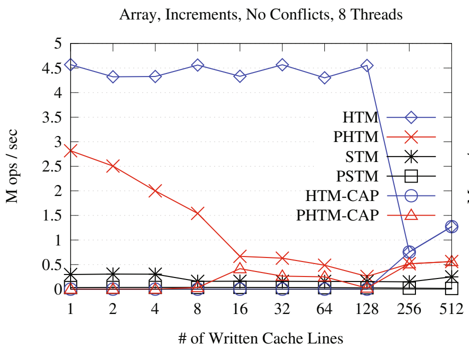
Read-Only: First, the performance of PHTM on a read-only workload is observed. This is the best case for PHTM, as a PHTM load is in processor speed. In Figure 3a, every transaction performs 512 `load` instructions cyclically to various numbers of consecutive cache lines. It can be seen that as long as HTM capacity limit is avoided, PHTM and HTM perform the same and outperform STM and PSTM by an order of magnitude. The tests in Figure 3a execute on all



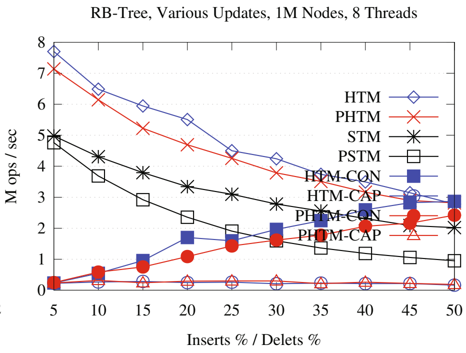
(a) Array - Read only



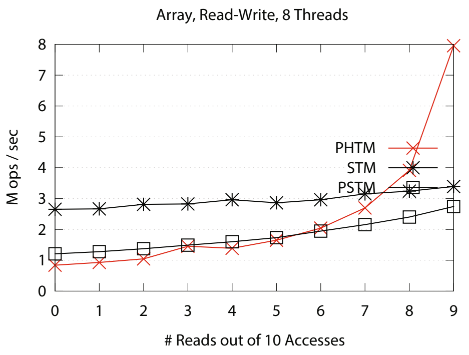
(b) RB - 1K Nodes



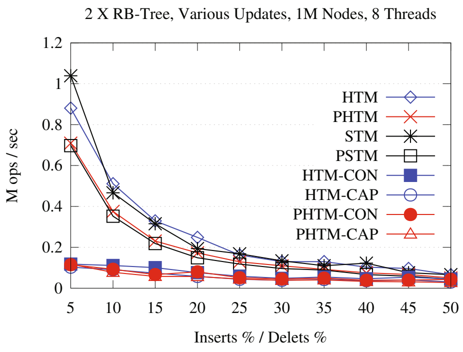
(c) Array - Write Only



(d) RB - 100K Nodes



(e) Array - Read and Write Mix



(f) RB - 1M Nodes

Fig. 3. Synthetic array and a red-black tree benchmarks

8 hardware threads. There is hyper threading and the cache size of each thread is half as that which is on a core, i.e. 16KB or 256 cache lines of 64 bytes, so when the access set size is 512 all HTM and PHTM transactions violate capacity limitation and abort. At this point, HTM and PHTM performance becomes equal to STM as it uses the fallback. As expected STM and PSTM perform equivalently.

Write-Only: In this test, the performance of PHTM writes is examined. The benchmark in Figure 3c is very similar to the one in Figure 3a. Every transaction performs 512 `store` instructions cyclically to various numbers of consecutive cache lines.

It is observed that as the number of accessed cache lines increases, the HTM performance is not affected but the PHTM approaches the performance of PSTM because the flushes to NVM dominate the performance. It must be emphasized that the PHTM and PSTM only flush a cache line once in a transaction even if they write the line multiple times. Therefore if the transaction accesses only one cache line, it will write the same cache line 512 times but flush only twice - one flush for the data after commit, and one for the log before commit. This overhead can be mitigated if the TF is made non-blocking.

When the HTM and PHTM transactions start reaching the capacity limitation, they execute with a global lock which dramatically reduces their performance and prevents scaling. PHTM reaches the capacity limitation well before standard HTM. This is because PHTM writes a log entry for every write, so it accesses a larger amount of memory. In addition, the log entries are not in continuous memory, so they can violate the cache associativity even before the cache is full. Note that in a real implementation non-transactional `store` instructions could be used for the log and avoid an increase the transactional footprint.

Read-Write Mix: This test checks how the proportion of transactional `load` instruction vs. the number of transactional `store` instructions affects the performance of PHTM compared to STM and PSTM when the capacity aborts issue is eliminated, i.e. in small transactions.

In Figure 3e, every transaction performs 10 accesses of 10 separated cache lines with various number of writes. HTM is not shown in because it performs a read and a write approximately in the same speed. Figure 3e illustrates that until the read-only part reaches 80%, STM is faster than PHTM. However, when the portion of read-only instructions reaches 90%, PHTM is already double the performance of STM. As seen in Figure 3a, when the portion reaches 100%, PHTM is 12 times faster than STM.

RB-Tree Workloads. To evaluate the PHTM in the face of contention, an RB-Tree benchmark is used. All transactions access a tree with random keys to insert, delete or lookup. The workloads executed run on 8 cores with a fixed keys-range size and vary the amount of updates from 10% to 100%. Each tree starts half full and the number of inserts equals the number of deletes to preserve the tree size.

The first set of tests is performed on a small 1K nodes tree. As seen in Figure 3b there are no capacity aborts in HTM and PHTM. Conflict aborts rise but the scalability is comparable to STM. In this test PHTM is about 6 times faster than PSTM. The second set of tests in Figure 3d is executed on a 1M tree, and we can see that capacity aborts are visible, but low. As a result, PHTM is only 40% faster than PSTM. Still the scalability is the same even though conflict aborts are high, which suggests STM also experiences similar aborts rate. To further increase the capacity challenge, we execute transactions that do the same operation on two 1M trees atomically. As seen in Figure 3f, the capacity aborts rises and PHTM performance drops to PSTM.

As expected, capacity limitation is the worst problem of HTM as it forces transactions to serialize, and the big obstacle to PHTM performance is capacity aborts which it inherits from HTM. In each of the performed tests PHTM keeps a constant difference from HTM (and PSTM from STM) throughout the contention levels. The difference is due to the portion of writes in the workload. The smaller the tree, less time is spent on traversing it, so the relative part of writing grows, and the persistent algorithm overhead increases.

5 Conclusion

Future generations of systems are expected to accommodate thousands of cores and petabytes of NVM. Lock based synchronization, as well as traditional log-based persistence will introduce unacceptable overhead in those systems. PHTM is a first step towards ACID transactions that avoid locking and provide persistence in a way that is specialized to NVM. Preliminary experiments show PHTM is 12x faster than its persistent STM counterpart on read-dominant workloads. When contention exists, it is still 6x faster, and when the capacity limit is hit, PHTM falls back to a software-based approach and then its performance equals the performance persistent STM. These performance advantages stem from the fact that PHTM avoids all locking in the reader path and its commit is instantaneously visible. PHTM presents a concept that should serve as a blueprint for possible realizations that would evolve with the availability of mature NVM technology. Once the HW model we used will be instantiated as a concrete system, further optimizations might be needed. Moreover, we expect that the current limitations of HTM technology will be alleviated and then the applicability of the PHTM scheme can be extended.

References

1. ARM architecture reference manual for ARMv8-a architecture profile. https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet6/DDI0487A_e_armv8_arm.pdf
2. Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf
3. Tm support in the gnu compiler collection. http://gcc.gnu.org/wiki/TransactionalMemory

4. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co. Inc., Boston (1987)
5. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: Proceedings of the Sixteenth International Conference, ASPLOS XVI, pp. 105–118. ACM, New York (2011)
6. DeBrabant, J., Arulraj, J., Pavlo, A., Stonebraker, M., Zdonik, S., Dulloor, S.: A prolegomenon on oltp database systems for non-volatile memory. In: ADMS@VLDB (2014)
7. Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014, pp. 15:1–15:15. ACM, New York (2014)
8. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246. ACM (2008)
9. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, PODC 2003, pp. 92–101. ACM, New York (2003)
10. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA 1993, pp. 289–300. ACM, New York (1993)
11. Leis, V., Kemper, A., Neumann, T.: Exploiting hardware transactional memory in main-memory databases. In: IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31–April 4, pp. 580–591 (2014)
12. Riegel, T.: Software Transactional Memory Building Blocks. Ph.D. thesis, Technische Universität Dresden, Dresden, 01062 Dresden, Germany (2013)
13. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: Lightweight persistent memory. SIGPLAN Not. **47**(4), 91–104 (2011)
14. Wang, Z., Qian, H., Li, J., Chen, H.: Using restricted transactional memory to build a scalable in-memory database. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014, pp. 26:1–26:15. ACM, New York (2014)