Journal of Parallel and Distributed Computing 130 (2019) 63–79

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Hardware Transactional Memory meets memory persistency☆

Daniel Castro *, Paolo Romano, João Barreto

INESC-ID & Instituto Superior Técnico, University of Lisbon, Portugal

HIGHLIGHTS

- Existing HTM systems do not ensure durability when used over persistent memory.
- The limitations of current HTM prevent from using conventional logging schemes.
- We tackle this challenge via a new HTM-aware software-based logging scheme.
- Usage of physical clocks and OS-level copy on write mechanisms.
- Up to 10x gains in terms of throughput and NVM cache line flushes.

ARTICLE INFO

Article history:
Received 11 July 2018
Received in revised form 21 January 2019
Accepted 15 March 2019
Available online 4 April 2019

Keywords:
Transaction
Memory
Persistent
Hardware
System

ABSTRACT

Persistent Memory (PM) and Hardware Transactional Memory (HTM) are two recent architectural developments whose joint usage promises to drastically accelerate the performance of concurrent, data-intensive applications. Unfortunately, combining these two mechanisms using existing architectural supports is far from being trivial. This paper presents NV-HTM, a system that allows the execution of transactions over PM using unmodified commodity HTM implementations. NV-HTM exploits a hardware–software co-design technique, which is based on three key ideas: (i) relying on software to persist transactional modifications after they have been committed via HTM; (ii) postponing the externalization of commit events to applications until it is ensured, via software, that any data version produced and observed by committed transactions is first logged in PM; (ii) pruning the commit logs via checkpointing schemes that not only bound the log space and recovery time, but also implement wear leveling techniques to enhance PM's endurance. By means of an extensive experimental evaluation, we show that NV-HTM can achieve up to 10× speed-ups and up to 11.6× reduced flush operations with respect to state of the art solutions, which, unlike NV-HTM, require custom modifications to existing HTM systems.

1. Introduction

Over the last years, several technological advancements have shaken the ground of memory technology. Hardware Transactional Memory (HTM) and Persistent Memory (PM) are, arguably, two of the most promising ones.

On the one hand, HTM [6,25,28], allows for speculatively parallelizing the execution of critical sections/transactions, by delegating to efficient hardware mechanisms the detection of conflicts arising among concurrently executing threads. HTM has been shown to bring about striking performance gains, especially when used to synchronize access to small in-memory data structures [18], as well as to significantly simplify the development of concurrent applications with respect to conventional lock-based schemes [25].

On the other hand, leading memory manufacturers have announced the imminent availability of innovative byte-addressable PM technologies (e.g., phase-change, spin transfer torque and resistive RAM [35]). Besides providing persistent working sets to applications via a DRAM-like memory bus, the future PMs are expected to achieve read access performance in the same order of magnitude as DRAM, together with significant enhancements on the capacity and energy efficiency fronts. However, despite notable improvements in write endurance and performance relatively to NAND SSD, the future PM technologies are still expected to suffer from write wearing and have much higher write latencies when compared to DRAM [38,55].

The combined power of HTM and PM has the potential to enable drastic accelerations of modern data-intensive applications. This unprecedented opportunity promises to deliver performance

☆ This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2019 and PTDC/EEISCR/1743/2014.

* Corresponding author.
E-mail addresses: daniel.castro@ist.utl.pt (D. Castro), romano@inesc-id.pt (P. Romano), joao.barreto@tecnico.ulisboa.pt (J. Barreto).

https://doi.org/10.1016/j.jpdc.2019.03.009
0743-7315/

and scalability levels not attainable with block-oriented storage, and at a fraction of the development cost/complexity compared to lock-based synchronization schemes.

Unfortunately, though, the design of existing HTMs raises non-trivial issues when these are used in conjunction with PM. Whenever a transaction commits, current HTM systems only guarantee that the writes issued by the transaction are made visible to other cores via the cache coherence protocol. However, the updates produced by committed transactions are not atomically flushed to (persistent) RAM, and may linger in the CPU cache for an arbitrary amount of time before being written back to PM. Unfortunately, caches are expected to remain volatile [10,50,53]. As such, upon failure, only a subset of the committed writes to PM may have been persisted — failing to guarantee crash-atomicity [4].

A natural approach to tackle this problem would be to borrow methodologies from durable transactions in traditional block-based storage, such as undo/redo logging. Such schemes resort to an additional persistent log, which tracks all the writes issued by transactions. To satisfy crash-atomicity, they rely on a crucial assumption: the ability to ensure that all transactional updates persist in log *before* any persistent data is modified.

Unfortunately, though, commodity HTM fail to meet this assumption. HTM implementations, such as Intel's TSX [28], do not allow an ongoing transaction to flush the cached log to (persistent) memory before the HTM commits the transaction.

This paper presents NV-HTM, a system that introduces a novel hardware–software co-design to allow the execution of transactions over PM using unmodified commodity HTM implementations.

In a nutshell, NV-HTM instruments the writes issued with HTM transactions to track them in a redo log. Once committed, a transaction makes its effects (and its logs) visible to other concurrent threads, but not necessarily persisted in PM — we call this a *non-durable commit*. However, when a thread commits an HTM transaction T, in durable fashion, it postpones the commit event until T's log (and the log of any transaction T may depend upon) has reached PM — but without waiting for the actual in-place writes to be flushed to PM. Only at this point we say that the transaction is *durably committed*.

This approach guarantees that whenever a transaction's commit is externalized, all of its log entries have been persisted; however, the application's state persisted in memory may reflect only partially the updates of both durably and non-durably committed transactions. The key insight at the basis of NV-HTM is to discard, upon recovery, the persisted application state and to reconstruct it, replaying the logs of all the durably committed transactions on a consistent checkpoint.

Turning this high level design into an efficient and correct algorithm required tackling three main challenges.

1. HTM-compatible asynchronous logging. The first challenge is how to build an efficient logging scheme that is compatible with current off-the-shelf HTM implementations. This raises two main obstacles: first, typical logging schemes that require flushing the log before the hardware transaction is committed are not acceptable; second, the use of centralized logging schemes in highly concurrent environments, such as typical HTM systems, would represent an inherent contention point and limit the system's scalability in update intensive workloads. NV-HTM tackles these issues by relying on a decentralized design, where each thread maintains a local log storing only information related to transactions that it executed. Additionally, NV-HTM builds a non-durable log during execution of a transaction. However, it is only persisted after the HTM commit event (non-durable commit), a commit marker is then appended via software (durable commit).

2. Enforcing transactions' dependencies. The necessary distinction between the non-durable and the durable commits raises a key challenge: how to ensure that the serialization order of non-durable commits (defined by HTM) is consistent with the order of durable commits (defined in software by NV-HTM when it flushes a transaction's log).

In fact, NV-HTM can only flush the logs generated by a transaction T (i.e., durably commit T), after T has been non-durably committed by HTM. This means that other transaction T' may observe T's updates before they are persisted, where T''s log entries persisted before Ts. Then, upon recovery, T would be discarded and T' replayed. This would break consistency, since T' depends on (i.e., reads from) T.

3. Checkpointing process. Another crucial issue is related to minimizing the computational and spatial overheads, as well as reducing PM wear off, associated with the checkpointing process that NV-HTM employs during recovery — to construct a consistent snapshot reflecting the execution of all and only the transactions durably committed before a crash — and during the normal operational mode — to bound the log's growth and the duration of the recovery process. We address this challenge with a checkpointing mechanism called Backward Filtering Checkpoint (BFC), which filters repeated writes and/or flushes to cache lines that are updated multiple times by different transactions in the log. Furthermore, NV-HTM relies on the Copy-on-Write (CoW) mechanism provided by modern OSs to minimize its spatial overhead.

NV-HTM efficiency is evaluated experimentally by means of synthetic and standard benchmarks and including as baselines both approaches based on pure software implementations [50] as well as relying on ad-hoc hardware mechanisms [4]. The results show that NV-HTM can achieve up to **10**× better performance and reduce number of flushes to PM by a factor **11.6**× with respect to state-of-the-art solutions.

This article extends a previous conference paper [9] by:

- Presenting the pseudo-code that formalizes the behavior of the BFC algorithm (Section 5.2).
- Discussing how to tackle the issue of unsynchronized physical clocks across different cores in multi-socket CPU architectures (Section 5.3) and how to cope with overflows of the physical clock, which may arise in long running applications if small timestamps are used to reduce memory consumption (Section 5.3.3).
- Extending the experimental study to include all STAMP benchmarks (Section 6.3), a porting of TPC-C [49], a popular benchmark for OLTP databases, adapted to operate in-memory and using hardware transactions (Section 6.3.1), as well as evaluating the performance of NV-HTM over volatile RAM, i.e., without injecting latency to emulate PM.
- Introducing a critical discussion on the limitations of the proposed solution (Section 7).

The remainder of this paper is structured as follows. Section 2 surveys the related work on PM and Transactional Memory on PM systems. Section 3 presents limitations and challenges of commodity HTM when combined with PM. Section 4 summarizes the main components of NV-HTM. Section 5 presents the different algorithms and data structures employed by NV-HTM, as well as the correctness proof of NV-HTM. In Section 6, we evaluate NV-HTM experimentally, by comparing it with other state-of-the-art solutions. Finally, Section 7 discusses the limitations of the proposed solution and Section 8 concludes this paper.

2. Related work

Several works investigate the problem of exposing emerging PM technologies to applications. A first approach is to retain the same abstractions that already exist for block-oriented storage, optimizing their implementation to benefit the most from distinguishing performance characteristics and low-level consistency guarantees of PM [2,12,20,33,44,53].

An alternative approach exposes PM as persistent heaps that applications can access directly via memory loads and stores [11,50]. While appealing, attaining the benefits of persistent heaps depends on maintaining persistent data in such a way that (*i*) efficiency is preserved, given the unique performance characteristics of PM (namely, minimizing the high costs of CPU cache flushes and PM writes), and (*ii*) updates are ensured to consistently survive across failures.

A general approach is to provide failure-atomic sections on top of persistent heaps. Besides the classical lock-based sections [10,29], failure-atomic sections can be implemented using the Transactional Memory (TM) abstraction. The first proposals of failure-atomic memory transactions for the emerging PM technologies were proposed in the context of Mnemosyne [50] and NV-heaps [11]. Both essentially depart from mainstream software transactional memory (STM) implementations and extend them with logging and recovery mechanisms to ensure durability. Since these initial proposals, failure-atomic memory transactions have received substantial attention in the literature, e.g., [26,34,40,41].

Unlike NV-HTM, which builds on existing HTM supports, the above-mentioned solutions rely on software-based implementations of the TM abstraction. As such, they interpose an extra software layer that can introduce significant overheads [18].

Avni et al. [4] and Wang et al. [52] were the first to exploit HTM for building failure-atomic transactions on PM. Like our solution, these proposals promise to hide the sequential overheads of STM-based approaches. In contrast to our solution, though, both proposals rely on non-trivial alterations to existing HTM designs, e.g., assuming the ability to atomically flush log entries to PM from within a hardware transaction or storing additional transactional metadata in each cache line. As such, these solutions cannot be used on current best-effort HTM implementations. Similar assumptions on the availability of non-standard HTM supports are required to PHyTM [3], which extends prior work [4] to support concurrent execution of both hardware-based and software-based transactions.

Underlying all paradigms, a number of common non-trivial problems at lower layers have received attention from the research community. These include hardware/hybrid mechanisms that aim at improving the performance, reliability and durability of PM, through mechanisms such as write buffering and coalescing, wear leveling and salvaging [42]. Furthermore, supporting PM as a first-class citizen has produced initial steps towards novel proposals to redesign fundamental operating system (OS) mechanisms [5]. All these techniques are complementary to our solution.

NV-HTM shares the approach of maintaining a stable snapshot in addition to a working copy with some recent works. In the PM domain, examples include AdaMS [19], SoftWrAP [22] and Kamino-Tx [41]. None of them, though, targets the problem of enabling the use of HTM over PM.

Concurrently with our research, the following works [23,24,31,39], which have very recently addressed the problem of supporting PM in commodity HTM implementations. DudeTM [39] provides a generic checkpointing mechanism, which can be coupled with both software and hardware TM. However, its reliance on a shared logical clock to serialize HTM transactions leads to poor scalability in update-intensive workloads, as we will show in Section 6.2. Giles et al. [23,24] avoid this issue by proposing cc-HTM [23] and WrAP [24], which uses physical clocks for logged transactions ordering, analogously to NV-HTM. However, writes are buffered in an alias table [22], thus, upon each read or write the address must be translated in software. Instrumenting read accesses in software introduces significant overheads, since read operations are typically dominant in TM workloads [7]. Further, differently from NV-HTM, cc-HTM and WrAP do not employ any filtering techniques during the log replay phase to extend the life expectancy of PM. Finally, Joshi et al. propose DHTM [31], which requires extensive modifications to hardware, including a log buffer in the L1 cache and tracking evicted cache lines in a transaction log in PM. Conversely, NV-HTM is designed to operate on commodity HTM implementations, such as Intel's TSX.

Finally, NV-HTM builds on the literature in DBMS [45], which developed numerous alternative approaches relying on undo/redo logging techniques. These approaches rely on the assumption that the system can exert control on the timing with which log entries and data updates are persisted — an assumption not met by available HTMs, precluding the direct applicability of this class of solutions.

3. Background on HTM

Existing HTM systems come with different flavors and limitations. Despite their differences, though, all HTM implementations keep track of the transactional memory footprint in the processor's caches, including conflict detection which is done at the cache coherency protocol. This architectural design has several important implications, which we discuss next.

The first one is that current HTM systems provide a *best-effort* implementation of the TM abstraction, in the sense that transactions are not guaranteed to commit even if they run in absence of concurrency. Existing HTM systems can only commit transactions whose memory footprint does not exceed cache capacity. And, even in such case, they typically provide no guarantee on the ability to successfully commit transactions. As such, HTM-based applications must rely on a fallback mechanism to guarantee progress.

The default approach is to re-try a hardware transaction some predetermined number of times, and then acquire a Single *Global* Lock (SGL). The SGL aborts any concurrent hardware transaction, hence ensuring the necessary isolation at the cost of serializing the execution of transactions.

Another important implication stemming from the cache-centric design of current HTM systems is that, upon the commit of a transaction, its updates are not immediately nor atomically flushed to the original memory locations (in DRAM or PM). Conversely, updates of committed transactions are made visible to other threads via the cache coherency protocol, which ensures that any copies that remote CPU caches may be storing are atomically invalidated.

Further, HTM implementations, such as Intel's TSX, simply abort transactions that attempt to flush any cache line that they have previously updated, as it would imply externalizing the writes produced by uncommitted transactions. In order to maximize portability, NV-HTM relies on a minimalist set of assumptions regarding the underlying HTM system, which are currently met by every existing HTM implementation (we are aware of). Specifically, NV-HTM assumes a best-effort HTM implementation, that commits transactions in volatile caches and exposes a conventional API for transaction demarcation to begin, commit and abort transactions.

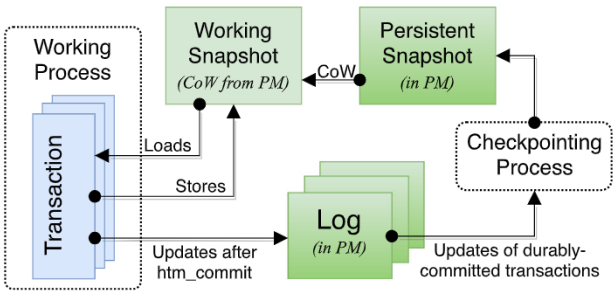

Fig. 1. High level architecture of NV-HTM.

4. System architecture

Analogously to other recent software libraries for building PM-based applications, e.g., [26,34,41,50], NV-HTM exposes PM to applications as transactional persistent heaps.

Each PM heap is uniquely identified by a file name, maintained in a local filesystem mount tree. Upon its initialization, NV-HTM *mmap*s the PM-backed heap in the virtual address space of an application process. This mapping uses Linux's direct access for files (DAX) option [48], which bypasses the OS page cache in DRAM, allowing applications to directly access the PM (via the CPU cache) by load/store instructions.

Furthermore, applications encapsulate PM accesses in hardware transactions via a set of macros that allow to intercept both transaction demarcation calls (begin/commit/abort) and load/stores to memory and inject NV-HTM's logic. NV-HTM assumes that transactions operate exclusively on memory locations belonging to a PM heap, exposing simple and intuitive semantics to applications. This would not be possible if transactions were allowed to span both persistent and volatile heaps: all the updates produced by (durably) committed transactions are guaranteed to be recoverable in presence of crash failures.

The high level architecture of NV-HTM is illustrated in Fig. 1 and comprises the following logical components:

- a ***working process*** **(WP)**, a process that *mmaps* the persistent heap, which we denote as the *Working Snapshot (WS)*. The WP runs a set of parallel *worker threads*, which execute hardware transactions on the WS. The WS is mapped as private (according to POSIX.1-2001), which determines that any update that the worker thread performs to a page in the persistent heap is not actually propagated to that page. Conversely, the OS uses Copy-on-Write (CoW) to transparently create a *volatile* copy of the PM page. Hence, although the WS initially maps pages that are entirely stored in PM, it will usually comprise a mix of clean pages in PM and dirty page copies in DRAM. As such, when the HTM commits a transaction issued by a worker thread, the updates of the transaction are volatile; at that point, we say that the transaction is *non-durably committed*. The commit event of a transaction is exposed to applications only after its updates are persisted in the PM-backed log (see next): only here, we say that a transaction is *durably committed*.
- a **durable log**, stored on a distinct PM heap, which is used to track the updates generated by (durably) committed transactions. The log is updated by each worker thread at some point *after* the HTM commit event. Only at this point we say that the transaction is *durably committed*. It is based on a decentralized/per-thread design (i.e., maximizing locality and minimizing synchronization issues): each thread maintains its own log that tracks solely the transactions the thread processes.
- a ***checkpointing process*** **(CP)**, which is in charge of applying the updates stored in the logs with the twofold purpose of (i) building a consistent *Persistent Snapshot* (PS), which reflects all and only the updates of durably committed transactions, and (ii) pruning the logs, so to ensure that their size never exceeds a predefined (user-tunable) maximum threshold.

This design has several key advantages. First, it allows for isolating, in a lightweight and efficient way, the WP and the CP. Executing hardware transactions on the WS and applying, in a controlled way, the corresponding updates to the PS are two key ideas at the basis of NV-HTM's design. Both are crucial for solving the CPU caching issue.

Further, the usage of an OS-based COW mechanism allows to achieve such isolation by minimizing both the instrumentation costs and memory overheads: instrumentation costs can be significantly reduced since modern CoW implementations are extremely optimized and leverage on dedicated hardware mechanisms [30]; memory overheads can be strongly reduced since only the recently updated pages require a copy (in WS).

Further, the choice of maintaining the updated pages of the WS on volatile memory, rather than on PM, provide a twofold benefit: the faster DRAM's write speed, and a drastic reduction of the write load that actually hits PM, which translates into a corresponding increase of its expected lifespan.[1]

5. Implementation

This section presents NV-HTM's design and implementation. Section 5.1 analyzes transaction processing and log management by the WP, Section 5.2 focuses on the CP and Section 5.3 discusses correctness.

5.1. Transaction processing

The pseudo-code formalizing the behavior of thread t (out of a total of N threads) of the WP is presented in Algorithm 1. For simplicity, the pseudo-code refers to transactions executing in hardware. The management of transactions that use the SGL path, though, is very similar and differences are briefly discussed at the end of this Section.

Data structures. Two main shared data structures are used:

- *log* (Line 2): a log maintained in PM, which, as discussed, has a per-thread structure and is also shared with the CP. Each thread's log is managed as a circular buffer via two pointers, *startP* and *endP*, which point to the first and last entries in that log, respectively. Log entries have a fixed structure composed of a pair of 8-byte values, which are used to store either the address and corresponding value written by a transaction, or a commit marker and the corresponding commit timestamp.
- *ts* (Line 3): an array of N scalars, which is stored in volatile memory. $ts[t]$ is set to ∞ if thread t is not processing a transaction; else, it stores a (physical) timestamp that is used to serialize the transaction being currently processed by t.

Additionally, each thread maintains two local variables: a scalar variable used to store the timestamp to be assigned to a committing transaction, *locTS*; a boolean flag, *isRO*, which identifies whether the transaction is read-only or not.

As already mentioned, NV-HTM relies on a hardware–software co-design: it builds on HTM's atomicity and isolation guarantees and extends them via a lightweight software instrumentation

[1] We note that it would be feasible to map the updated WS pages to a different PM heap instead of volatile memory. This would lead to renouncing to the above advantages, and require a custom implementation of the *mmap* system call in order to instruct the OS to use PM as target of the CoW mechanism; hence, we did not opt for this option in our current implementation of NV-HTM. However, since in the future PMs are expected to achieve higher density/become more cost-effective than current DRAM, such an alternative may, at some point in time, become more attractive than the current hybrid architecture that relies jointly on volatile and persistent memories.

Algorithm 1 WP: transaction processing at thread t

```
1:  Shared variables:
2:      log[N]                                   ▷ One log per thread, stored in PM
3:      ts[N] ← {+∞, ..., +∞}
                  ▷ Per-thread timestamp of active tx; +∞ if none is active
4:  Thread local variables:
5:      locTs                                    ▷ timestamp of committing transactions
6:      isRO                                     ▷ flag used to identify read-only tx
7:  function BEGIN
8:      isRO ←TRUE
9:      ts[t] ←READTS()
10:     mem_fence                                ▷ Ensure other threads know we are in a tx
11:     htm_begin()                              ▷ Start hw tx
12: function WRITE(addr, value)
13:     isRO ←FALSE
14:     if logCheckSpace (log[t])=FULL then
15:         ABORT(LOG_FULL)
16:     *addr ← value                            ▷ Write to working snapshot
17:     log[t].append(< addr, value >)
18: function ABORT(abort_code)
19:     htm_abort(abort_code)
20:     ts[t] ← +∞
21: function COMMIT
22:     if isRo then                             ▷ Commit logic for read-only txs
23:         htm_commit()
24:         ts[t] ← +∞                           ▷ Others do not need to wait for RO tx
25:         WAITCOMMIT()
26:     else                                     ▷ Commit logic for update txs
27:         locTs ← READTS()
28:         htm_commit()
29:         ts[t] ← locTs
30:         logFlush(log[t])                     ▷ Flush current log entries
31:         WAITCOMMIT()
32:         log[t].append(< COMMIT, locTS >)
33:         log[t].endP ← locEndP
34:         logFlush(log[t])                     ▷ Flush commit marker and endP
35:         ts[t] ← +∞
36: function WAITCOMMIT
37:     for all t* ∈ [1, N] s.t. t* ≠ t do
38:         wait until ts[t*] > ts[t]
```

to ensure crash atomicity. Specifically, NV-HTM requires instrumenting the methods used to begin and end (i.e., commit/abort) transactions, plus the method used to write. It is worth noting that NV-HTM spares from the cost of instrumenting read operations: this is key to minimizing run-time overhead, since read operations tend to largely outnumber write operations in typical TM workloads [15].

Transaction begin (Line 7). Before activating a hardware transaction via the *htm_begin*() primitive, t performs the following steps: sets the *isRO* flag to true, marking the transaction initially as read-only; it updates $ts[t]$ with the current value of the machine's physical clock (via the RDTSCP() instruction [27]) and ensures that this value is visible to other threads via a memory fence. As we will see, this mechanism allows to safely establish, before durably committing a transaction T, whether there is still any non-durably committed transaction T' that may precede T in the serialization order.

Write operations (Line 12). Upon a write, the transaction is marked as non read-only via the *isRO* flag and an entry is appended to the log. This is done only after having ensured, via the logCheckSpace() primitive, that the log has sufficient capacity for storing both the current entry and the transaction's commit timestamp — otherwise aborting right away the transaction.[2] This ensures that, if the transaction reaches the commit phase, there is enough log capacity to append the commit marker.

Commit (Line 21). The commit logic differs for read-only and update transactions. Let us analyze first update transactions.

Before using the *htm_commit*() primitive to perform a non-durable commit, the current value of the physical clock is read and stored in the variable *locTs*. If a transaction T is successfully committed in hardware, t first advertises, via the $ts[t]$ variable, the commit timestamp of T. Next, it flushes the current log entries to PM and starts a waiting phase (WAITCOMMIT() function) that aims at ensuring the following key property: in the moment in which T is durably committed, i.e., the commit marker for T is flushed into the persistent log, the system must have already durably committed every transaction T^* that (i) was serialized before T by the HTM system and (ii) with which T has developed a read-from or write–write dependency either directly or indirectly.

NV-HTM ensures this by having t compare the commit timestamp of T with the value advertised in the *ts* array by all other threads: if t finds that there exists some thread, say t^*, which advertises a time stamp smaller than t's, it means that t^* has either started a transaction T^* before T obtained its commit timestamp, or that T^* obtained a commit timestamp smaller than T. In both cases, it is possible that T read from T^* or that T overwrote some memory region that T^* also wrote to. In both cases it could be unsafe to durably commit T, as there are no guarantees that T^*, which T might depend on, has already been durably committed: if T^* fails to durably commit (because of a crash) then, upon recovery, T would be replayed, but T^* would not, thus yielding an inconsistent state.

Once the waiting phase is completed, the commit marker for T is appended to the log, the log's end pointer is updated (based on *locEndP*) and these changes are flushed to PM. At this point, t's timestamp in the *ts* array can be reset to $+\infty$, to advertise that t is no longer processing a transaction.

The commit logic for read-only transactions is simpler: as read-only transactions do not alter the WS, their timestamp is set to $+\infty$ right after they are non-durably committed (so to ensure that no other concurrent transaction waits for them). However, before externalizing their commit to applications, read-only transactions need still to undergo the waiting phase (Line 37) in order to ensure that any transaction they may have read from has already been durably committed.

Abort. Upon abort, all other threads must be aware that t is no longer running a transaction (Alg. 1 line 20).

Fallback path. The instrumentation for the fallback path is similar to the speculative path, with just some minor differences. If t executes in the fallback path, it must ensure that any concurrent transaction that will start after t releases the SGL will durably commit only after the log changes produced by t's transaction have been fully flushed to log. This is achieved by having the SGL-holding transaction advertise the current timestamp in $ts[t]$ after releasing the SGL. Analogously, the fallback path also needs to go through the wait commit phase, in order to take into account dependencies that could arise between the SGL path and any transaction that was non-durably committed when the SGL was acquired.

5.2. Log checkpointing

Unlike existing solutions [3,4] NV-HTM removes the propagation of updates to the PS from the critical path of transactions; only the flushing of the transaction's log to PM is kept within the critical path. This design choice brings about both opportunities and challenges. The key challenge is how to efficiently bound

[2] We omit the abort handling logic, which, in this case, will wait till additional log space is available before re-starting the transaction to avoid the lemming effect [14] and unnecessary activations of the SGL path.

Algorithm 2 CP: replaying the log - BFC

```
1:  Variables:
2:      log[N]                          ▷ One log per thread, stored in PM
3:      tmp_ptrs[N]                     ▷ Stores the startP for each log, stored in PM
4:      CPLog                           ▷ Auxiliary log to advance the startPs, stored in PM
5:      log_next_tx[N]                  ▷ Pointer to the next transaction, stored in RAM
6:      tx_queue                        ▷ Queues the sorted transactions, stored in RAM
7:      filterMap                       ▷ Indexes written cache lines and words, stored in RAM
8:  function EXEC_BFC
9:      INIT_TRUNCATE(threshold);
10:     SORT();
11:     REPLAY();
12:     FLUSH_MODIFIED();
13: function INIT_TRUNCATE(threshold)
14:     approxEndP ←compute_approxEndP(log[1], threshold)
15:     firstTx ←find_TX_at(log[1], approxEndP)
16:     log_next_tx[1] ← firstTx        ▷ First transaction to replay (near endP)
17:     tmp_ptrs[1] ← firstTx.pos       ▷ Stores log position of the first transaction to replay
18:     for l ∈ [2, N] do
19:         log_next_tx[l] ←find_next_TX(log[l], firstTx.ts)
20:         tmp_ptrs[l] ← log_next_tx[l].pos
21: function SORT
22:     (nextTx, l) ←next_TX()
23:     put_in_tx_queue(nextTx)
24:     log_next_tx[l] ←find_next_TX(log[l], nextTx.ts)
25: function WRITE_TX(tx)
26:     for each write ∈ tx.writeSet do
27:         CLPos ←find_cache_line(filterMap, write.addr)
28:         if CLPos ≠ ∅ then
29:             if write.addr ∉ CLPos then
30:                 *write.addr ← write.value
31:                 add_addr_to_cache_line(CLPos, write.addr)
32:         else
33:             *write.addr ← write.value
34:             add_modified_cache_line(filterMap, write.addr)
35: function REPLAY
36:     while TXsToApply do
37:         tx ←get_from_tx_queue()
38:         WRITE_TX(tx)
39: function FLUSH_MODIFIED
40:     for each cl ∈ filterMap do
41:         flush(cl)
42:     log_startP(tmp_ptrs)            ▷ Log startP to enable redo in case of crash
43:     set_and_flush_new_startP(tmp_ptrs)
44:     destroy_CPLog()
```

the growth of the log, a property that is desirable both to minimize consumption of PM resources and to limit the duration of the recovery phase. NV-HTM tackles this challenge via a novel log checkpointing mechanism that we named *Backward Filtering Checkpointing* (BFC), presented in Algorithm 2.

Backward Filtering Checkpointing (BFC) Intuitively, BFC considers a snapshot of the per-thread logs (obtained when BFC is activated) and persists all the updates of durably committed transactions logged in such a snapshot to the PS.

The design of BFC is influenced by the observation that many TM benchmarks and real applications tend to concentrate large streams of updates (issued by different transactions) over a small memory region (i.e., *hot spots*). For each hot spot, the logs contain many repeated updates, which are particularly costly in PM — not only performance and energy-wise, but also given PM's limited write endurance.

The key insight of BFC is that, when checkpointing a set of updates that target the same memory location, only the most recent one needs to be propagated to the PS (in PM), as that update supersedes the older ones in the logs. Another notable feature is that the checkpointing may occur simultaneously with worker threads, which may continue to run and durably commit transactions in the logs.

The BFC algorithm is activated by launching the EXEC_BFC function (Line 8). The first step of BFC is to determine an initial point in the log from which to start the replay phase. As worker threads concurrently append new entries in the log, starting the replay exactly from the end of each per-thread's log may affect hardware transactions in the Working Process — as transactions write the shared log while the Checkpoint Manager reads it, causing possible conflicts. In order to avoid this issue the starting point of the replay phase (*approxEndP*) is computed using a user tunable threshold (by default set to 50% of the maximum log size) starting from the base of the log (i.e., the lowest *startP* pointer across all per thread logs).

Next, BFC iteratively traverses the per-thread logs in anti-commit timestamp order and analyzes the log entries of each durably committed transaction (SORT, Line 21). This step determines which is the latest durably committed transaction, T_{lat}, within the set of logged transactions that still need to be checkpointed. This is easily determined by comparing the commit timestamps of the most recent durably committed transactions in each per-thread log.

As mentioned before, BFC filters out any update that is found to target the same location as a more recent update (which BFC already propagated to PM). To this end, BFC maintains a hash map called *filterMap* (Line 7) indexed by cache line address, whose values store a bitmap encoding which "positions" (*CLPos*) of each cache line have been already updated (due to a more recent transaction) during the current checkpointing instance (WRITE_TX, Line 25). Tracking updates at the granularity of 8 bytes, and assuming cache line width of 64 bytes (as in typical current processors), the *CLPos* bitmap can be compactly encoded using a single byte.

Then, after sorting, transactions are replayed (REPLAY, Line 35). For each logged update of T_{lat}, the *filterMap* is first consulted to determine if the corresponding address has already been encountered (Lines 28 and 29) — in this case the update is skipped. Else, the write is executed, but not flushed, and *filterMap* is accordingly updated to keep track of it.

Although not shown (for simplicity) in the pseudo-code, this two-step approach can be parallelized, using one thread for the SORT and other for the REPLAY. The SORT thread produces log locations, which are enqueued in *tx_queue* (Line 23). The REPLAY thread consumes log locations (Line 37) and immediately replays the transaction write-set.

We note that, upon recovery, the log could contain entries of non-durably committed transactions, which can be easily recognized since they do not have a final commit marker. These transactions can be safely skipped during the checkpointing process, as their effects have not been externalized, neither directly nor indirectly (via other dependant transactions).

Once the backward scanning of the log completes, there is no guarantee that all the updates performed on the PS have effectively reached PM, as some may still be in the CPU cache. Hence, the next phase ensures that all the checkpointed updates are durable: this is achieved by iterating over the *filterMap* and forcing the flush of the cache lines that it tracks (Line 40). The design choice of postponing the flushing of the updated cache lines after the whole backward scanning of the log has not only the advantage of avoiding flushing the same cache line more than once; it also increases the likelihood that, when the flushing of a cache line (updated during the log scanning phase) is requested, the cache line has actually already been written back to PM due to the cache eviction mechanism.

The final step of the checkpointing process consists in advancing the start pointers to the per-thread logs in PM, so to

effectively free up log space for the WP. This last step has to be executed in an atomic fashion, in order to preserve correctness in presence of crashes. This is achieved using a classic technique in the literature on DBMS recovery [45]: a special checkpoint log (*CPLog*) is allocated in PM, and an entry is appended to it for each start pointer to be updated; once all the updates of the start pointers have been recorded in *CPLog*, a special *CP − COMPLETED* marker is appended to it and flushed to PM (Line 42). At this point, the per-thread logs' start pointers are actually updated and flushed too. Any crash occurring during this phase will trigger the replay of *CPLog*, ensuring the atomicity of the start pointers' update. The *CPLog* can instead be safely destroyed after the start pointers of all the per-thread logs have been flushed to PM (Line 44), marking the actual completion of the checkpointing process.

Usages of BFC. NV-HTM relies on BFC in 3 scenarios:

Log pruning. This process is triggered whenever a worker thread detects that its per-thread log has reached a user-defined threshold of its capacity (e.g., 50%). In this case the worker thread unblocks the CP, which executes the BFC algorithm as a non-blocking/background task. The setting of this threshold is associated with a key trade-off: using large threshold values allows for buffering more transactions, and, hence, potentially filtering more writes and flushes. However, it also reduces the portion of the logs that is available to store the updates of transactions that run *concurrently* with BFC. Consequently, there is an increased risk that the worker threads fill up its log before log pruning completes and frees up log space.

Recovery. The BFC algorithm is also activated upon recovery. In this case, the CP is forked from the WP and the former is used to replay the transactions in the log to bring the PS up to a consistent state. Only at this point, the WP *mmap*s the PS (in private mode) and activates transaction processing.

Memory consolidation. Finally, checkpointing the logs via BFC can also be used to achieve, what we call, memory consolidation. Over time, applications may end up updating overly large portions of the PS, which might cause cloning large parts of it into DRAM (via the OS-driven CoW mechanism). The memory consolidation process can be requested by applications when they detect the usage of excessive DRAM memory consumption, discarding every page that the WP may have cloned in DRAM. This is achieved in three phases:
(1) non-blocking log pruning is executed when the following phase starts, its objective is to minimize the number of transactions present in the log (and, hence, its duration);
(2) transaction processing is blocked temporarily and a second pruning is executed, applying in the PS any transaction that durably commits during the first phase;
(3) before resuming transaction processing, the WP *munmap*s the WS and then *mmap*s it again — hence, allowing the WP to release any page of the PS that had been previously cloned in DRAM (via CoW).

5.3. Correctness arguments

The key property at the basis of the NV-HTM's algorithm is that the serialization order obtained by totally ordering the transactions in the log via their commit timestamp is equivalent to the serialization order imposed by HTM. An important preliminary observation is that if two transactions are not *dependant*, i.e., they do not develop any read-from or write–write dependency, either directly or indirectly, then, even if they are ordered in different ways in the log and by the HTM system, they will still produce the same results. Hence, it suffices to prove that the serialization order of durably committed *dependant* transactions determined by their commit timestamp in the log does not contradict the HTM serialization order.

Satisfying the above property depends on the accuracy and synchronization properties of the physical timestamps provided by the CPU. Modern processors provide specific hardware timestamp counters (TSC) that allow programs to read high-resolution CPU timing information with a low overhead, e.g., via the `RDTSC` instruction in Intel processors. However, `RDTSC` may be reordered in the processor's instruction pipeline, thus, the obtained TSC may not provide correct transaction ordering. Hence, in the context of NV-HTM, we rely on `RDTSCP` (Read TSC and Processor ID), which, conversely, prevents this issue.

Most modern processors (e.g., all Intel CPUs supporting *Invariant TSC* since Nehalem family [27,36]) ensure that all cores (including across different sockets), observe monotonic TSC ticking at the same rate, which yields perfect TSC synchronization. Next, we prove the correctness of NV-HTM's algorithm under these assumptions and postpone the discussion on how to cope with clock skew across different core and with wrap-arounds of the TSC to Sections 5.3.2 and 5.3.3, respectively.

The notations $T' \xrightarrow{HTM} T$ and $T' \xrightarrow{LOG} T$ indicate that T′ is serialized before T by the HTM system and according to their commit timestamp in the log, respectively.

5.3.1. Perfectly synchronized TSCs

Under the assumption of perfectly synchronized clocks we can reason assuming a single time source aligned with real-time. We start by proving that, for any two dependant transactions T and T', if $T' \xrightarrow{LOG} T$, then $T' \xrightarrow{HTM} T$. The relative order of T and T' in the log is defined by their timestamps $T.ts$ and $T'.ts$ (resp.), which, we recall, are acquired after the transactions performed all of their memory accesses, i.e., right before committing. Let us denote with $T.commit$ and $T'.commit$ the instants in real time where T and T' commit (resp.). Denote with op and op' any pair of conflicting operations issued by T and T' on a common data item, respectively, and with $t(op)$ and $t(op')$ the real time instant in which these operations are executed. The first observation is that, in order for *both* T and T' to be able to commit in HTM, then when the last of the two conflicting operations is executed by any of the two transactions, the other transaction must have already committed, namely:

$$max(t(op), t(op')) > min(T.commit, T'.commit) \quad (1)$$

Else, if both transactions were still active when the last of the conflicting operation were to be executed, every HTM implementation we are aware of [43] would trigger the abort of (at least) one of the two transactions.

Let us assume, by contradiction, that $T' \xrightarrow{LOG} T$, i.e., $T'.ts < T.ts$, and $T \xrightarrow{HTM} T'$, namely $T.commit < T'.commit$, where T and T' are dependant transactions. Since the timestamp of a transaction is established right before requesting its commit, we have that:

$$t(op) < T.ts < T.commit \quad t(op') < T'.ts < T'.commit \quad (2)$$

However, since both T and T' commit despite having generated (at least a pair of) conflicting operations, and since T commits before T' in real-time, using inequality (1) we have: $T.commit < t(op')$. Combining the last inequality with the inequality (2), we obtain $T.ts < T'.ts$ and contradict the hypothesis.

It remains to prove that if T is durably committed and it depends on (i.e., it reads from or overwrites a memory address previously written by) a transaction T' (which implies $T' \xrightarrow{HTM} T$), then T' is also durably committed. This is necessary to guarantee recoverability, since it ensures that if a transaction T is replayed upon recovery, so is any other transactions T may depend upon.

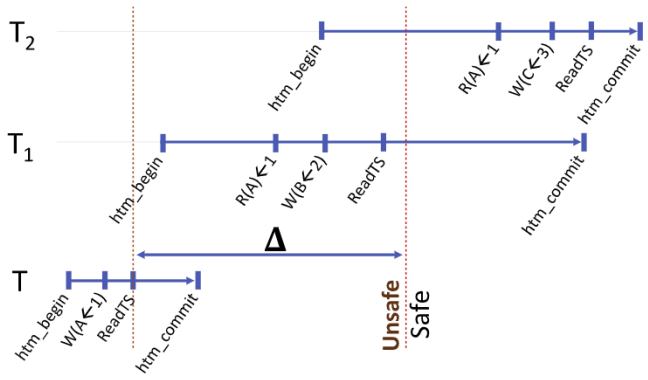

Fig. 2. T_1 can be reordered after T in the log (despite having been serialized before T by the HTM), since T_1 commits within Δ time units (where Δ is maximum skew between TSCs) since T does. T_2, conversely, is spared from the risk of being reordered in the log before T, given that it commits more than Δ time units after T.

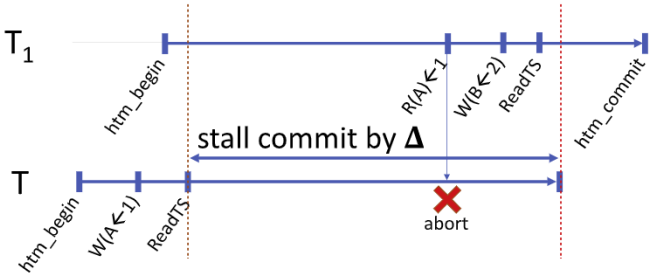

Fig. 3. By stalling the commit of T for Δ time units (where Δ is maximum skew between TSCs), we force a conflict between T and any other transaction that depends on T and may be reordered before T due to a clock skew issue (T_1 in the exemplified scenario).

A transaction T is durably committed only if it passes the WAITCOMMIT() function, which forces T to wait for all threads who advertise a timestamp $ts^* < T.ts$. Hence, if T' has not fully flushed its log (Alg. 1, line 34), when T reaches its wait phase, T will necessarily block until the thread executing T' sets its ts to $+\infty$ (Alg. 1, line 35), i.e., until T' has flushed its log to PM.

This implies that, if a transaction externalizes its commit (by returning from the invocation of the commit procedure), it cannot depend on any non-durably committed transaction.

5.3.2. Bounded skew between TSCs

The algorithm described so far assumed perfectly synchronized TSCs. However, perfect synchronization of physical clocks is not necessarily guaranteed, e.g., on older CPUs or in some machines equipped with multiple CPUs.[3]

In the above scenarios, it is no longer be possible to use a single time source when reasoning on the timestamps obtained by different threads. All that can be assumed is that TSCs grow monotonically at each core, and that the TSCs across multiple cores are synchronized within a given upper bound, which we assume to be known and equal to Δ (measurable using existing techniques in the literature, e.g., [32,54]). In the following we prove that NV-HTM's algorithm remains correct by simply forcing a delay equal to Δ after reading the TSC (Alg. 1, line 27) and before requesting the HTM to commit (line 28). The rationale underlying the mechanism is illustrated in Fig. 2. A transaction T' that was serialized by the HTM after some transaction T and depends on T (i.e., issued a conflicting operation with T, e.g., T' read what T wrote) can only be reordered in the log before T if T' obtained its timestamp within Δ time units after T did (Fig. 3). By forcing T to wait out Δ time units before committing, any transaction T' that may risk to be reordered before T in the log, must necessarily issue an operation that conflicts with T while T is still active, causing the abort of either T or T'.

In the following, we prove that for any two dependant transactions T and T', if $T' \xrightarrow{LOG} T$, then $T' \xrightarrow{HTM} T$. We proceed by contradiction and assume that $T' \xrightarrow{HTM} T$, where T depends on T', and that $T \xrightarrow{LOG} T'$. Let us denote with $RT(T'.ts)$ and $RT(T.ts)$ the real time moments in which the timestamps of T' and T', respectively, are obtained. We can express the timestamps of T and T' as follows: $T.ts = RT(T.ts) + \varepsilon_0$ and $T'.ts = RT(T'.ts) + \varepsilon_1$, where ε_0 and ε_1 denote the skews relatively to the real-time clock for T and T', respectively. For the sake of simplicity and with no loss of generality, we assume one of the core-local clocks to be synchronized with the real-time clock, e.g., $T.ts$, thus $\varepsilon_0 = 0$.

By inequality (1), and given the assumptions that $T' \xrightarrow{HTM} T$ and that T depends on T', we have that $RT(T'.ts) < RT(T.ts)$. But if T' is ordered after T in the log ($T \xrightarrow{LOG} T'$) then $T.ts < T'.ts$, which implies that $RT(T'.ts)+\Delta > RT(T.ts)$, i.e., $RT(T'.ts)-RT(T.ts) < \Delta$. However, if in real-time T reads its TSC within Δ time units after T' did, then it means that when T read its TSC T' must be still active — since T' is forced to wait Δ time units after $RT(T'.ts)$ before it is allowed to commit. If this is the case, though, either T or T' should be aborted by the HTM, since they conflict — contradicting the assumption $T' \xrightarrow{HTM} T$.

5.3.3. Clock overflow

x86 CPUs usually provide a 64-bit TSC. This means that TSC overflows (resets to 0) after many years, depending on the CPU clock rate. For instance, assuming a clock rate of 3.0 GHz and a TSC increment upon each clock cycle, then the TSC period is $\approx$ 195 years.

For the sake of completeness, though, this Section presents an extension to cope with the eventuality of a TSC overflow. This extension can also be used to enable implementations that use smaller timestamps (smaller than the architecture's TSC width) for compressing the memory footprint of NV-HTM's logs.

NV-HTM relies on TSCs to allow the CP to order transactions. Thus, in the eventuality of overflows, the CP is the component that must detect such situations, i.e., when it reads a timestamp that is not consistent with the adjacent (previous/next) transactions.

As the CP sorts transactions in the per-thread logs in reverse timestamp order (whose values are read from TSC), it can easily detect whether an overflow occurred as, in that case, no per thread log would contain a timestamp smaller than the one of last replayed one (Alg. 2, Line 22). In this case, the CP can simply set the current timestamp to the maximum largest value possible and resume searching in reverse timestamp order.

6. Experimental evaluation

This section presents an extensive experimental evaluation aimed at comparing NV-HTM[4] with two state-of-the-art solutions, based both on software and hardware TM mechanisms.

On the STM's side, we consider as baseline a scheme (PSTM) based on Mnemosyne [50]'s algorithm, which we re-implemented

[3] The TSC at each node is only guaranteed to be synchronized in multi-CPU machines if the hardware ensures that all CPUs receive the reset event synchronously, which depends on the motherboard [27].

[4] Source of our NV-HTM prototype can be found here: https://bitbucket.org/daniel_castro1993/nvhtm.

on top of TinySTM [21] using in-place updates to minimize read-instrumentation costs (analogously to what was done by Avni et al. [4]). In order to support in-place updates, upon each write, an undo log entry is flushed to PM before modifying the data in-place; upon commit, changes are flushed to PM and a commit marker is added to the log; next the transaction's log is discarded.

On the HTM's side, we consider PHTM [4]. Recall that, as discussed in Section 2, PHTM assumes two mechanisms that are not supported by existing HTM implementations, namely the ability to:

1. Transparently flush the logs during transaction execution, without causing its abort.
2. Flush a commit marker to the log atomically upon commit.

Further, unlike NV-HTM, PHTM acquires write locks during transaction execution, which are maintained beyond its commit and until data changes are persisted to PM. As we will see, this has an impact both on the effective memory capacity of hardware transactions, as well as on the contention proneness of transactions. In order to emulate the non-standard mechanisms assumed by PHTM, we follow the approach previously used by the PHTM's authors [4], i.e., we do not issue cache lines' flushes, but replace them with spins in order to account for the latency of persisting those cache lines to PM. Specifically, we inject a latency of 500 ns per cache line flush, as suggested by a recent work surveying NVRAM technology [42]. Note that, in order to ensure a fair comparison among all solutions, we use the same approach (i.e., replacing cache flushes to PM with spins) also when implementing the other approaches considered in this study.

On the STM's side, we consider as baseline a scheme (PSTM) based on Mnemosyne [50]'s algorithm, which we re-implemented on top of TinySTM [21] using in-place updates to minimize read-instrumentation costs (analogously to what was done by Avni et al. [4]). In order to support in-place updates, upon each write, an undo log entry is flushed to PM before modifying the data in-place; upon commit, changes are flushed to PM and a commit marker is added to the log; next the transaction's log is discarded.

All tests were conducted on an Intel Xeon CPU E5-2648L v4 @ 1.80 GHz with 14 physical cores and 28 hardware threads in hyper-threading mode. The machine is equipped with 32 GB of RAM and runs Ubuntu Server 16.04.2LTS (kernel version 4.4.0-57). All the results reported below are obtained as the average of at least 10 runs.

We use both standard benchmarks, i.e., the STAMP suite [8] and a porting of TPC-C [47] for HTM, as well as a synthetic micro-benchmark, called Bank. Bank manipulates an array of d bank accounts, each storing 8-byte long values, via two types of transactions: read-only transactions, which read r accounts selected uniformly at random and return their sum; update transactions, which transfer a random amount between w pairs of accounts, also selected uniformly at random. In order to avoid false conflicts due to cache aliasing, and simplify the analysis of the results, the accounts are cache-aligned. By controlling the above parameters, as well as the percentage of update transactions u, this benchmark allows for precisely shaping the workload and stress different aspects of the compared solutions.

6.1. Impact of checkpointing

The first aspect we evaluate is the impact, on both performance and write wearing reduction, of having the CP running in background to perform log pruning. To this end, we use Bank to generate write intensive workloads, composed of 90% of update transactions transferring money between 2 pairs of accounts. We consider two workloads: a lightly contented one, which uses an array of 16 K elements and where read-only transactions read 128 accounts (i.e., less than 1% of the total), and a contention-prone workload, where the array has 64 elements and read-only transactions read all of them.

In practice, the frequency of activation of the CP depends on the ratio between the amount of log entries generated by the application and the maximum log capacity: the lower the ratio, the least frequently the CP has to perform log pruning, and vice versa. Based on this insight, we consider three scenarios:

- 10× scenario: where an execution generates 10× more log entries than the log's capacity. This is a worst-case scenario for NV-HTM, which is a representative of the steady performance achievable by write-intensive, long running applications that continuously generate a large amount of log entries.
- *NLP* scenario: where there is no need of executing log pruning during the application's run. This can be seen as a best-case scenario, representative of situations in which logs have sufficient capacity to accommodate the entries produced along the whole run and in which log pruning can be avoided or postponed to non-performance critical periods.
- 85% scenario: in which the log entries generated by a run fill approximately 85% of the log's capacity. This implies that the log pruning is activated in background during the run, but that there is sufficient log capacity to ensure that worker threads never block because they exhausted their log.

For the 85% and 10× scenarios, we allocated 40 MB of space per-thread log; set the activation threshold for the log pruning process to 50%; and configured the Bank benchmark to produce a fixed number of transactions per thread, so to ensure that the target "log fill up" ratio is achieved. For the NLP workload we use 256 MB large per-thread logs and generate 1M transactions per thread, which fill $\approx$ 30% of the log capacity, ensuring that log pruning is never activated.

The results of this study are reported in Fig. 4, which considers also two NV-HTM's variants that employ alternative checkpointing schemes, noted NV-HTM$_{FFF}$ (Forward Flush Filtering) and NV-HTM$_{FNF}$ (Forward No Filtering). As their name suggests, unlike BFC, these checkpointing schemes scan the log forward and use less aggressive filtering policies. The FNF scheme simply performs no filtering, while FFF filters out duplicate cache line flushes after the replay.

Performance gains. By analyzing the throughput results (top plots) we observe that, in both workloads, the performance of NV-HTM clearly dominates the alternative schemes' up to 14 threads, i.e., before hyper-threading is activated, with peak gains of up to 2× vs. PSTM and up to 10× vs. PHTM. The speed-ups of NV-HTM vs. STM are due to the hardware-based nature of NV-HTM, which allows for sparing costly software instrumentations. Instead, the striking throughput gains of NV-HTM over PHTM can be explained by analyzing the abort probability plot, which shows that PHTM suffers from significantly larger contention rates (especially in the high contention scenario). This is explicable by considering that, in PHTM, write locks are maintained during a time window that encompasses both the flush of the log (before committing) and of the data (after committing) to PM. This time window, during which any concurrent access to locked data triggers a transaction abort, is, relatively speaking, very large compared to the execution time of transactions in NV-HTM.

Further, up to 14 threads, NV-HTM delivers very similar performance in all the three considered scenarios of log pruning frequency (NLP, 85% and 10×) — which provides experimental evidence on the efficiency and limited overhead of the BFC algorithm. It is worth noting that the NV-HTM's variants that use the simpler FNF and FFF checkpointing schemes incur a dramatically larger overhead: this confirms how crucial it is, from a performance-oriented perspective, BFC's ability to filter both duplicate writes and (even more importantly) cache line flushes

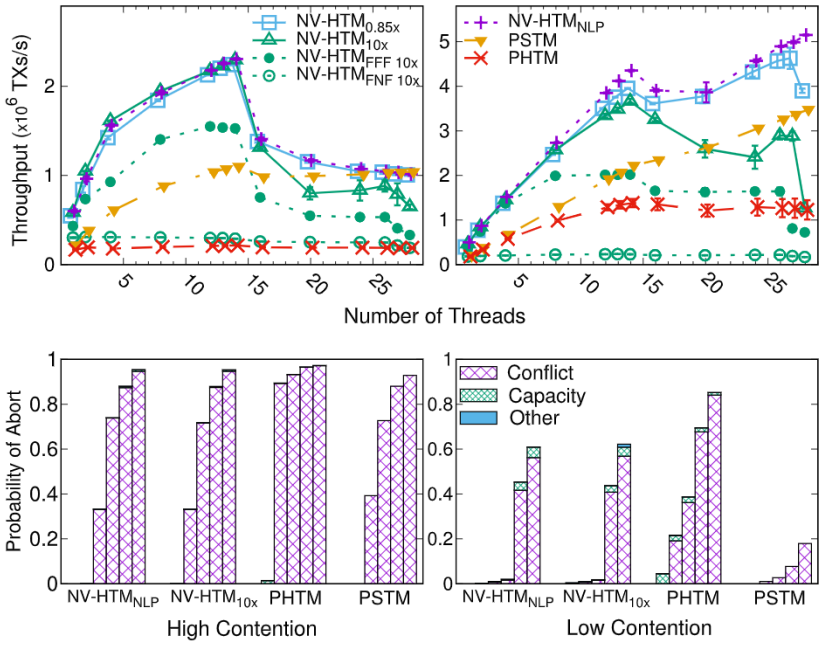

Fig. 4. Performance with different log pruning frequencies. Probability of abort presented for 1, 4, 8, 16 and 26 threads.

to PM and to remove them from the critical path of execution of log scanning.

Above 14 threads, when the CPU starts to operate in hyper-threading mode, HTM is well-known to suffer from performance penalties due to contention on shared architectural resources [18] and, as such, also NV-HTM's performance naturally degrades, although remaining still significantly better than PHTM's and competitive with PSTM. It can also be observed that, in the high log pruning scenario (10×), the performance of NV-HTM degrades on average by ≈15%/40% in the high/low contention scenario compared to the *NLP* scenario from 20 to 27 threads, and an even larger performance toll is paid at 28 threads. This can be explained by considering that, with 20 to 27 threads, the CP shares its underlying physical core with one worker thread; while, when using 28 threads, the CP shares the same physical core with two worker threads. The latter scenario can simply be avoided by statically reserving one logical core; a more interesting alternative is to resort to previously proposed self-tuning parallelism adaptation techniques for TM [1,17,46].

Impact of PM emulation. As mentioned above, in the previous study we emulated the flush of cache lines to PM, replacing them with *spin* times aimed at emulating the expected latency of modern NVRAM technology. This was done in order to allow a fair comparison with PHTM, which requires flushing the transaction's log atomically with its commit — a feature unsupported by existing HTM implementations.

In this section, we exclude from the comparison PHTM and evaluate NV-HTM in a scenario where no latency is injected and cache lines are flushed out of processor's caches, via the `CLFLUSH` x86 instruction, to conventional volatile RAM (and not NV-RAM, which is not commercially available, yet). As such, this study cannot be considered representative of the performance of NV-HTM when used with modern NV-RAM supports — which represent the target PM technology for which NV-HTM was designed. However, it can be seen as representative of a deployment scenario in which DRAM is backed by some external persistent storage (e.g., SSD drives) and capacitors, which are used to power the copy of DRAM to PM in case of system crashes [13].

The plots in Fig. 5 report the results obtained by running the same workloads considered in Fig. 5, but without injecting latency and using CLFLUSH. The high-contention workload (left plot) is, in this case, much less scalable than in the previous study. This can be explained by considering that the latency injected to emulate the latency of flushing logs to PM had the effect of increasing the inter-transactional time,[5] serving as a back-off mechanism that alleviated contention — which, in this new scenario, is significantly higher (especially in NV-HTM, where, unlike PSTM, persists all the log entries of a transaction only after it commits). One can still observe, though, that, also in this scenario, NV-HTM achieves significant performance gains (1.5× to 2× throughput gains) when compared to PSTM.

Let us now analyze the right plot of Fig. 5, which considers the low contention scenario. We observe that, with this more scalable workload, the three variants of NV-HTM that use different log pruning frequency (NLP, 85% and 10×) exhibit different performance. The NLP variant, which assumes the availability of sufficient log-space to avoid triggering the log-pruning process, has the largest gains and scales almost linearly up to 14 threads. Beyond that point, when the CPU operates in hyper-threading mode, performance drops sharply, arguably because transactions' execution time increases (due to contention on hardware resources) and abort rates, consequently, spike. Across all the considered thread counts, though, NV-HTM $_{NLP}$ consistently outperforms PSTM.

When increasing the frequency of activation of the log pruning process (variants 85% and 10×), though, the performance of NV-HTM starts to degrade (when compared with the NLP variant) beyond 8 threads, and remains higher than PSTM's one only up to 14 threads, i.e., before starting to operate in hyper-threading mode. We argue that the reason due to which NV-HTM $_{85\%}$ and NV-HTM $_{10\times}$ perform relatively worse in the settings considered in Fig. 5 than in those considered in Fig. 4 is the following. The emulated PM latency (500 ns) is larger than the latency of CLFLUSH to RAM. Consequently, the system achieves a higher absolute peak throughput, generating a larger load pressure (write

[5] Recall that NV-HTM persists all the log entries of a transaction after it commits. PSTM, unlike NV-HTM, flushes the log entries of a transaction before this commits, but, analogously to NV-HTM, it does persist information after the transaction commits, i.e., it flushes any cache line it updated.

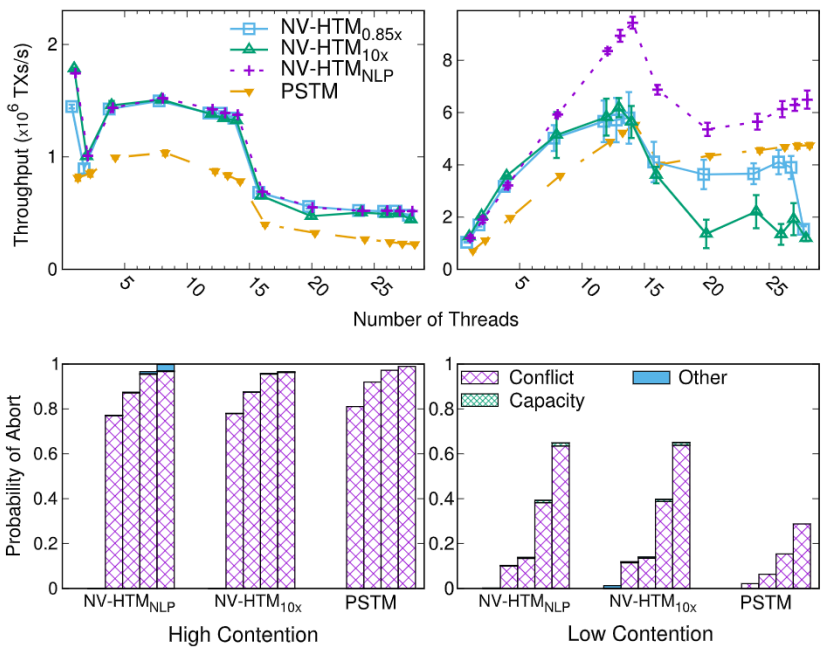

Fig. 5. Same workloads as in Fig. 4, without injecting latency to emulate PM. Probability of abort presented for 1, 4, 8, 16 and 26 threads.

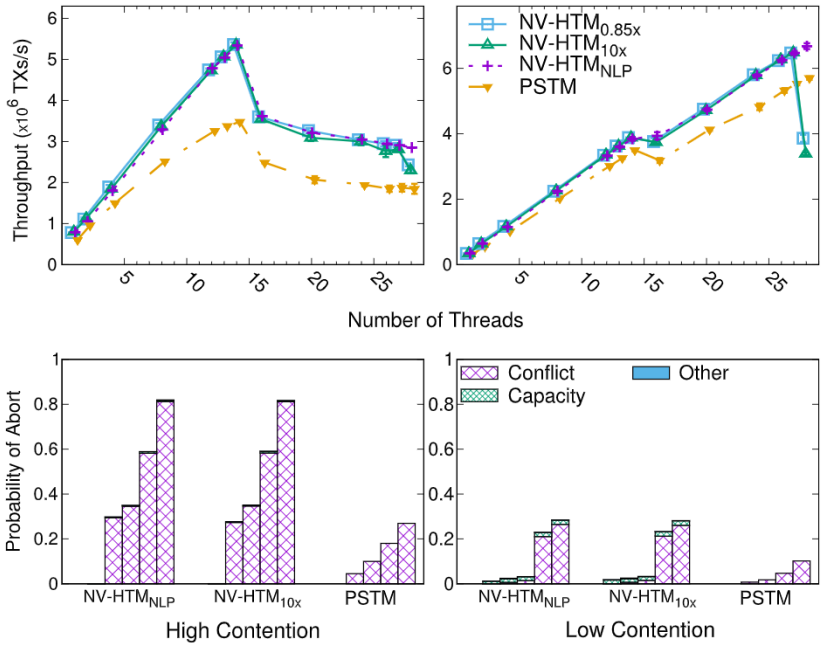

Fig. 6. Same workloads as in Fig. 4, but with 10% of update transactions (instead of 90%) and without injecting latency to emulate PM.Probability of abort presented for 1, 4, 8, 16 and 26 threads.

traffic) to the memory subsystem, which is furtherly amplified by the cache coherency traffic due to the use of CLFLUSH (which not only flushes, but also invalidates the cache line). In these operating conditions, the concurrent activation of the checkpointing process, which injects write traffic to replay the logs to PM, incurs a relatively higher performance toll.

To validate this argument, we conduct a further experiment, in which we changed the workload to generate only 10% of update transactions. This corresponds to reducing by a factor 9× the write traffic generated by the benchmark in comparison to the previous study and, in order to preserve the frequency of log activations in the 85% and 10× NV-HTM variants, we have accordingly increased the number of transactions generated by the benchmark by the same factor. The corresponding results are reported in Fig. 6. The results in the plots show that all the three variants of NV-HTM exhibit similar performance and consistent gains with respect to PSTM.

Overall, this study shows that, despite not originally designed for PM supports based on capacitory-backed RAM, NV-HTM represents a competitive solution also for this type of PM technology. In this case, though, our results suggest that, with workloads that generate intense write traffic, the concurrent execution of the log-pruning process can impose a larger performance toll, especially when the processor operates in hyper-threading mode.

PM write wearing reduction. Table 1 quantifies the gains that NV-HTM attains in terms of PM write wearing reduction by

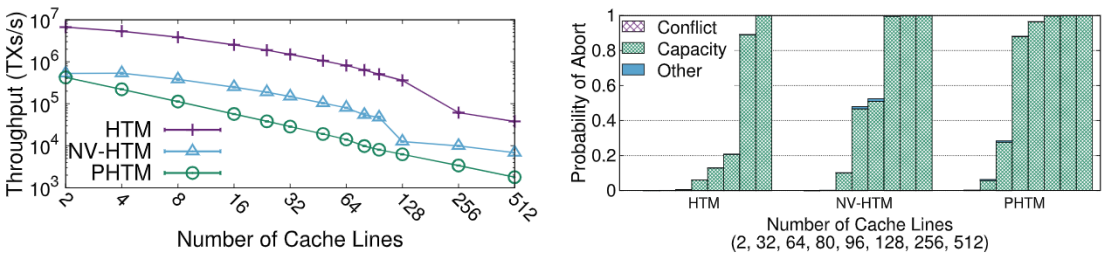

Fig. 7. Evaluating the maximum write capacity of NV-HTM.

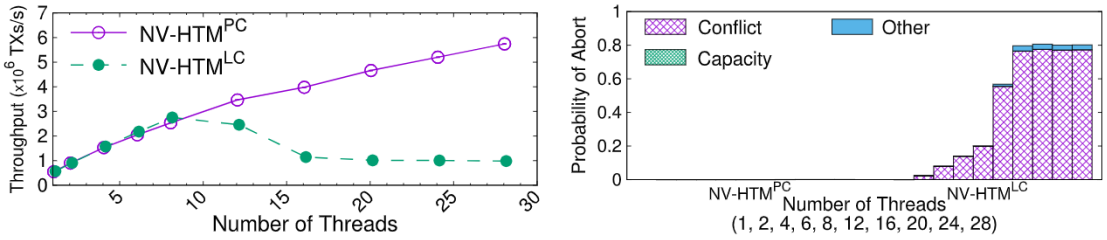

Fig. 8. Contrasting the scalability of NV-HTM with a variant using logical clock to serialize transactions (NV-HTM^{LC}).

Table 1
Average writes/flushes to PM per transaction.

	NV-HTM					PHTM	PSTM
	NLP	(85%)	(10×)	$(10\times)_{FFF}$	$(10\times)_{FNF}$		
Writes	4.5	4.5	4.55	8.1	8.1	9.1	35.2
Flushes	1.8	1.8	1.83	1.83	6.4	6.3	21.2

reporting the average number of memory words written (with 8-bytes granularity) and cache lines flushed to PM per transaction, at 28 threads. NV-HTM performs ≈ 2× less writes and ≈ 3.4× less flushes than PHTM. This gain is directly imputable to NV-HTM's design choice of letting transactions' updates accumulate in the log and of periodically/upon need filtering duplicates via the BFC algorithm; conversely, since PHTM immediately applies a transaction's logged updates to PM right after its commit, it has no opportunity to filtering repeated writes/flushes. Similar considerations apply to the case of PSTM, although the benefits of NV-HTM are even further amplified in the high contention scenario. In fact, in HTM-based solutions (like PHTM and NV-HTM) transactions that do not commit in HTM, their written cache lines never reach (persistent) memory. Being a purely software based solution, though, PSTM writes/flushes log entries to PM during transaction execution, independently of whether they will eventually commit; as such, aborted transactions, end up contributing in a non-negligible way to the write traffic to PM.

6.2. Write capacity and scalability

Our next study focuses on contrasting the available write capacity of transactions when using NV-HTM vs. PHTM and a pure HTM-based system not generating any additional write to ensure crash atomicity. To this end, we synthesized a workload in bank where the transaction size is increased iteratively up to the maximum capacity of current Intel's HTM implementation (i.e., 512 cache lines [43]).

Fig. 7 shows that, despite NV-HTM's maximum write capacity is lower than that of pure HTM, it is significantly larger than PHTM's. Indeed both NV-HTM and PHTM have to generate a log entry per transactional write. However, in both systems, log entries are 16-bytes long and, since logs are stored sequentially in memory, up to 4 log entries can fit a single cache line — which amortizes significantly the write capacity consumed to produce the log. However, PHTM further acquires a write lock per transactional write, and each of these locks is, with high probability, mapped to a different cache line. So, while on average NV-HTM consumes ≈0.25 additional cache lines per write, PHTM consumes on average ≈1.25 caches lines, i.e., ≈5× more.

Finally, we conduct a study aimed at assessing the scalability of NV-HTM's physical-clock based scheme [37]. The usage of a logical-clock in an HTM system is a source of "false" conflicts, which is translated in extra aborts. Given that DudeTM [39]'s approach uses a logical-clock, this experiment shows its overheads in an HTM system. To this end we use the bank benchmark to synthesize a conflict-free workload composed exclusively by short update transactions (emulating a transfer between a pair of bank accounts) and consider a NV-HTM's variant, noted NV-HTM^{LC}, in which transactions establish their serialization order in the log by increasing a single logical clock right before committing. The plots in Fig. 8 clearly highlight the inherent scalability limitations of approaches relying on a single logical clock, which generates abort rates above 60% when using 16 threads or more. Conversely, thanks to the use of physical clocks, NV-HTM avoids inducing any additional sources of conflicts among transactions, achieving almost linear scalability.

6.3. STAMP and TPC-C benchmarks

We now evaluate NV-HTM using more complex benchmarks that generate realistic workloads, namely the STAMP benchmark suite [7] and TPC-C [49].

STAMP. The results of the STAMP benchmark suite are reported in Fig. 9 and Table 2. The STAMP suite (from which we omit the Bayes benchmark, which is known to suffer from very high variance and yield unreliable results [16,51]) contains both workloads amenable to HTM, as well as workloads more favorable to STM, where most transactions exceed HTM's capacity or suffer from spurious conflicts due to HTM's coarser conflict detection

Table 2
Average writes/flushes per transaction to PM in STAMP.

	Genome		Intruder		Kmeans (low)		Vacation (low)		Labyrinth	
	Writes	Flushes	Writes	Flushes	Writes	Flushes	Writes	Flushes	Writes	Flushes
NV-HTM$_{NLP}$	2.326	1.870	2.915	2.396	27.00	8.249	7.799	3.439	15.30	5.461
NV-HTM$_{10\times}$	2.751	2.261	3.408	2.787	27.09	8.266	9.81	4.734	15.88	5.612
PHTM	7.176	1.321	10.93	2.172	45.99	5.000	33.29	6.621	80.39	16.50
PSTM	3.826	3.807	22.45	13.88	200.3	142.2	1.446	1.420	46.83	37.87
	SSCA2		Vacation (high)		Kmeans (high)		Yada		Average of all	
	Writes	Flushes	Writes	Flushes	Writes	Flushes	Writes	Flushes	Writes	Flushes
NV-HTM$_{NLP}$	4.000	3.000	12.62	4.601	27.00	8.249	18.36	6.016	13.04	4.809
NV-HTM$_{10\times}$	4.100	3.000	13.06	4.798	27.03	8.253	25.55	10.10	14.30	5.534
PHTM	19.00	3.999	55.01	10.84	45.99	5.000	77.44	14.91	41.69	7.374
PSTM	6.015	6.007	174.4	98.08	198.1	125.0	68.69	45.61	80.23	52.65

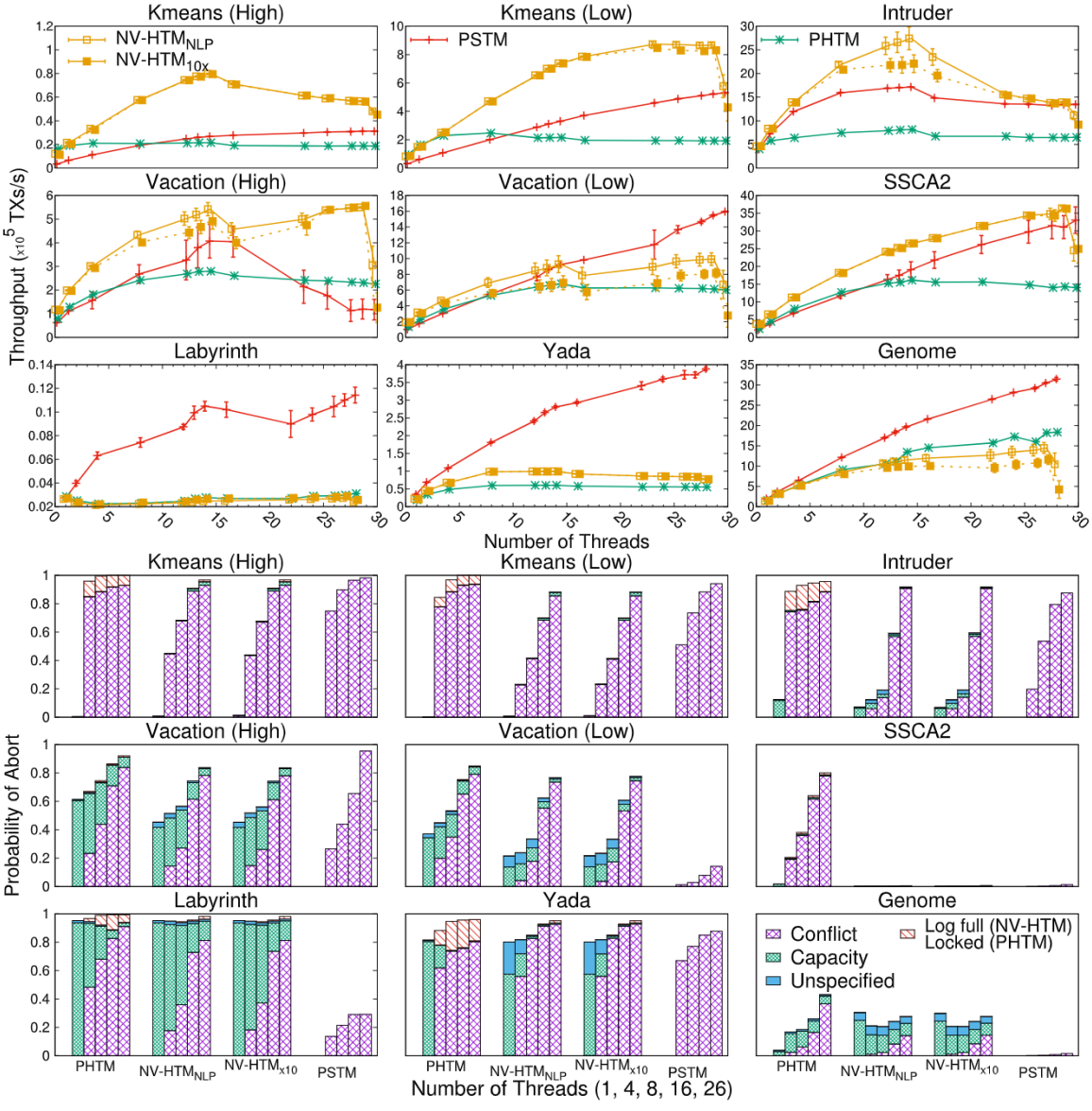

Fig. 9. STAMP benchmarks.

granularity. Labyrinth, Yada and Genome belong clearly to the second group of workloads, and, as such, PSTM has clearly an edge over both NV-HTM and PHTM. This is true also for Vacation (Low) at high thread counts. Yet, it is relevant to note that, even with these non-HTM-friendly workloads, NV-HTM achieves significant speed-ups over PHTM, up to $\approx 2\times$ in Yada. The only exception to this rule is represented by the Genome benchmark. By analyzing Genome more in detail we noticed that this benchmark issues

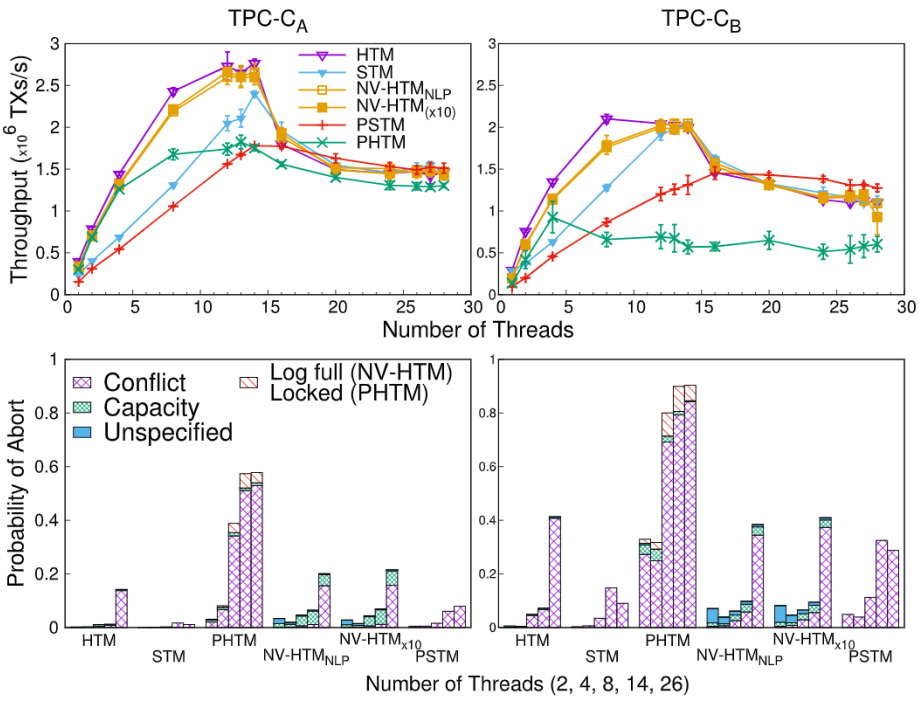

Fig. 10. TPC-C benchmark. TPC-C_A: $w = 10$, 20% Delivery, 65% Order-status and 15% payment; TPC-C_B, $w = 30$, 2% Delivery, 26% Order-status, 70% payment and 2% New-order.

a number of repeated writes to the same memory positions. As already mentioned (Section 6.2), PHTM uses a lock table to track ownership of cache lines and ensure that log flushes are ordered consistently with the commit order of transactions in hardware. This mechanism (which, as discussed, introduces non-negligible overhead) ensures that if a transaction (as it is the case for Genome) writes multiple times to the same memory position/cache line, a single log entry is actually produced and flushed to PM for all of these writes. NV-HTM, on the contrary, only filters duplicate writes/flushes to the same cache lines (via the background CP) if they are issued by different transactions. As such, in this workload, NV-HTM issues, on average, a larger number of flushes per committed transaction — which, we recall, incur a fixed penalty of 500 ns in our emulation of PM. We argue, though, that these results should be regarded as a lower bound on NV-HTM's performance since, in realistic PM settings, if the same cache line is flushed multiple times during the post commit log flushing phase, the full flushing penalty is expected to be incurred only once — unlike in our PM emulation.

Vacation and KMeans are more favorable to HTM, although with different characteristics: Vacation spends >90% of the time running transactions, whereas Kmeans spends >90% executing non-transactional code; further, in Kmeans, transactions are much less prone to incur capacity exceptions that in Vacation. In the light of these considerations, Fig. 9 suggests that the more favorable the workload characteristics are to HTM, the ampler are the speed-ups achievable by NV-HTM over both PHTM and PSTM, with gains in the peak throughput of 4× with respect to both solutions in Kmeans (High) (14 threads), ≈40% vs. PSTM and ≈2.5× vs. PHTM in Vacation (High).

We note that NV-HTM achieves almost indistinguishable performance in the scenario of high frequency of activation of the CP (10×) and in case the CP is never activated (NLP), confirming the efficiency of the BFC algorithm. Some exceptions are Vacation (Low), Genome and Intruder, where the slowdown is, in average, 20%, 16% and 7%, respectively.

Table 2 reports data on the number of write and flushes to PM. On average, NV-HTM $_{10\times}$ produces 2.92× less writes than PHTM and 5.61× less than PSTM, while only producing 8.8% more writes than NV-HTM $_{NLP}$ (Table 2 column *Average of All*), which confirms that the filtering technique at the core of BFC remains very effective also when applied to complex, realistic workloads.

A closer look to Table 2 reveals that PHTM issues a smaller number of flushes in Genome, Kmeans and Intruder. We argue that this is due to PHTM's ability of filtering duplicate flushes to the same cache line by the same transaction. Despite this, Genome (as already discussed) is the only benchmark where PHTM slightly outperforms NV-HTM, whereas in Kmeans and Intruder NV-HTM achieves up to 6× speedups. Such discrepancy is explained by analyzing the probability of abort in these benchmarks for PHTM and NV-HTM, which highlights how the software-based locking mechanism employed by PHTM makes this solution significantly more prone to contention. For example, in Intruder (4 threads), PHTM aborts 9× more than NV-HTM resulting in 2× slowdown.

6.3.1. TPC-C

We now evaluate NV-HTM using an in-memory porting [47] of TPC-C [49], a well known benchmark for OLTP systems. We report the corresponding results in Fig. 10 and Table 3.

The TPC-C benchmark specifies 5 transaction profiles, encompassing 2 read-only transactions (*Order Status* and *Stock Level*) and 3 update ones (*Delivery*, *New Order*, and *Payment*). This benchmark defines a parameter, i.e., the number of warehouses (w), which allows for controlling the contention level among transactions — larger w values yielding lower conflict probability. As the focus of this work is on HTM, we configure the benchmark not to generate a transaction profile (named *Stock level*), which generates a very long read-only transaction that deterministically exceeds HTM capacity, forcing the acquisition of the pessimistic fallback path and hindering scalability of HTM-based solutions. We considered two workloads, referred to as TPC-C_A and TPC-C_B. TPC-C_A is a read-intensive workload (65% read-only transactions) that generates a mild contention level ($w = 10$), whereas TPC-C_B is a write-intensive workload (26% read-only transactions) that

Table 3
Average writes/flushes to PM per transaction in TPCC.

	TPC-C_{write}		TPC-C_{read}	
	Writes	Flushes	Writes	Flushes
NV-HTM$_{NLP}$	4.7	2.2	1.6	1.2
NV-HTM$_{10\times}$	4.7	2.5	1.6	1.2
PHTM	16	3.1	4.0	0.63
PSTM	6.6	6.4	2.7	2.7

generates a slightly higher contention level w.r.t. TPC-C_A ($w = 30$).

The evaluated solutions are the same as in STAMP, with the addition of pure HTM and STM operating on plain RAM (i.e., no PM).

As Fig. 10 shows, not only does NV-HTM outperform both PHTM and PSTM. NV-HTM's throughput is also only 10% worse than a RAM-based HTM solution in TPC-C_B and 5% in TPC-C_A (average of all data points). Further, the comparison of NV-HTM$_{10\times}$'s and NV-HTM$_{NLP}$'s performance shows that the overhead imposed by the background log pruning process is, in this workload, negligible.

Regarding the amount of writes and flushes to PM, the conclusions are similar to the ones already drawn while analyzing STAMP. Table 3 shows that NV-HTM incurs approximately 3×/50% less writes to PM than PHTM/PSTM, respectively. In TPC-C_B, NV-HTM also requires the least flushes to PM (2.5 vs. 3.1/6.4 for PHTM/PSTM, respectively), due to its ability to filter duplicate updates to the same cache lines across transactions.

Overall, these results showcase the efficiency of NV-HTM even when faced with demanding real world workloads.

7. Limitations and discussion

This section aims at critically analyzing the proposed solution and at shedding lights on the implications, and possible drawbacks, of its design. In particular, our discussion focuses on analyzing two main design choices of NV-HTM, namely: (i) its reliance on a background log-pruning process and (ii) the technique it employs to establish a total order on the commit events of update transactions.

Impact of the concurrent log-pruning process. Differently from PSTM [50] and PHTM [4], NV-HTM does not prune the log entries generated by a committed update transaction after this commits. Conversely, it relies on a background Checkpointing Process (CP) to replay, in an asynchronous fashion, the logs to PM and prune them.

As also discussed in Section 6.1, in workloads that generate an intense write traffic and whose performance is memory bound, the concurrent execution of the background CP can impose a non-negligible overhead if: (i) there is not enough log capacity to accommodate incoming bursts of update transactions, triggering frequent activation of the CP; (ii) the latency of writing to PM associated with the scarcity of duplicate accesses does not compensate the efforts of filtering them; or, (iii) the CP shares its physical core (and/or other resources) with one or more working thread(s), thus, slowing down the CP. Point (i) depends on how much PM the application reserves for the logs; point (ii) could be tackled by adaptively disabling duplicate filtering; and point (iii) would require the application to not oversubscribe the cores reserved for the CP.

Totally ordering transactions in the log. NV-HTM allows threads to flush the log entries that describe the updates performed by a transaction in a fully concurrent fashion. However, in order to impose a total order on the commit events of the (update) transactions recorded in the log, NV-HTM serializes the flushing of the transactions' final commit marker, i.e., at most one thread at a time can flush its commit marker to PM.

Conversely, PSTM [50] and PHTM [4] impose only a partial order on the transactions registered in the persistent log. This is made possible via the use of an additional locking mechanism, which prevents conflicting transactions from flushing their logs concurrently. This way, correctness of the replay phase can be preserved, while allowing concurrent access to the log by non-conflicting transactions.

We argue to be in presence of a trade-off. If, on the one hand, the use of locking enables a fully concurrent access to the log by non-conflicting transactions (unlike with NV-HTM's commit logic), on the other hand, it also introduces significant overhead, especially in the context of HTM-based solutions. As discussed in Section 6.2, in fact, the software instrumentation required to implement a locking scheme reduces significantly the effective write capacity of transactions executing in hardware. Further, both in PHTM and PSTM, since locks are acquired during transaction's execution and maintained till all of the transaction's log entries have been flushed to PM, the time during which transactions can be subject to suffer from lock contention (and a consequent restart) is significantly larger than for the case of NV-HTM. In fact, NV-HTM fully removes the latency of log flushes from the critical path of execution of transactions by postponing them to after their commit.

8. Conclusion and future work

This paper presents NV-HTM, system capable of combining (unmodified) commodity HTM with PM. By coupling a novel *asynchronous logging scheme* with a background *log pruning* process, NV-HTM avoids the need for flushing cache lines within transactions. This is not only key to ensure the interoperability between HTM and PM, but brings also relevant benefits in terms of both performance and reduction of PM write wearing: on one hand, it allows applications to execute on a faster, volatile working image; on the other hand, by offloading the overhead of synchronizing application's state to PM to a background process, it enables the opportunity to filter duplicate writes across transactions, significantly reducing the write/flush traffic to PM.

Our experimental evaluation shows that NV-HTM can achieve strong gains, in terms of throughput (up to 10×) and PM write wearing reduction (up to 11.6× less flushes to PM), even when compared to existing solutions that demand custom hardware.

The design of NV-HTM opens a number of interesting research avenues, which we intend to pursue in our future work. In particular, we argue that the ability of NV-HTM to define a total order over the commit events of HTM transactions – which NV-HTM exploits to ensure the correctness of the checkpointing process – can be valuable in at least two contexts that are outside of the scope of PM, namely debugging and replication. The former is notoriously hard in HTM, and NV-HTM's ability to pinpoint and deterministically reproduce the serialization order of HTM transactions represents a valuable tool for programmers to identify possible issues in their applications. The latter often relies on primary-based approaches where a privileged/master process is responsible for establishing a total order on update operations, which has then to be deterministically applied by a set of backup/slave processes. NV-HTM's ability to produce a totally order transactions' log comes in clearly very handy in this context. Further, in scenarios where one wants to employ distributed replication based on HTM and PM, the ability of NV-HTM to filter duplicate writes across transactions could be exploited both to reduce the communication overhead and the cost of replay at the backup processes.

Acknowledgments

We would like to thank all the insightful reviews from the IPDPS Program Committee and from the JPDC reviewers. We also would like to thank our funding agency Fundação para a Ciência e Tecnologia which funded the projects UID/CEC/50021/2019 and PTDC/EEISCR/1743/2014.

References

[1] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, I. Watson, in: P. Stenström (Ed.), Transactions on High-performance Embedded Architectures and Compilers III, Springer, 2011, pp. 236–255.
[2] J. Arulraj, A. Pavlo, S.R. Dulloor, Let's talk about storage: Recovery methods for non-volatile memory database systems, in: SIGMOD, 2015, pp. 707–722.
[3] H. Avni, T. Brown, PHyTM: Persistent hybrid transactional memory, in: VLDB, Vol. 10, 2016, pp. 409–420.
[4] H. Avni, E. Levy, M. Avi, Hardware transactions in nonvolatile memory, in: DISC 2015. Lecture Notes in Computer Science, Vol. 9363, Springer, Berlin, Heidelberg, 2015, pp. 617–630.
[5] K. Bailey, L. Ceze, S.D. Gribble, H.M. Levy, Operating system implications of fast, cheap, non-volatile memory, in: HotOS, 2011.
[6] P. Bergner, A.S. Houfater, M. Kandeasamy, D. Wendt, S. Warrier, J. Wang, B.K. Smith, W. Schmidt, B. Schmidt, S. Munroe, T. Magno, A. Mericas, M. Oliveira, B. Hall, Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8, IBM Redbooks, 2015.
[7] C. Cao Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: IISWC, 2008.
[8] C. Cao Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: IISWC, 2008.
[9] D. Castro, P. Romano, J. Barreto, Hardware transactional memory meets memory persistency, IPDPS (2018) 368–377.
[10] D.R. Chakrabarti, H.-J. Boehm, K. Bhandari, Atlas: Leveraging locks for non-volatile memory consistency, in: OOSPLA, 49 (10) (2014) 433–452.
[11] J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, S. Swanson, NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories, in: ASPLOS, Vol. 47, 2011.
[12] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, D. Coetzee, Better I/O through byte-addressable, persistent memory, in: SOSP, 2009, pp. 133–147.
[13] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, D. Coetzee, Better i/o through byte-addressable, persistent memory, in: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, in: SOSP '09, ACM, New York, NY, USA, 2009, pp. 133–146, http://dx.doi.org/10.1145/1629575.1629589, URL http://doi.acm.org/10.1145/1629575.1629589.
[14] D. Dice, Y. Lev, M. Moir, D. Nussbaum, Early experience with a commercial hardware transactional memory implementation, in: ASPLOS, Vol. 44, 2009.
[15] D. Dice, N. Shavit, Understanding tradeoffs in software transactional memory, in: CGO, 2007.
[16] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, P. Romano, Proteustm: Abstraction meets performance in transactional memory, in: ASPLOS, ACM, 2016, pp. 757–771.
[17] D. Didona, P. Felber, D. Harmanci, P. Romano, J. Schenker, Identifying the optimal level of parallelism in transactional memory applications, Computing 97 (9) (2015) 939–959.
[18] N. Diegues, P. Romano, L. Rodrigues, Virtues and limitations of commodity hardware transactional memory, in: PACT, 2014, pp. 3–14.
[19] X. Dong, Y. Xie, AdaMS: Adaptive MLC/SLC phase-change memory design for file storage, in: ASP-DAC, 2011, pp. 31–36.
[20] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, J. Jackson, System software for persistent memory, in: EuroSys, pp. 1–15.
[21] P. Felber, C. Fetzer, P. Marlier, T. Riegel, Time-based software transactional memory, IEEE TPDS 21 (12) (2010) 1793–1807.
[22] E.R. Giles, K. Doshi, P. Varman, SoftWrAP: A lightweight framework for transactional support of storage class memory, in: MSST, 2015, pp. 1–14.
[23] E. Giles, K. Doshi, P. Varman, Continuous checkpointing of HTM Transactions in NVM, ISMM (2017) 70–81.
[24] E. Giles, K. Doshi, P. Varman, Brief announcement: Hardware transactional persistent memory, in: SPAA, ACM Press, 2018, pp. 227–230.
[25] M. Herlihy, J.E.B. Moss, Transactional memory: Architectural support for lock-free data structures, in: ISCA, 1993, pp. 289–300.
[26] J. Huang, K. Schwan, M.K. Qureshi, NVRAM-aware logging in transaction systems, in: VLDB, Vol. 8, 2014, pp. 389–400.
[27] I. Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, 2010.
[28] Intel Corporation, Desktop 4th Generation Intel Core Processor Family (Revision 028), Tech. rep., Intel Corporation (2015).
[29] J. Izraelevitz, T. Kelly, A. Kolli, Failure-atomic persistent memory updates via JUSTDO logging, in: ASPLOS, 2016, pp. 427–442.
[30] B. Jacob, T. Mudge, A look at several memory management units, TLB-refill mechanisms, and page table organizations, in: ASPLOS, 1998, pp. 295–306.
[31] A. Joshi, V. Nagarajan, M. Cintra, S. Viglas, DHTM: Durable hardware transactional memory, in: ISCA, Vol. 7, IEEE, 2018, pp. 452–465.
[32] S. Kashyap, C. Min, K. Kim, T. Kim, A scalable ordering primitive for multicore machines, in: EuroSys, 2018.
[33] W.-H. Kim, J. Kim, W. Baek, B. Nam, Y. Won, NVWAL: Exploiting NVRAM in write-ahead logging, in: ASPLOS, Vol. 1, 2016, pp. 385–398.
[34] A. Kolli, S. Pelley, A. Saidi, P.M. Chen, T.F. Wenisch, High-performance transactions for persistent memories, in: ASPLOS, 2016, pp. 399–411.
[35] M. Kryder, C. Kim, After hard drives-what comes next? IEEE Trans. Magn. 45 (10) (2009) 3406–3413.
[36] E.V. Kumar, clock() or gettimeofday() or ippGetCpuClocks()? (last access: 2017-5-15) (2010). URL https://software.intel.com/en-us/articles/best-timing-function-for-measuring-ipp-api-timing/.
[37] Y. Liu, J. Gottschlich, G. Pokam, M. Spear, TSXProf: Profiling hardware transactions, in: PACT, 2015, pp. 75–86.
[38] Q. Liu, P. Varman, Ouroboros wear leveling for NVRAM using hierarchical block migration, ACM Trans. Storage 13 (4) (2017) 1–30.
[39] M. Liu, M. Zhang, K. Chen, X. Qian, DudeTM: Building durable transactions with decoupling for persistent memory, in: ASPLOS, 2017, pp. 329–343.
[40] Y. Lu, J. Shu, L. Sun, Blurred persistence in transactional persistent memory, in: MSST, 2015, pp. 1–13.
[41] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, S. Swanson, Atomic in-place updates for non-volatile main memories with Kamino-Tx, in: EuroSys, 2017, pp. 499–512.
[42] S. Mittal, J.S. Vetter, A survey of software techniques for using non-volatile memories for storage and main memory systems, IEEE TPDS (2015) 1537–1550.
[43] T. Nakaike, R. Odaira, M. Gaudet, M.M. Michael, H. Tomari, Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8, in: ISCA, 2015, pp. 144–157.
[44] S. Park, T. Kelly, K. Shen, Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data, in: EuroSys, 2013, pp. 225–238.
[45] R. Ramakrishnan, J. Gehrke, Database Management Systems, third ed., McGraw-Hill, Inc., New York, NY, USA, 2003.
[46] D. Rughetti, P. Romano, F. Quaglia, B. Ciciani, Automatic tuning of the parallelism degree in hardware transactional memory, in: Euro-Par, 2014.
[47] M. Stonebraker, S. Madden, D.J. Abadi, S. Harizopoulos, N. Hachem, P. Helland, The end of an architectural era: (it's time for a complete rewrite), in: VLDB, VLDB Endowment, 2007, pp. 1150–1160.
[48] The Linux Foundation Direct Access for files (last access: 2017-06-21). URL https://www.kernel.org/doc/Documentation/filesystems/dax.txt.
[49] Transaction Processing Performance Council, TPC Benchmark C, Standard Specification, Revision 5.11 (Feb. 2010).
[50] H. Volos, A.J. Tack, M.M. Swift, Mnemosyne: Lightweight persistent memory, in: ASPLOS, 2011, pp. 91–104.
[51] Q. Wang, S. Kulkarni, J. Cavazos, M. Spear, A transactional memory with automatic performance tuning, ACM Trans. Archit. Code Optim. 8 (4) (2012) 1–23.
[52] Z. Wang, H. Yi, R. Liu, M. Dong, H. Chen, Persistent transactional memory, IEEE Comput. Archit. Lett. 14 (1) (2015) 58–61.
[53] J. Xu, S. Swanson, NOVA: A log-structured file system for hybrid volatile/non-volatile main memories, in: FAST, 2016, pp. 323–338.
[54] X. Yuan, C. Wu, Z. Wang, J. Li, P.C. Yew, J. Huang, X. Feng, Y. Lan, Y. Chen, Y. Guan, ReCBuLC: reproducing concurrency bugs using local clocks, in: ICSE, 2015.
[55] Z. Zhang, D. Feng, Z. Tan, J. Chen, W. Zhou, L.T. Yang, File aware wear leveling for PCM-based mobile consumer electronics, in: HPCC/SmartCity/DSS, IEEE, 2017, pp. 555–562.

Daniel Castro received the Master degree in Engineering Systems and Computer Engineering from Instituto Superior Técnico (University of Lisbon). He is currently a PhD student at Instituto Superior Técnico (University of Lisbon) and INESC-ID. His research interests are in transactional memory, emergent non-volatile memory, heterogeneous computing and performance systems modeling.

Paolo Romano received his PhD from Rome University "Sapienza" (2007) and his Master degree summa cum laude from Rome University "Tor Vergata" (2002). He is currently an Associate Professor at Técnico (ULisboa) and a Senior Researcher at INESC-ID. His research interests include parallel and distributed computing, dependability, autonomic systems, performability modeling and evaluation, data management in large scale systems, cloud and high performance computing.

João Barreto holds a PhD from the Univ. of Lisbon. He is with INESC-ID and Univ. Of Lisbon where he is an assistant professor in the areas of Architectures, Operating Systems and Distributed Systems. His research interests include parallel programming, consistency and replication, data deduplication, mobile sensing.