Hardware Transactional Memory meets Memory Persistency

Daniel Castro*
daniel.castro@ist.utl.pt

Paolo Romano*
romano@inesc-id.pt

João Barreto*
joao.barreto@tecnico.ulisboa.pt

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

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 relies on 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 levelling 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.**

Index Terms—**transaction, memory, persistent, hardware, system**

I. Introduction

Over the last years, two game shifting advancements have shaken the ground of memory technology: Hardware Transactional Memory (HTM) and Persistent Memory (PM).

On the one hand, HTM [23], [33], [34], 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 [22], as well as to significantly simplify the development of concurrent applications with respect to conventional lock-based schemes [33].

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 [36]). 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 DIMMs will still suffer from write wearing and have much higher write latencies when compared to DRAM.

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 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 [8], [13], [42]. As such, upon failure, only a subset of the committed writes to PM may have been persisted — failing to guarantee crash-atomicity [12].

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 [34], 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 leverage on 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

This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2013 and PTDC/EEIS-CR/1743/2014.

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 [13] as well as relying on ad-hoc hardware mechanisms [12]. The results show that NV-HTM can achieve up to **10**$\times$ better performance and reduce number of flushes to PM by a factor **11.6**$\times$ with respect to state-of-the-art solutions.

II. 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 [14], [16], [25], [26], [28], [42].

An alternative approach exposes PM as persistent heaps that applications can access directly via memory loads and stores [13], [15]. 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 [8], [18], 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 [13] and NV-heaps [15]. 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., [5], [6], [17], [30].

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 [22].

Avni et al. [12] and Wang et al. [31] 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 [1], which extends prior work [12] 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 [39]. Furthermore, supporting PM as a first-class citizen has produced initial steps towards novel proposals to redesign fundamental operating system (OS) mechanisms [19]. 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 several recent works. In the PM domain, examples include AdaMS [4] and Kamino-Tx [6]. — none of them, though, targets the problem of enabling the use of HTM over PM. Concurrently with our research, two works have very recently addressed the problem of supporting PM in commodity HTM implementations, [21], [41]. DudeTM [21] provides a generic checkpointing mechanism, which can be coupled with both software and hardware TM implementation. 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 VI-B. Giles et al. [41] avoid this issue, by using physical clocks to order transactions in the log, analogously to what NV-HTM does. However, unlike NV-HTM, it requires to instrument read access, which introduces significant overheads (and arguably defeats the purpose of HTM). Further, differently from NV-HTM, the work by Gilles et al. [41] does not employ any log filtering techniques to extend the life expectancy of PM.

Finally, NV-HTM builds on the literature in DBMS [40], which developed a large number of 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.

III. Background on HTM

Existing HTM systems come with different flavours 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 minimalistic 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.

IV. System Architecture

Analogously to other recent software libraries for building PM-based applications, e.g., [5], [6], [13], [17], 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 inizialization, 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 [32], 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 Figure 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

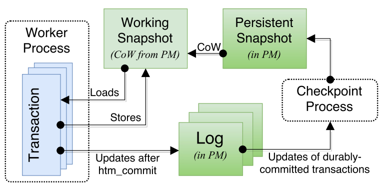

Fig. 1: High level architecture of NV-HTM.

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 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 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 [35]; 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]

V. Implementation

This section presents NV-HTM's design and implementation. Section V-A analyzes transaction processing and log management by the WP, Section V-B focuses on the CP and Section V-C discusses correctness.

A. 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: 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 entry 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: 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 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 [3].

Transaction begin. 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 [2]) 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. 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 (not reported in the pseudo-code for space constraints), 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.

[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 PM 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.

[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 [9] and unnecessary activations of the SGL path.

Algorithm 1 WP: transaction processing at thread t

```
1: Shared variables:
2:     $log[N]$                                        ▷ One log per thread, stored in PM
3:     $ts[N] \leftarrow \{+\infty, \dots, +\infty\}$
                        ▷ Per-thread timestamp of active tx; $+\infty$ 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:     $isRO \leftarrow$TRUE
9:     $ts[t] \leftarrow$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:    $isRO \leftarrow$FALSE
14:    if logCheckSpace ($log[t]$)=FULL then
15:        ABORT(LOG_FULL)
16:    $*addr \leftarrow value$                              ▷ Write to working snapshot
17:    $log[t]$.append($< addr, value >$)
18: function ABORT(abort_code)
19:    htm_abort($abort\_code$)
20:    $ts[t] \leftarrow +\infty$
21: function COMMIT
22:    if $isRo$ then                               ▷ Commit logic for read-only txs
23:        htm_commit()
24:        $ts[t] \leftarrow +\infty$                  ▷ Others do not need to wait for RO tx
25:        WAITCOMMIT()
26:    else                                           ▷ Commit logic for update txs
27:        $locTs \leftarrow$ READTS()
28:        htm_commit()
29:        $ts[t] \leftarrow locTs$
30:        logFlush($log[t]$)                             ▷ Flush current log entries
31:        WAITCOMMIT()
32:        $log[t]$.append($<$ COMMIT$, locTS >$)
33:        $log[t].endP \leftarrow locEndP$
34:        logFlush($log[t]$)                        ▷ Flush commit marker and $endP$
35:        $ts[t] \leftarrow +\infty$
36: function WAITCOMMIT
37:    for all $t^* \in [1, N]$ $s.t.$ $t^* \neq t$ do
38:        wait until $ts[t^*] > ts[t]$
```

Commit. 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 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.

B. Log checkpointing

Unlike existing solutions [1], [12] 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 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).

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 a large number of 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 first step of BFC is to take an atomic snapshot of the end pointers of each per-thread log — which can be efficiently achieved via a read-only hardware transaction.

Next, BFC iteratively traverses the per-thread logs in anti-commit timestamp order and analyzes the log entries of each durably committed transaction. 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 $filterMap$ indexed by cache line address, whose value stores a bitmap encoding which "positions" (*CLPos*) of that cache line have been already updated (due to a more recent transaction) during the current checkpointing instance. Tracking updates at the granularity of 8 bytes, the *CLPos* can be compactly encoded using a single byte (i.e., cache line width of 64-bytes).

More precisely, each iteration proceeds as follows. Firstly, BFC 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 transaction in each per-thread log. Then, for each logged update of T_{lat}, the $filterMap$ is first consulted to determine if the corresponding address has already been encountered while scanning the log — 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.

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. 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 DBMS literature [40]: 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. 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, 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 allow 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* wih 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 the 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).

C. 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 any develop any read-from or write-write dependency, either directly or indirectly, even if their HTM and serialization orders are distinct, they will 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. A similar approach is taken by TSXProf [29] to replay HTM transactions.

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 get high-resolution CPU timing information with a low overhead (e.g., `RDTSC*` in Intel processors).

Most modern processors (e.g., all Intel CPUs supporting *Invariant TSC* since Nehalem family [2], [37]) ensure that all cores (including across different sockets), observe the TSC ticking at the same rate, which yields perfect TSC synchronization. Also, TSC grows monotonically and, for simplicity, we do not tackle the case where the physical clock overflows and wraps around. Let us now consider such assumptions.

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.

We start by proving that, if two transactions T and T' are dependent, if $T' \xrightarrow{LOG} T$, then $T' \xrightarrow{HTM} T$. T and T' are associated with their timestamps $T.ts$ and $T'.ts$ (resp.), taken within the transaction after all accesses. Additionally, $T.commit$ and $T'.commit$ are the instant in real time where T and T' commit (resp.).

Let us assume that $T' \xrightarrow{LOG} T$ and that both transactions have a read-from or write-write dependency. This means that there are two accesses (from T and T') to a common location (i.e., *op* and *op'*, resp.). *op* occurs at real time $ts1$ and op' at real time $ts2$. Given that *op* conflicts with *op'*, if both transactions overlap in real time, then the HTM implementation decides to kill either T or T'.

By contradiction, assume that $T.ts < T'.ts$. Given that $T' \xrightarrow{HTM} T$ then $T'.commit < T.commit$, which implies $T.ts < T'.ts < T'.commit < T.commit$, because $T.ts < T.commit$ and $T'.ts < T'.commit$ (TSC assumption). Given that $op' < T'.ts$ and $op < T.ts$, then we get $op < T.ts < op' < T'.ts < T'.commit < T.commit$, which is absurd because despite dependent/conflicting, they overlap.

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 are all other transactions T' it depends upon.

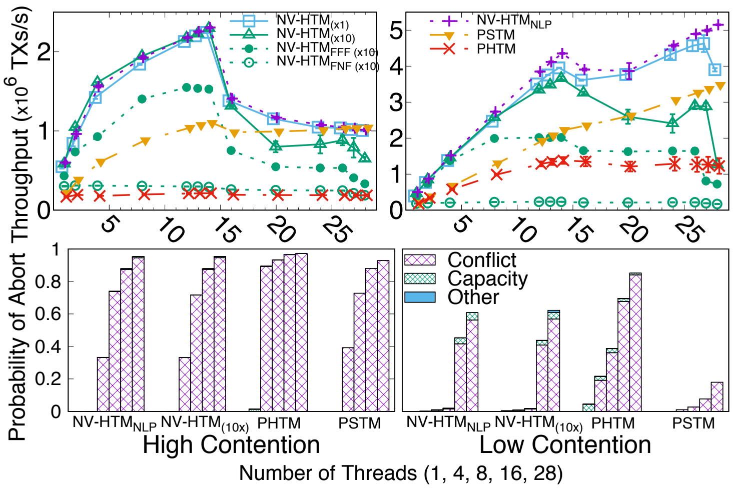

Fig. 2: Performance with different log pruning frequencies.

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 $ts^* < T.ts$. We have shown that, with perfectly synchronized TSCs, if there exists a non-durably committed transaction, T', such that $T' \xrightarrow{HTM} T$, then $T.ts > T'.ts$. Obviously, $T'.ts$ must be larger than the start timestamp of T'. 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 from any non-durably committed transaction.

Perfectly synchronized TSCs, however, are not necessarily guaranteed on older CPUs or in a multi-socket machine. In such scenarios, the most that can be assumed is that TSCs grow monotonically and synchronized within a given upper bound, Δ. In this scenario, it is no longer be possible to use a single time source when reasoning on the timestamps obtained by different threads. Instead, correctness is achieved by injecting a delay equal to Δ after reading the TSC (Alg. 1, line 27) and before requesting the HTM to commit (line 28). For space restrictions, we omit the proof for this case.

VI. Experimental Evaluation

This section presents an extensive experimental evaluation aimed at comparing NV-HTM[3] with 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 [13]'s algorithm, which we re-implemented on top of TinySTM [24] using in-place updates to minimize read-instrumentation costs (analogously to what was done by Avni et al. [12]). 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

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

TABLE I: Writes/flushes to PM.

	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

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 [12], which, as discussed in Section II, assumes non-standard hardware mechanisms to ensure that the log is transparently and atomically flushed to PM 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.

All tests were conducted on an Intel Xeon CPU E5-2648L v4 @ 1.80GHz with 14 physical cores and 28 hardware threads in hyper-threading mode. The machine is equipped with 32GB of RAM and runs Ubuntu Server 16.04.2LTS (kernel version 4.4.0-57). As PM is still not widely available, we emulate it by injecting a 500ns latency upon each flush operation, analogously to previous works in the area [12], [39].

We use both standard benchmarks, i.e., the STAMP suite [38], 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.

A. *Impact of checkpointing*

The first aspect we evaluate is the impact, on both performance and 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 16K 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 40MB 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 Figure 2, 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

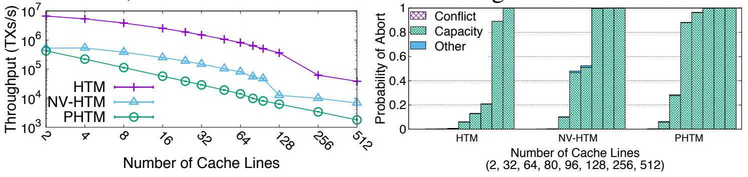

Fig. 3: Evaluating the maximum write capacity of NV-HTM.

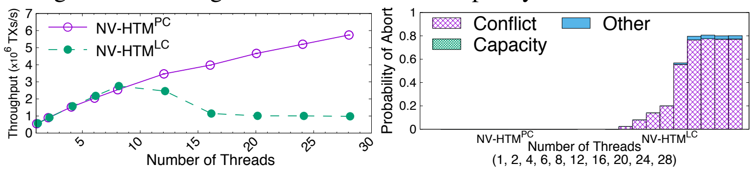

Fig. 4: Contrasting the scalability of NV-HTM with a variant using logical clock to serialize transactions (NV-HTM^{LC}).

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 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 [22] 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 [10], [11], [20].

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

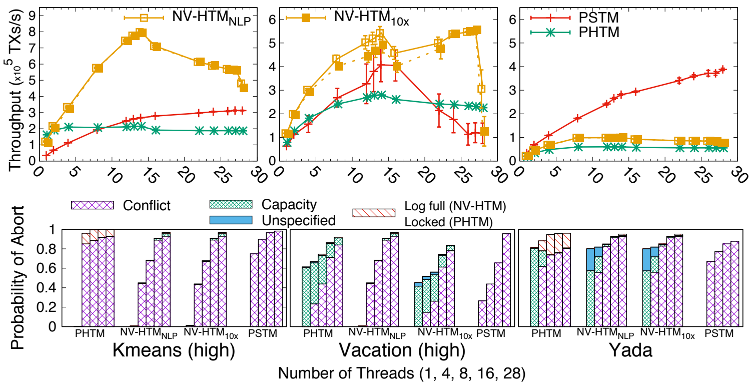

Fig. 5: Subset of the STAMP benchmarks.

B. *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 [27]).

Figure 3 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 [29]. 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 [21]'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 Figure 4 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.

C. *STAMP benchmarks*

We now evaluate NV-HTM using more complex benchmarks that realistic workloads, namely the STAMP benchmark

TABLE II: Writes/flushes to PM in STAMP.

	Vacation (high)		Kmeans (high)		Yada	
	writes	flushes	writes	flushes	writes	flushes
NV-HTM$_{NLP}$	12.62	4.601	27.00	8.249	18.36	6.016
NV-HTM$_{10\times}$	13.06	4.798	27.03	8.253	25.55	10.10
PHTM	55.01	10.84	45.99	5.000	77.44	14.91
PSTM	174.4	98.08	198.1	125.0	68.69	45.61

suite [7]. For space constraints we can only report the results for a subset of the benchmarks, namely Kmeans, Vacation and Yada, which are however representative of the main performance trends exhibited also by the remaining benchmarks.

This benchmark contains both workloads amenable to HTM, as well as workloads more favourable to STM, where most transactions exceed HTM's capacity or suffer from spurious conflicts due to HTM's coarser conflict detection granularity. Yada belongs clearly to the second of workloads, and, as such, PSTM has clearly an edge over both NV-HTM and PHTM (still, NV-HTM achieves $\approx 2\times$ speed-ups over PHTM).

Vacation and KMeans are more favourable 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, Figure 5 suggests that the more favourable 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 $4\times$ with respect to both solutions in Kmeans (14 threads), $\approx 40\%$ vs PSTM and $\approx 2.5\times$ vs PHTM in Vacation.

We note that NV-HTM achieves almost indistinguishable performance in the scenario of high frequency of activation of the CP ($10\times$) and in case the CP is never activated (NLP), confirming the efficiency of the BFC algorithm.

Finally Table II reports data on the number of write and flushes to PM. In average, NV-HTM$_{10\times}$ produces $2.72\times$ less writes than PHTM and $6.72\times$ less than PSTM, while only producing 13% more writes than NV-HTM$_{NLP}$, which confirms that the filtering technique at the core of BFC remains very effective also when applied to complex, realistic workloads.

VII. Conclusion

This presents NV-HTM, system capable of combining (unmodified) commodity HTM with PM. Our experimental evaluation shows that NV-HTM can achieve strong gains, in terms of throughput and PM wearing reduction, even when compared to existing solutions that demand custom hardware.

Acknowledgments

We would like to thank all the insightful reviews from the IPDPS Program Committee.

References

[1] H. Avni and T. Brown. PHyTM: Persistent Hybrid Transactional Memory. *VLDB'16.*

[2] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2010.

[3] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. CGO'07.

[4] X. Dong and Y. Xie. AdaMS: Adaptive MLC/SLC phase-change memory design for file storage. *ASP-DAC'11.*

[5] A. Kolli et al. High-Performance Transactions for Persistent Memories. *ASPLOS'16.*

[6] A. Memaripour et al. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. *EuroSys'17.*

[7] C. Cao Minh et al. STAMP: Stanford Transactional Applications for Multi-Processing. *IISWC'08.*

[8] D. Chakrabarti et al. Atlas: Leveraging Locks for Non-volatile Memory Consistency. *OOSPLA'14.*

[9] D. Dice et al. Applications of the Adaptive Transactional Memory Test Platform. *3rd ACM SIGPLAN Workshop on Transactional Computing.*

[10] D. Didona et al. Identifying the Optimal Level of Parallelism in Transactional Memory Applications. *Networked Systems*, 2013.

[11] D. Rughetti et al. Analytical/ML mixed approach for concurrency regulation in software transactional memory. *CCGrid'14.*

[12] H. Avni et al. Hardware Transactions in Nonvolatile Memory. *DISC'15.*

[13] H. Volos et al. Mnemosyne: Lightweight Persistent Memory. *ASPLOS'11.*

[14] J. Arulraj et al. Let's talk about storage: Recovery methods for non-volatile memory database systems. In *SIGMOD'15.*

[15] J. Coburn et al. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. *ASPLOS'11*, 47(4).

[16] J. Condit et al. Better I/O through byte-addressable, persistent memory. In *SOSP '09.*

[17] J. Huang et al. NVRAM-aware logging in transaction systems. *VLDB'14.*

[18] J. Izraelevitz et al. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. *ASPLOS'16.*

[19] K. Bailey et al. Operating System Implications of Fast, Cheap, Non-Volatile Memory. *HotOS'13*, 2011.

[20] M. Ansari et al. Transactions on high-performance embedded architectures and compilers iii. Springer-Verlag, 2011.

[21] M. Liu et al. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. *ASPLOS'17.*

[22] N. Diegues et al. Virtues and Limitations of Commodity Hardware Transactional Memory. *PACT'14.*

[23] P. Bergner et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8.* IBM Redbooks, 2015.

[24] P. Felber et al. Time-Based Software Transactional Memory. *IEEE TPDS*, 2010.

[25] S. Park et al. Failure-Atomic msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. *EuroSys '13.*

[26] S. R. Dulloor et al. System software for persistent memory. *EuroSys'14.*

[27] T. Nakaike et al. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *ISCA'15.*

[28] W. Kim et al. NVWAL: Exploiting NVRAM in Write-Ahead Logging. *ASPLOS'16*, 1(212).

[29] Y. Liu et al. TSXProf: Profiling Hardware Transactions. *PACT'15.*

[30] Y. Lu et al. Blurred persistence in transactional persistent memory. *IEEE Symposium on Mass Storage Systems and Technologies*, 2015.

[31] Z. Wang et al. Persistent transactional memory. *IEEE Computer Architecture Letters*, 14(1), 2015.

[32] The Linux Foundation. Direct Access for files (last access: 2017-06-21). *https://www.kernel.org/doc/Documentation/filesystems/dax.txt.*

[33] M. Herlihy and J. Moss. Transactional memory. *ISCA'93*, 21(2).

[34] Intel Corporation. Desktop 4th Generation Intel Core Processor Family (Revision 028). Technical report, Intel Corporation, 2015.

[35] B. Jacob and T. Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. ASPLOS VIII, 1998.

[36] M. Kryder and C. Kim. After hard drives-what comes next? *IEEE Transactions on Magnetics*, 45(10), 2009.

[37] E. K. Vipin Kumar. https://software.intel.com/en-us/articles/best-timing-function-for-measuring-ipp-api-timing/, 2010.

[38] C. Cao Minh. STAMP: stanford transactional applications for multi-processing. In *IISWC'08.*

[39] S. Mittal and J. Vetter. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE TPDS*, 2015.

[40] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, Inc., New York, NY, USA, 3rd edition, 2003.

[41] E.Giles et al.. Continuous Checkpointing of htm Transactions in nvm. *ISMM'17.*

[42] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. *FAST'16.*