SPHT: Scalable Persistent Hardware Transactions

Daniel Castro, *INESC-ID & Instituto Superior Técnico;* Alexandro Baldassin, *UNESP - Universidade Estadual Paulista;* João Barreto and Paolo Romano, *INESC-ID & Instituto Superior Técnico*

https://www.usenix.org/conference/fast21/presentation/castro

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings of the 19th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

SPHT: Scalable Persistent Hardware Transactions

Daniel Castro*, Alexandro Baldassin†, João Barreto*, Paolo Romano*
*INESC-ID & Instituto Superior Técnico †UNESP - Universidade Estadual Paulista

Abstract

With the emergence of byte-addressable Persistent Memory (PM), a number of works have recently addressed the problem of how to implement persistent transactional memory using off-the-shelf hardware transactional memory systems.

Using Intel Optane DC PM, we show, for the first time in the literature, experimental results highlighting several scalability bottlenecks of state of the art approaches, which so far have only been evaluated via PM emulation.

We tackle these limitations by proposing SPHT (Scalable Persistent Hardware Transactions), an innovative Persistent Transactional Memory that exploits a set of novel mechanisms aimed at enhancing scalability both during transaction processing and recovery. We show that SPHT enhances throughput by up to 2.6× on STAMP and achieves speedups of up to 2.8× in the log replay phase vs. state of the art solutions.

1 Introduction

The emerging byte-addressable Persistent Memory (PM) is poised to be the next revolution in computing architecture. In contrast to DRAM, PM has lower energy consumption, higher density and retains its contents even when powered off. Nearly one decade after the first research papers started investigating PM, typically resorting to inaccurate software-based emulations/simulations, the first DIMMs of PM are finally commercially available [23]. This constitutes a notable opportunity to validate, with real PM hardware, the efficiency of the PM-related methods that have been proposed so far.

Along this research avenue, this paper focuses on one problem that has received significant attention in the recent literature: how to implement Persistent Transactional Memory (PTM) in commodity systems equipped with PM and Hardware Transactional Memory (HTM).

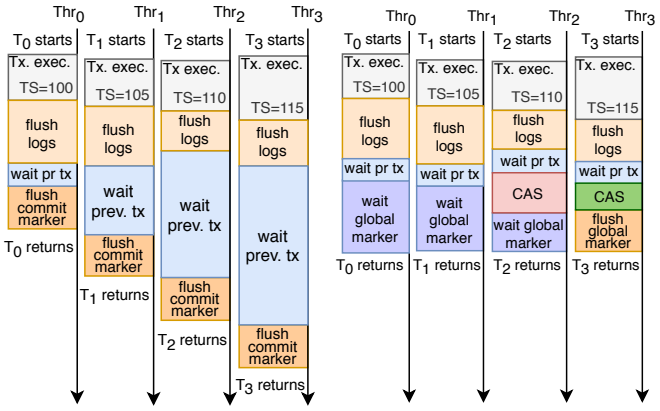
HTM implements in hardware [19, 21, 32] the abstraction of Transactional Memory (TM), an alternative to lock-based synchronization that can significantly simplify the development of concurrent applications [34]. Due to its hardware nature, HTM avoids the overhead imposed by software-based TM implementations. However, the reliance of commodity HTM implementations on CPU caches raises a crucial problem when applications access data stored in PM from within a HTM transaction. Since CPU caches are volatile in today's systems, HTM implementations do not guarantee that the effects of a hardware transaction are atomically transposed to PM when the transaction commits — although such effects are immediately visible to subsequent transactions.

To tackle this issue, recent proposals [4, 14, 15, 28] rely on a set of software-based extensions that, conceptually, are based on Write Ahead Logging (WAL) schemes [31]: first they log modifications and only then they modify the actual data. However, implementing a WAL scheme on commodity HTM raises several challenges. The fact that commercial HTMs deterministically abort transactions that try to persist the cached logs in PM is an impediment to reuse classical DBMS solutions [31]. Instead, logs need to be flushed outside of the transaction boundaries. This essentially decouples transaction *isolation* – as provided by the HTM's concurrency control – from transaction *durability* – as ensured by WAL.

This decoupling introduces a second challenge: PTM implementations need to ensure that the order by which the effects of a transactions become visible is consistent with the order by which it is persisted.

Existing solutions for commodity HTM cope with this challenge by introducing a sequential phase in the critical execution path of the commit logic. To circumvent this limitation, several solutions allow transactions that commit in HTM to externalize their results before their durability is ensured.

Unfortunately, this approach relaxes correctness, since it no longer guarantees *immediate durability* [26]. This is a fundamental limitation for applications that, after committing a transaction, can trigger externally visible actions. An external entity might observe actions that causally depend on a transaction whose writes to PM may not be recovered after a crash (under such relaxed PTMs). To cope with this, applications are extended with intricate compensation logic, which, we argue, is at odds with the original simplicity of transac-



(a) NV-HTM (b) SPHT

Figure 1: (a) The main scalability limitation of NV-HTM [4]: the commit marker is updated in a decentralized but sequential way. (b) How SPHT avoids this limitation (see §3.1).

tional memory. We aim to avoid the pitfalls of relaxed PTM, by providing a PTM for commodity HTM/PM that ensures immediate durability while achieving high scalability.

As a **first contribution**, we experimentally evaluate the cost of ensuring immediate durability with today's state of the art PTMs in a real system equipped with HTM (Intel TSX) and PM (Intel Optane DC PM). We implement 6 PTM systems (disabling their relaxed durability optimizations) and experimentally evaluated them in STAMP [6] and TPC-C [37]. To the best of our knowledge, this represents the first study to evaluate PTMs on a real PM/HTM-equipped system.

As a **second contribution**, we address the question of whether immediate durability can scale on commodity HTM. We devise novel scalable techniques to address the limitations of existing PTMs, which we incorporated in SPHT (Scalable Persistent Hardware Transactions). In a nutshell, SPHT introduces a new commit logic that considerably mitigates the scalability bottlenecks of previous alternatives, providing up to 2.6×/2.2× speedups at 64 threads in, resp., STAMP/TPC-C. Moreover, SPHT introduces a novel approach to log replay that employs cross-transaction log linking and a NUMA-aware parallel background replayer. In large persistent heaps, the proposed approach achieves gains of 2.8×.

The remainder of the paper is organized as follows. §2 provides background on PTM, highlighting the scalability issues of previous solutions. §3 presents SPHT, which we evaluate in §4 against other 5 state of the art PTMs. Finally, §5 concludes the paper.

2 Background on PTM

Various works have investigated how to implement PTM systems. Existing solutions differ by the durability semantics they offer, the nature (hardware and/or software) of the mechanisms they adopt and in some key design dimensions.

Durability semantics. Some PTMs consider "classic" strong guarantees, i.e., if a transaction T returns successfully from its commit call, then T shall be recovered upon a crash and any transaction whose effects T observed shall also be recovered.

We use the term "immediate durability" [26] to refer to the above guarantees, although other papers call it "immediate persistence" [14] or "durable linearizability" [22].

Relaxed durability semantics, which other systems [14, 15, 22, 28] have considered, only ensure that the recovered state is equivalent to one produced by the sequential execution of a *subset* of the committed transactions. As such, these systems can fail to recover transactions that successfully returned from the commit call, e.g., because a crash occurs briefly after that. Intuitively, implementations that rely on relaxed durability semantics have higher throughput for two main reasons: (i) they require a less strict synchronization among concurrent transactions in their commit phase; and, (ii) they allow for removing the costs incurred to ensure durability out of the critical path of execution of the transaction commit logic.

However, when applications do require stricter semantics, programmers are faced with additional complexity: having to develop compensation logic or to manually specify for which sub-transactions immediate durability should be guaranteed [15]. The focus of this work is on immediate durability (despite some tested systems supporting relaxed durability). Next we discuss how systems ensure such semantics.

Software vs hardware implementations. The first PTM proposals relied on software mechanisms [8, 38]. These initial works paved the way for the following generations of software-based PTM implementations [9, 10, 20, 25–27, 29, 30, 41]. Essentially, these proposals extend different software transactional memory (STM) algorithms with logging and recovery mechanisms to ensure durability.

With the introduction of HTM support in mainstream CPUs [17, 33], a second wave of proposals has focused on how to enable the execution of hardware transactions (i.e., transactions executed using HTM) on PM. Due to its hardware nature, HTM avoids the notorious instrumentation costs of STM, which can impose significant overhead especially in applications with short-lived transactions [13]. In existing HTM systems, though, committed transactions are not guaranteed to be atomically persisted, as some of their writes may be lingering in the cache and not have been applied to PM. Further, commodity HTMs do not allow persisting the cached logs to PM within the hardware transaction context. This prevents the use of classical WAL schemes (conceptually at the basis of existing software-based PTMs), which assume that logs are always persisted before application data is.

Some works tackled these issues by proposing *ad hoc* hardware extensions [1, 3, 16, 24, 40]. As such, these solutions cannot be used with existing off-the-shelf systems. More recent works have overcome this shortcoming by proposing software-based approaches that operate on top of unmodified commodity HTM. The most notable examples are DudeTM [28],

cc-HTM [15], NV-HTM [4] and Crafty [14]. All these solutions implement some form of WAL on top of HTM. For improved throughput, the log is typically implemented as a set of per-thread logs in PM[1].

Enforcing the WAL rule. Existing PTMs for commodity HTM rely on two main alternative strategies to enforce the WAL rule, i.e., ensure that the log of a transaction is persisted before any of its changes is applied to persistent data.

A first approach, adopted by DudeTM and NV-HTM, is to have hardware transactions access a *volatile* "shadow" copy of the persistent snapshot.

A second option is non-destructive undo logging, as proposed in Crafty [14]. In this approach, transactions execute directly in PM, with their writes tracked in a persistent undo log and a volatile redo log. To ensure the WAL rule, every write issued in an HTM transaction is undone before commit, which guarantees that the transaction does not alter the PTM's state. Next the undo log is persisted and only then the transaction's writes can be applied to PM.

In both strategies, after committing in HTM, the generated log(s) are flushed to PM in two steps. First, a commit marker is appended to the log. This marker defines the transaction as durable and includes a timestamp that is used to order durable transactions. Next, each logged write is *replayed*, in timestamp order, on the target memory location in PM.

The choice between shadow copy or non-destructive undo logging strongly impacts the available solution space. As we show next, the state of the art systems that implement the above approaches suffer from severe scalability limitations.

Ordering transactions in the logs. One key issue is how to establish the replay order of update transactions in the logs. Existing proposals opt for logical or physical timestamps.

In the first alternative a global logical clock is incremented before each transaction commit and is later appended to the redo log. This type of clocks are likely to become a contention point at high thread counts (§4), hindering scalability by generating frequent spurious aborts. DudeTM employs logical timestamps and, thus, suffers from the above limitation. An additional scalability issue of DudeTM is that its volatile per-thread redo-log has to be processed, copied and flushed by auxiliary thread(s) to a centralized redo-log in PM, incurring relevant synchronization costs.

In contrast, physical timestamps can be acquired at low latency and with no synchronization via, e.g., the x86 RDTSCP instruction. cc-HTM, NV-HTM and Crafty exploit this mechanism. To ensure that the state recovered after a crash is consistent, though, these systems require ensuring an additional property: before a transaction T with timestamp TS can append its commit marker to the log, any other committed transaction T' with timestamp $TS' < TS$ must be already marked as committed in the log. In fact, if this property were violated, upon recovery, T' may not be replayed, whereas T will - this would yield an inconsistent state in case T had observed some write of T' (since T logically depends on T').

Fig 1a illustrates the scheme employed by NV-HTM. An inherently sequential phase in the commit logic ultimately bounds the maximum system throughput to the rate at which commit markers can be persisted in the log. Considering that flushes incur a higher latency in PM than in DRAM [23], this scheme can severely hinder scalability.

Besides the above issue, Crafty adopts a non-destructive undo logging scheme, which incurs additional problems. After flushing the undo log of an HTM transaction T (recall that this is done outside the scope of T, after its commit), Crafty starts a new HTM transaction that atomically: (1) checks if a global clock has changed since T's first execution; and (2) in the negative case, replays T's redo log in PM and updates the global clock. If the global clock is found to have increased (i.e., *any* concurrent transaction did commit), the whole transaction logic of T is re-executed: if T produces the same writes as in its first execution, T is marked as durable; else, T's undo log is discarded and the whole process is restarted.

This approach has two main limitations: (1) the update of the global clock is likely to generate contention at high thread counts, causing frequent transaction re-executions; (2) executing twice a transaction not only introduces overhead, but also increases the likelihood of conflicts by extending the period of time during which transactions execute concurrently.

Log replay. Another key design choice is how to replay the writes in the redo log on PM, while respecting the timestamp order. With the exception of Crafty, log replay occurs only after the transaction(s) being replayed is already durable (as ensured by the persistent redo log). Therefore, the application threads do not need to wait for this phase in order to continue, which can be performed in background. However, there are two relevant exceptions where the progress of the application is affected by the log replay. The first one is upon recovery, where a stable snapshot is rebuilt from the persistent logs. The second one is during transaction processing: once the available log space is exhausted, the application threads have to wait for the log replay to reclaim log space. The efficiency of the log replay process is, thus, of paramount importance.

All the analysed solutions (but Crafty) adopt a non-scalable log replay mechanism: they sequentially replay the logs via a single background thread. Moreover, the efficiency of the replay phase in existing systems decreases as the number of threads processing transactions grows. The larger the thread count, in fact, the larger the number of per-thread logs that need to be examined in the replay phase to determine which transaction (from some per-thread log) should be replayed next, according to the timestamp order.

Summary. Table 1 summarizes the main scalability limitations of state of the art solutions. As we show in the remainder of the paper, SPHT avoids all of them. Regarding the

[1] With the exception of DudeTM, which maintains per-thread logs in volatile memory, whose entries are later flushed to log(s) in PM.

	DudeTM	cc-HTM	NV-HTM	Crafty	SPHT
Global clock updated by txs	Y	N	N	Y	N
Extended tx vulnerability window	N	N	N	Y	N
Sequential mechanism to ensure durability	N	Y	Y	Y	N
Sequential Log Replay	Y	Y	Y	N	N

Table 1: Summary of the factors limiting the scalability of proposed PTM implementations for commodity HTM (assuming their operation with immediate durability semantic).

scalability challenges associated with orderly redo logging, SPHT addresses them by introducing a novel, highly scalable commit protocol (§3.1) that amortizes the cost of ensuring immediate durability across multiple concurrent transactions. SPHT's design avoids spurious aborts due to the access to shared metadata from within hardware transactions [28] or to the need to execute a transaction twice [14]. Concerning log replay, SPHT overcomes the scalability limitations of existing solutions by introducing a mechanism for NUMA-aware parallel log replaying (§3.3) as well as a "log linking" technique (§3.2) that spares replayers from the cost of scanning every thread's log to determine the transaction replay order.

3 SPHT

SPHT assumes a system in which PM is exposed to applications by means of *persistent heaps*. A persistent heap is created by using the operating system (OS) support to memory-map the persistent data, stored in a PM-aware file system, into the application address space [36]. SPHT exposes a classic transaction demarcation API and transparently exploits the underlying HTM support along with a novel software-based scheme to ensure immediate durability.

Fig 2 illustrate SPHT's architecture, which includes two main processes: the **Transaction Executor** (TE), which runs the TM-based parallel application, and the **Log Replayer** (LR). Transactions are executed by multiple worker threads spawned by the TE process. The TE process also *mmaps* a persistent heap into its address space using the OS Copy-on-Write (CoW) option. This option creates a shadow copy of the persistent heap shared by all worker threads, which serves as a working snapshot (WS) that transactions access directly. Updates to the WS are not immediately propagated to the persistent heap. Thus, the updates generated by committed HTM transactions are still volatile.

Like most systems analysed in §2, each worker thread has a private durable redo log that it uses to track the updates performed by each transaction. Once a transaction commits, its updates may still reside in the cache. Thus, the redo log needs to be explicitly forced to persistent memory *after* the HTM commit. At that point, a timestamped commit marker declares the transaction as durable. We discuss how SPHT implements this mechanism in a highly scalable way in §3.1.

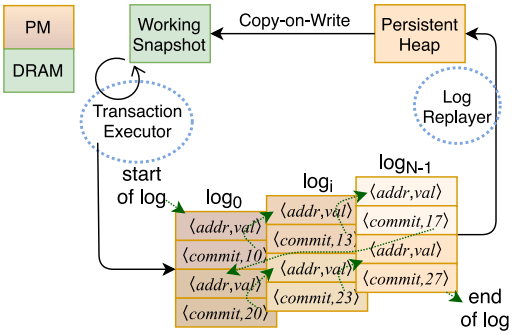


Figure 2: High level view of SPHT architecture.

Since transactions work on a shadow volatile working snapshot, the persistent heap is kept up to date by eventually replaying the redo logs. To handle this, the LR process *mmaps* the persistent heap into its address space, but in a shared (instead of private) state. As such, the LR can directly write to the persistent heap. As mentioned in §2, the LR process takes advantage of two novel ideas to ensure high scalability of log replay. First, it relies on a novel *log linking* mechanism that spares the replayer threads from the cost of having to determine which transaction should be replayed next. We present this mechanism in §3.2. Second, the log replay is parallelized in a NUMA-aware fashion, as detailed in §3.3.

3.1 Transaction processing and durability

The key idea that SPHT exploits to overcome the scalability limitations of state of the art solutions is to mitigate the cost of ensuring immediate durability by amortizing it across multiple transaction commits. SPHT's design is based on the observation that, at high thread count, a large number of transactions is likely to be concurrently trying to commit. SPHT exploits this observation by ensuring the immediate durability of *all* of them via a *single* update of a *persistent global marker*, noted ${}^{p}marker$, which stores the timestamp of the most recent durable transaction. This approach is similar in spirit to the group commit mechanism used in DBMSs [12], which in SPHT we customize to make use of HTM and PM.

Fig 1b illustrates the idea at the basis of the proposed mechanism, by considering the same execution used to illustrate the scalability limitations of NV-HTM (in Fig 1a). Similarly to NV-HTM, SPHT relies on physicals clocks to establish the order by which transactions are replayed. After an HTM commit, SPHT allows the threads to flush their logs out of order (i.e., without any inter-thread synchronization). As discussed in §2, there is a key issue with prior proposals based on physical timestamps (e.g., NV-HTM). The issue is that the log of a transaction T, after being flushed to PM, cannot be marked as durable just yet. In fact, there may exist some other transaction T' with a smaller timestamp and still not marked as durable, whose effects T may have observed.

SPHT copes with this issue as follows: each thread externalizes a timestamp ${}^{v}ts$ in volatile memory that contains the

Algorithm 1 SPHT- Base algorithm

```
   Shared Volatile Variables
1: vts[N], vmarked[N], visUpd[N]
   Persistent Variables
2: pwriteLog[N], pmarker
   Thread Local Volatile Variables
3: vts′, vskipCAS
4: function BEGINTX
5:     visUpd[myTid] ← FALSE
6:     vskipCAS ← FALSE
7:     UNSETPERSBIT(vts[myTid])                    ▷ Logs are not persistent
8:     vts[myTid] ← RDTSCP                         ▷ lower bound of final ts
9:     HTM_BEGIN                                   ▷ begin hw tx
10: function WRITE(addr, val)
11:     logWrite(addr, val)                        ▷ log to PM, no flush
12:     ∗addr← val                                 ▷ execute write
13: function COMMITTX
14:     vts′ ← RDTSCP                              ▷ store physical clock to local var.
15:     HTM_COMMIT                                 ▷ commit hw transaction
16:     vts[myTid] ← ts′                           ▷ Externalize the final timestamp
17:     if isReadOnly then                         ▷ Read-only txs...
18:         SETPERSBIT(vts[myTid])                 ▷ ...unblock the others
19:         return                                 ▷ ...and return immediately
20:     visUpd[myTid] ← TRUE                       ▷ Mark as update tx
21:     logCommit(pwriteLog[myTid], ts′)           ▷ Flush tx log.
22:     SETPERSBIT(vts[myTid])                     ▷ Signal logs are durable
23:     WAITPRECEDINGTXS
24:     UPDATEMARKER
25: function WAITPRECEDINGTXS
26:     for t ∈ [0..N − 1] do
27:                                  ▷ Wait until prec. txs have flushed their logs
28:         while vts[t] < vts[myTid] ∧ ¬ISPERSBIT(vts[t]) wait
29:                                  ▷ If any update tx with large ts exists...
30:         if vts[t] > vts[myTid]∧ visUpd[t] then
31:             vskipCAS ← TRUE                    ▷ this tx can skip the CAS
32: function UPDATEMARKER
33:                          ▷ Is it needed to and am I responsible for updating pmarker?
34:     if pmarker < vts[myTid] ∧ ¬vskipCAS then
35:         val← pmarker
36:         while val < vts[myTid] do
37:             val ← CAS(pmarker, val, vts[myTid])
38:         if (CAS was successful) then
39:             flush(pmarker)
40:             vmarked[myTid] ← vts[myTid]        ▷ Signals pmarker is flushed.
41:             return
42:     while TRUE do                              ▷ Wait till flush of pmarker
43:         for t ∈ [0..N − 1] do                  ▷ ...is complete
44:             if vmarked[t] ≥ vts[myTid] then return
```

following information: (i) the timestamp of the last transaction; and, (ii) whether the log of the last transaction is persistent (*isPers* bit). After flushing its logs, T advertises to all other threads its completion by setting the *isPers* bit in its ^{v}ts. T then enters a wait phase during which T scans the timestamps of the other worker threads with a two-fold purpose: (i) ensuring that any transaction with a smaller timestamp has finalized persisting its own logs; (ii) determining which is the transaction with the largest timestamp, among the ones currently in the commit phase.

The former condition guarantees that T can be safely marked as durable, by updating (and flushing) the global marker ($^{p}marker$). The latter condition enhances efficiency by exploiting, opportunistically, the presence of other concurrent transactions to reduce the number of updates (and flushes) of $^{p}marker$. Specifically, if T detects a transaction T' with a larger timestamp, T avoids updating $^{p}marker$, as T' will do so. When T' updates $^{p}marker$, the durability of T is also implicitly ensured, since T' will store in $^{p}marker$ its own timestamp, which is larger than the one of T and, as such, ensures also the durability of T.

This mechanism is not exempt from critical races. In fact, two transactions may assume to have the largest timestamp and attempt to update concurrently $^{p}marker$. We tackle this issue by manipulating $^{p}marker$ via a Compare-and-Swap (CAS) instruction. Fig 1b illustrates an example execution. Transactions T_0 and T_1 detect the presence of T_2 and/or T_3 and delegate to them the update of the global marker. T_2 and T_3 compete via a CAS to update $^{p}marker$. Assuming that T_3 succeeds, T_3 flushes $^{p}marker$, thus ensuring the durability of the 4 transactions. As shown in Fig 1, not only SPHT reduces the number of synchronous updates of the commit marker with respect to solutions like NV-HTM (reducing the pressure on the bandwidth-constrained PM [23]), but it also allows multiple transactions to be marked as durable in parallel.

Note that, if no concurrent transaction is detected after flushing the logs, the proposed solution has a cost similar to NV-HTM, as both require synchronously updating a commit marker. In the case of SPHT, though, a single global marker is updated, whereas in NV-HTM, each thread appends a commit marker to its own log. Because of this, SPHT uses a more expensive operation (i.e., a CAS) to update the global marker. As we will show in §4, though, this cost is largely outweighed by the scalability benefits that the SPHT's scheme enables. It is also worth pointing out that this design tends to minimize the chance that multiple transactions contend to CAS the global marker. In fact, $^{p}marker$ is only updated by a transaction that detects to have the largest timestamp. Thus, most of the CAS operations are uncontended and, therefore, introduce relatively low overhead in modern processors [35].

Pseudo-code. The above scheme is formalized in Alg 1. For simplicity, memory and persist barriers are omitted in the pseudo-code and are discussed below. First, all loads/stores to shared variables abide by C/C++ acquire/release semantics. Second, we use synchronous flushes (CLWB followed by SFENCE) in *logCommit* (l.21) and *flush* (l.39).

DATA STRUCTURES. We mark volatile data structures with a superscript v (^{v}) and persistent variables with a superscript p (^{p}) for clarity. SPHT maintains two persistent data structures: (i) per-thread redo-logs ($^{p}writeLog[N]$); and, (ii) a global marker ($^{p}marker$), which stores the timestamp (physical clock) of the most recently durably commmitted transaction, i.e., guaranteed to be replayed in case of a crash.

Each thread t (of the N available in the system) also uses the following global *volatile* data structures: (i) the timestamp of the last (or current) transaction T executed by t ($^{v}ts[N]$); (ii) the *isPers* flag, implemented by reserving a bit in $^{v}ts[t]$, which serves to notify whether t has (synchronously) flushed the logs of T to PM; (iii) the last timestamp t wrote to and flushed in $^{p}marker$ ($^{v}marked[N]$); and, (iv) a flag that advertises whether T is an update transaction ($^{v}isUpd[N]$).

The logs are per-thread circular buffers containing an ordered sequence of transactions. Each logged transaction is a sequence of (i) $\langle addr, val \rangle$ pairs (i.e., the transaction's write set) followed by (ii) a timestamp that serves also as an end delimiter. The timestamp is distinguishable from an address by setting its first bit to 1. For simplicity, we omit the metadata used to track the log's start and end.

BEGIN TRANSACTION. Before a thread t starts a hardware transaction T (via HTM_BEGIN, line 9), t stores the current value of the physical clock (obtained via RDTSC) in its ${}^{v}ts$ variable and sets its *isPers* bit to 0. It also sets its ${}^{v}isUpd$ variable to false, which informs other threads that T is not guaranteed to be an update transaction, yet.

It should be noted that, at this stage, the timestamp advertised in ${}^{v}ts$ represents a lower bound estimate on the final timestamp (i.e., the one establishing the durability order) that T will obtain right before committing (via HTM_COMMIT, l.15). This mechanism ensures the visibility of T throughout its execution to other concurrent threads.

WRITE INSTRUMENTATION. SPHT logs the writes (Alg 1, l.10) in PM via the logWrite primitive. Logging a write consists in appending a pair $\langle addr, val \rangle$ at the tail of the log.

COMMIT PHASE. Before committing the hardware transaction via HTM_COMMIT (Alg 1, l.15), the final timestamp is obtained by reading the physical clock and storing it in a local variable (${}^{v}ts'$). This timestamp is only advertised in the shared variable ${}^{v}ts$ after HTM commits. The latter, in fact, is accessed non-transactionally by concurrent threads (in the WAITPRECEDINGTXS function) and updating it from within the hardware transaction would induce (spurious) aborts.

Read-only transactions, which produce no log, can return immediately. Before, though, they set *isPers* to 1, which, as we will see, unblocks concurrent threads that may be waiting.

Update transactions, instead, append their timestamp to the log and flush it (via the logCommit primitive). Next, they advertise that they are update transactions and that they are durable by setting their ${}^{v}isUpd$ and *isPers* flags, respectively.

Next, in WAITPRECEDINGTXS, T examines the timestamps of every concurrent transaction and waits until the ones with a smaller timestamp have finished flushing their logs (l.28). At this point it is safe to update the global marker with the timestamp of T. However, to enhance efficiency, in l.30, T determines whether there is an update transaction with a larger timestamp, say T'. In this case, when T' updates the global marker with its own timestamp, it also ensures the durability of T. Hence, T omits the updating of the global marker, sets the ${}^{v}skipCAS$ flag and just waits until a timestamp larger than its own has been persisted in the global marker.

Finally, the transaction executes UPDATEMARKER. Here, it verifies if the global marker does not yet ensure its own durability (${}^{p}marker <^{v} ts[myTid]$) and if it cannot count on other transactions with larger timestamp to update ${}^{p}marker$ (${}^{v}skipCAS$ is false): in such a case, the transaction attempts to CAS ${}^{p}marker$ (l.37) to the value of its ${}^{v}ts$, until a timestamp larger than or equal to its own is present.

If T successfully executes its CAS, T ensures that the write it performed is persisted by flushing ${}^{p}marker$ (l.39). T advertises that ${}^{p}marker$ is flushed by writing its ${}^{v}ts$ in its ${}^{v}marked$ variable. After that, T returns.

If T fails the CAS, T needs to wait until it observes a value in the ${}^{v}marked$ array that is larger than its timestamp: this guarantees that some thread must have CASed and flushed a value in ${}^{p}marker$ that also ensures T's durability.

SINGLE GLOBAL LOCK (SGL). HTM is a best effort synchronization mechanism that, to ensure progress, normally relies on pessimistic fall-back path (e.g., activated if the transaction fails repeatedly to commit in hardware) based on a Single Global Lock (SGL). When this mode is activated, any concurrent hardware transaction is immediately aborted. However, in SPHT, if a thread activates the SGL path, there may still be transactions that have already completed executing in HTM, but are still in their commit phase (e.g., flushing their logs). To guarantee correct synchronization with these transactions, the SGL path ensures the durability of its updates by using the same logic of HTM transactions.

Correctness arguments. We prove the correctness of SPHT by showing that it satisfies two properties (which were already used to define NV-HTM's correctness criteria [4]): (C_1) the timestamps obtained during transaction execution reflect the HTM commit order[2]; (C_2) if a transaction T returns from a commit call to the application, the effects of T and of every committed transaction that precedes T in the HTM serialization order, are guaranteed to be durable.

SPHT and NV-HTM share the same timestamping scheme, which was already proved to ensure property C_1 [4]. Thus, in the following, we focus on proving that SPHT ensures C_2.

If a transaction T, executing at thread t, returns successfully from its commit call to the application then: (i) the log of t necessarily includes T, including its final commit marker (Alg 1 l.21); (ii) ${}^{p}marker$ persists a value larger or equal than the timestamp of T (${}^{v}ts[t_T]$), since either T set ${}^{p}marker$ to ${}^{v}ts[t_T]$ (l.38-40) or some other concurrent transaction T' s.t. ${}^{v}ts[t'_{T'}] > {}^{v}ts[t_T]$ updated and flushed ${}^{p}marker$ (l.42-44).

These conditions ensure that upon recovery T will be replayed. It is only left to prove that, if T returns from its commit call, any committed transaction T', s.t. ${}^{v}ts[t'_{T'}] < {}^{v}ts[t_T]$, will also be replayed. This is guaranteed since, before returning from its commit call, T ensures that any thread that may be executing a transaction T' with a smaller timestamp has set *isPers* (l.22). Hence, the log of the thread that executed T' necessarily includes T', with its final commit marker, which, together with the condition ${}^{p}marker \geq {}^{v}ts[t_T] > {}^{v}ts[t'_{T'}]$, ensures that T' will also be replayed.

[2] More formally, if a transaction T with a timestamp ts conflicts with T' with ts', and $ts < ts'$, then T is serialized by HTM before T'.

3.2 Linking transactions in the log

The algorithm presented in the previous section (similarly to other solutions [4, 14, 15, 28]) requires replayers to scan the whole set of per-thread logs to determine the transaction that should be replayed next. This can have a significant impact on the log replay performance (up to 3.5× slowdown, §4), especially in systems where a large number of threads can process transactions (since each thread maintains its own log).

SPHT tackles this issue by extending the transactions' log with an additional entry that is used to store a pointer to the beginning of the next transaction in the replay order. Transactions update this pointer during their commit stage.

Let us denote with T_i the i-th transaction in replay order and assume that transactions are replayed from the oldest to the most recent one. In a nutshell, once transaction T_i has committed in hardware and established its final (physical clock based) timestamp, it needs to determine the identity of transaction $T_{i\text{-}1}$, and update the link slot in the log of $T_{i\text{-}1}$ with a pointer to (the start of) T_i's log.

Unfortunately, extending the algorithm presented in §3.1 to allow T_i to determine the identity of $T_{i\text{-}1}$ is not trivial. The key problem is that, when transaction T_i reaches its commit phase, the thread that committed $T_{i\text{-}1}$, denoted $t_{T_{i\text{-}1}}$, may have already started a new transaction and overwritten its timestamp ${}^{v}ts[t_{T_{i\text{-}1}}]$. This makes it impossible for T_i to determine the identity of $T_{i\text{-}1}$ by inspecting the ${}^{v}ts$ array.

To address this issue, SPHT tracks also the metadata of the previous transaction processed by each thread. This is sufficient since we ensure that if T_i has not determined the identity of $T_{i\text{-}1}$ yet, then $t_{T_{i\text{-}1}}$ will be able to start at most one new transaction. To ensure this property, T_i scans the metadata of the other threads and establishes its predecessor *before* setting its *isPers*. Recall that this scheme allows T_i to prevent transactions with larger timestamps from completing their commit phase. Thus, it prevents $t_{T_{i\text{-}1}}$ from committing any transaction that $t_{T_{i\text{-}1}}$ started after committing $T_{i\text{-}1}$.

During this scanning phase, T_i discriminates between (concurrent) transactions with smaller timestamps that have their *isPers* set to 1 or 0. Transactions with *isPers* set to 1 already established their final ${}^{v}ts$, so their timestamp can immediately be analyzed to determine if any of them may be T_i's predecessor. Further, T_i does not need to wait for these transactions before moving on with UPDATEMARKER.

Transactions with smaller timestamps that have their *isPers* set to 0, though, prevent T_i from executing UPDATEMARKER. T_i tracks these transactions in *precTXs*, a set that will be consulted during T_i's wait phase. Before starting to wait, T_i sets *isPers* to 1 to unblock transactions with larger timestamps.

Next, the algorithm proceeds similarly to the base version. Namely, T_i waits for all transactions in *precTXs* (i.e., that may precede T_i) and then executes the update marker logic. The key difference is that, in the wait phase, once T_i can determine the final timestamp for a transaction T_j, it also verifies whether T_j might be its preceding transaction. This is achieved by checking whether T_j has the largest timestamp among the transactions that precede T_i (i.e., $T_j = T_{i\text{-}1}$). Finally, before returning from the commit call, T_i updates the link slot of $T_{i\text{-}1}$ to point to the start of T_i's log.

Pseudo-code. The pseudo-code formalizing the proposed mechanism is reported in Alg 2. The lines of code that are unchanged with respect to Alg 1 are coloured in brown. For space constraints we have to omit the correctness proof for Alg 2, which can be found in our technical report [5].

Algorithm 2 SPHT- Forward linking.

```
   Additional Shared Volatile Variables:
1: vlogPos[N], vprevLogPos[N], vprevTs[N]

   Additional Thread Local Volatile Variables
2: vpTs, vpLogPos, vpThread, vprecTXs
3: function BEGINTX
4:     visUpd[myTid] ← FALSE ; vskipCAS ← FALSE ;
5:     atomic do                                   ▷ vectorial instr stores multiple fields
6:         vlogPos[myTid] ← myLinkSlot             ▷ flags link pos for the next tx
7:         UNSETPERSBIT(vts[myTid])
8:         vts[myTid] ← RDTSCP
9:     HTM_BEGIN
10: function COMMITTX
11:     ts′ ← RDTSCP ; HTM_COMMIT
12:     ∗ vlogPos[myTid] ← ts′                      ▷ flags stable ts in own log
13:     vts[myTid] ← ts′
14:     if isReadOnly then
15:         SETPERSBIT(vts[myTid])
16:         return
17:     visUpd[myTid] ← TRUE ;
18:     logCommit(pwriteLog[myTid], ts′)
19:     SCANOTHERS                                  ▷ estimate prev & unstable txs
20:     SETPERSBIT(vts[myTid])                      ▷ next tx can write in link
21:     WAITUNSTABLETXS                             ▷ discover prev tx
22:     UPDATEMARKER
23:     ∗ vpLogPos ← myLinkSlot                     ▷ link prev tx to my log
24:     atomic do                                   ▷ keep track of this tx
25:         vprevLogPos[myTid] ← vlogPos[myTid]
26:         vprevTs[myTid] ← vts[myTid]
27: function SCANOTHERS                             ▷ estimate preceding TXs
28:     vpThread ← myTid                            ▷ init search with own prev. tx
29:     vpTs ← vprevTs[myTid] ; vpLogPos ← vprevLogPos[myTid] ;
30:     for t ∈ [0..N − 1] do
31:         atomic do                               ▷ for each t take a snapshot using a...
32:             tmpLogPos ← vlogPos[t]              ▷ ... vectorial load
33:             tmpPrevLogPos ← vprevLogPos[t]
34:             tmpTs ← vts[t]
35:             tmpPrevTs ← vprevTs[t]
36:         if tmpTs < vts[myTid] then              ▷ search preceding txs
37:             if ISPERSBIT(tmpTs) then            ▷ search stable txs
38:                 if tmpTs > vpTs then
39:                     vpTs ← tmpTs ; vpThread ← t ;
40:                     vpLogPos ← tmpLogPos
41:                     continue
42:             else                                ▷ prec. tx in t that is still running
43:                 append(vprecTXs, ⟨t,tmpLogPos⟩)
44:         if tmpPrevTs < vts[myTid] ∧ tmpPrevTs > vpTs then
45:             vpThread ← t                        ▷ search preceding txs
46:             vpTs ← tmpPrevTs
47:             vpLogPos ← tmpPrevLogPos
48: function WAITPRECEDINGTXS
49:     for ⟨t, vlogPos[t]⟩ ∈ precTXs do
50:         while vts[t] < vts[myTid] ∧ ¬ISPERSBIT(vts[t]) wait
51:         if ISTS(∗vlogPos[t]) ∧ ∗vlogPos[t] < vts[myTid] ∧ ∗vlogPos[t] > vpTs then
52:             ⟨vpTs, vpThread⟩ ← ⟨∗ vlogPos[t],t⟩
53:             vpLogPos ← vlogPos[t]
54:         if vts[t] > vts[myTid] ∧ visUpd[t] then
55:             vskipCAS ← TRUE
```

ATOMIC ACCESS TO METADATA. In the scanning phase (SCANOTHERS , l.27), T_i needs to obtain a consistent snapshot of the metadata of every other thread. This was not an issue in Alg 1, since the per-thread metadata that T_i had to observe was just the timestamp and *isPers*, which are stored within then same single memory word. Alg 2, though, requires T_i to atomically observe a larger set of metadata, i.e., the timestamp (included *isPers*) and the position in the log of the last two transactions processed by each thread, which amounts to 32 bytes. To cope with this issue, we store these metadata contiguously in (volatile) memory and read/write them using vectorial instructions (i.e., x86 AVX), which in recent CPUs guarantee atomic multi-word manipulations [39].

TRACKING THE PRECEDING TRANSACTION. As discussed above, in SCANOTHERS, by setting its *isPers* to 0 (l.7), T_i prevents thread $t_{T_{i-1}}$ from completing the commit of its next transaction. As soon as T_i sets its *isPers* to 1, though, $t_{T_{i-1}}$ can commit a possibly unbounded number of transactions and, by the time T_i accesses $t_{T_{i-1}}$'s metadata, in WAITPRECEDINGTXS, the information regarding T_{i-1}'s timestamp and log pointer may have been already overwritten. We address this issue as follows: (i) once a thread establishes the final timestamp for a transaction T_i, it stores T's timestamp also in the link slot of T_i's log, so that this information remains accessible to the thread that executes T_{i+1} even after *isPers* of T_{i+1} is set (which, as mentioned, allows t_{T_i} to commit an arbitrary number of transactions); (ii) in SCANOTHERS, when T_i detects a transaction T_j with a smaller timestamp and *isPers* set to 0, T_i stores in *precTXs* both the identifier of the thread t_{T_j} and the position of T_j in t_{T_j}'s log; (iii) in WAITPRECEDINGTXS, T_i can then access the timestamp of T_j in t_{T_j}'s log via the pointer stored in *precTXs*.

Note that, when a transaction T_i executes WAITPRECEDINGTXS, it can include in *precTXs* transactions that have a smaller timestamp but are not T_i's immediate predecessor. Denote such a transaction as T_j. By the time T_i inspects the link slot in their log via the ${}^{v}logPos$ pointer (l.51), the link slot may have been already updated by T_j's immediate successor (i.e., T_{j+1}). In this case, T_i must safely detect that T_j cannot be its own predecessor. This is achieved by exploiting the fact that whenever a transaction timestamp is stored in the link slot (l.12), its first bit (which, recall, we use to encode *isPers*) is always set to 0. So, in order to tell whether the link slot is storing a timestamp or a pointer (ISTS() primitive, l.51), we always set to 1 the first bit of any pointer that we store in the log (and reset it 0 during the reply). This is safe since in typical architectures the first bits of an address in user-space is always guaranteed to be zero, but alternative approaches should be used if SPHT were to be used within the kernel.

Backward linking. The proposed technique can be adapted straightforwardly to link transactions in "backward" order, i.e., from the most recent to the oldest one. This enables techniques for filtering duplicate writes [4] by replaying the logs backwards and applying only the the most recent write to each memory position. Backward linking can be achieved by adapting the above logic so to have T_i store into its own log a reference to the start of T_{i-1}'s log.

3.3 NUMA-Aware Parallel Log Replay

The LR makes use of a snapshot in PM and the per-thread logs to produce a fresher persistent snapshot. The last transaction to be considered for replay, say T_i, is the one, among the transactions in the log, to have the largest timestamp that is also smaller than or equal to the persistent global marker (${}^{p}marker$). Any transaction more recent than the ${}^{p}marker$ is guaranteed to not have returned from the commit call. Thus, it can be safely discarded. Using a single threaded replayer, it suffices to apply the modifications by following the links stored in the logs (see §3.2). Before pruning the log, to ensure the durability of the replayed writes, SPHT calls the x86 WBINVD instruction, which efficiently drains the caches.

The key problem to enable parallel replay in the LR is how to ensure that the order by which writes are replayed by multiple concurrent replayers respects the sequential order established in the logs. SPHT circumvents the usage of additional synchronization among different replayer threads by ensuring that their writes target disjoint memory regions. This sharding makes the replay completely parallel and spares threads from enforcing a specific write order.

Fig 3 illustrates the concept, by showing two replayer threads that navigate through the (decentralized) log following the linking information. For illustration purposes, we consider a simplistic sharding policy, which assigns responsibility of even/odd addresses to replayer threads 0 and 1, respectively.

In reality, SPHT uses a more sophisticated policy, which aims at pursuing four goals: (i) minimize overhead for the replayer; (ii) balance load among different threads; (iii) promote cache locality; and, (iv) take advantage of NUMA systems.

Specifically, SPHT shards the transactional heap in contiguous chunks of configurable size, which are strided across a fixed number of parallel replayers. This allows for mapping a given memory address to the corresponding replayer thread via an efficient hash function that simply inspects the most significant part of the address.

Arguably, using small chunks can benefit load balancing: by interleaving in a fine-grained way the regions that each replayer thread is responsible for, it is less likely that a single frequently accessed memory region is assigned exclusively to a single thread (which may generate load imbalances and hamper the global efficiency of the parallel replay process). We observed that chunks with a granularity close to the cache line size generate excessive cache traffic, leading to poor replay performance. This led us to opt for a granularity of 4KB (typical size of pages mapped in DRAM).

As mentioned, one of the design goals of the SPHT's replay logic is to take advantage of the asymmetry of modern

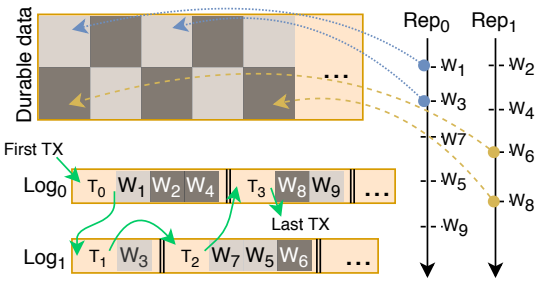


Figure 3: Parallel log replay. The notation W_i denotes a write to address i. Rep_0 and Rep_1 are responsible for odd/even addresses respectively.

NUMA systems, where accesses to local memory regions experience lower latency and higher throughput than accesses to memory regions hosted by a remote node. We exploit this feature by scattering (i.e., pinning) in round robin the set of replayers across the available NUMA nodes and making each replayer responsible of applying only the writes that target an address in their local NUMA node. To this end, we developed a simple NUMA-aware memory allocator that organizes the transactional heap into N different arenas, one for each of the N available NUMA nodes. During transaction processing, the allocation of memory regions across NUMA nodes uses a simple round robin policy to balance memory usage (but clearly alternative policies could be used [11, 18, 42]). This custom memory allocator ensures that the arenas associated with each NUMA node are placed at known address ranges. This allows the replayer threads to detect in a precise and efficient way whether any memory address in the log is mapped to their local NUMA node.

Note that, although this approach requires all replayers to scan the whole log, it is effective for two reasons: (i) since replayers execute at roughly the same speed and issue repeated read requests for the same log regions close in time, these reads are likely to be served from the CPU caches (as we will experimentally confirm in Section 4.2); (ii) PM's performance is asymmetric (read bandwidth is ~3× larger than write bandwidth [23]) hence the main bottleneck of the replay process is the apply phase, rather than log scanning.

Finally, this sharding scheme can be used in conjunction with duplicate filtering schemes [4], e.g., which scan the logs from the most recent to the oldest entry to avoid replaying duplicate writes to the same memory position. In this case, the key issue to address is how to ensure that the tracking of duplicate writes remains correct despite the existence of multiple concurrent replayers. In order to avoid costly synchronization among replayers, SPHT avoids using shared data structures to filter duplicates (e.g., thread-safe set implementations). Conversely, each replayer thread r maintains a volatile bitmap that only tracks writes to memory regions that r is responsible for (each bit of the bitmap tracking writes to a 8 bytes in PM, i.e., the granularity of each write in the log).

4 Experimental Evaluation

Our experiments seek answers to the following main questions: (i) how severe are the scalability limitations of state of the art solutions mentioned in §2 when evaluated on a real PM system (§4.1 and §4.2)? (ii) what are the performance benefits of SPHT's commit logic (§4.1)? (iii) what are the gains of the linking technique during log replay (§4.2) and what are the costs it introduces during transaction processing (§4.1)? (iv) how scalable is the parallel replay technique (§4.2)?

Experimental settings. We conducted all experiments in a dual-socket Intel Xeon Gold 5218 CPU (16 Cores / 32 Threads) equipped with 128GB of DRAM and 512GB of Intel Optane DC PM (4× 128GB). The PM is configured in "App mode" [23] using 2 namespaces and interleaved access. The presented results are the average of 10 runs.

We consider 8 different PTMs, whose implementation we make publicly available[3]: SPHT-NL (no linking), SPHT-FL (forward linking), SPHT-BL (backward linking), NV-HTM [4], DudeTM [28], Crafty [14], cc-HTM [15] and PSTM [38]. PSTM is a software TM that extends TinySTM with durable transactions using Mnemosyne's [38] algorithm. For fairness, we implemented all systems in §2 in a common framework and all of them provide immediate durability. Checkpointing is disabled during transaction processing for all solutions that accumulate logs[4]. The HTM solutions fall back to SGL after 10 retries.

4.1 Transaction processing

We evaluate the performance of SPHT using the STAMP [6] benchmark suite and TPC-C [37]. STAMP was already used to evaluate several prior related solutions [4, 14, 15], since it encompasses transactional applications that, although not originally proposed for PM, would transparently benefit from PM to attain crash-tolerance and/or have access to larger heaps. TPC-C is widely used to benchmark database systems.

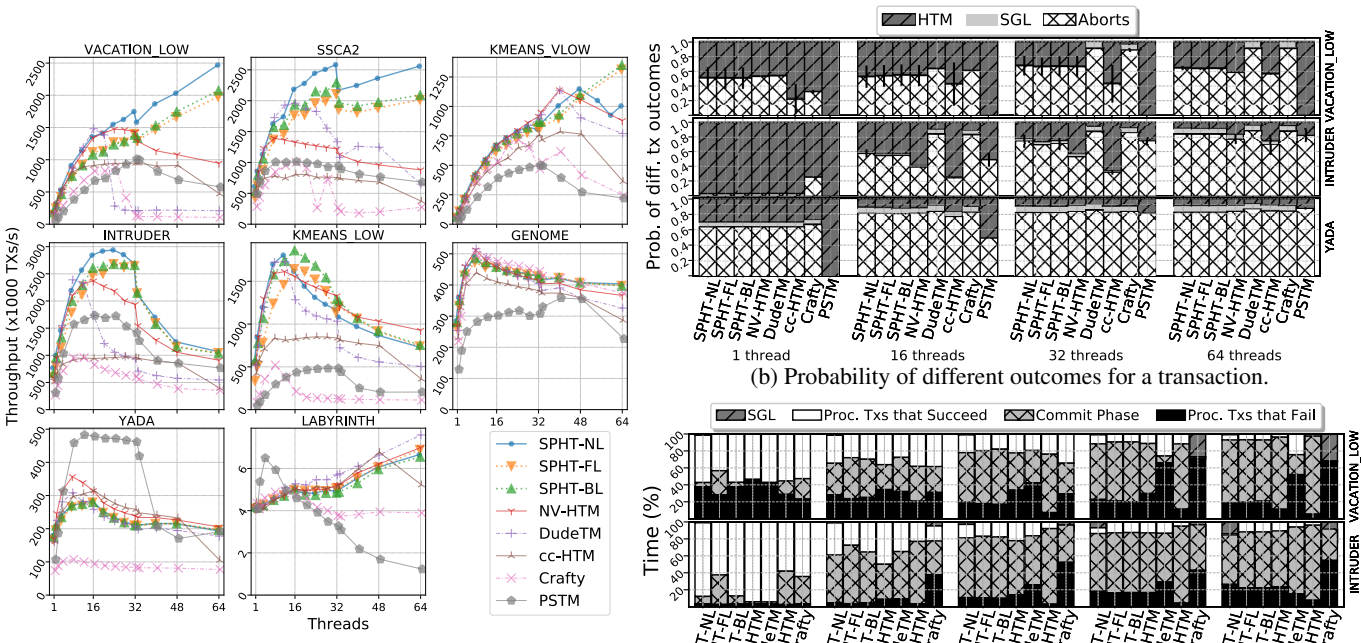
4.1.1 STAMP

STAMP includes 8 benchmarks, but we do not consider Bayes, as it is known to generate unstable performance results [7]. We consider the standard low contention workloads for Vacation and Kmeans. We also configured Kmeans to generate an additional workload with lower contention (KMEANS_VLOW), thus enabling the PTMs to achieve higher scalability levels.

Fig 4a reports throughput as a function of the number of worker threads. The top row contains low contention workloads (VACATION_LOW, SSCA2 and KMEANS_VLOW). The second row contains contention-prone workloads (INTRUDER, KMEANS_LOW and GENOME). And the bottom row have

[3] bitbucket.org/daniel_castro1993/spht

[4] cc-HTM has to activate checkpointing upon completing each transaction in order to comply with immediate durability (see transaction barrier in [15]).



(a) Throughput.

(b) Probability of different outcomes for a transaction.

(c) Transaction time breakdown.

Figure 4: STAMP [6] using standard (++) parameters: (a) throughput; (b) probability for a transaction to commit or abort in HTM, or enter the SGL; (c) breakdown of time spent: in SGL, commits in HTM, aborts in HTM, and in the commit phase.

workloads that are notoriously unsuited for HTM (YADA and LABYRINTH), since their long transactions have a memory footprint that often exceeds the CPU cache capacity.

Low contention benchmarks. SPHT achieves the largest gains with respect to the considered baselines in VACATION_LOW, where it scales up to 64 threads (with a small drop at 33 threads, when we start activating threads on the second socket [2]). At the maximum thread count, SPHT is 2.6× faster than NV-HTM (the second best solution). This can be explained by analyzing the data in Fig 4c, which reports a breakdown of the percentage of time spent by each solution in different activities: at 64 threads, NV-HTM spends significantly more time in the commit phase than SPHT, 76% vs 62%. As a consequence, the ratio between the time spent processing transactions and the time spent committing is ~2× higher for SPHT (~60% vs ~30%). Analogous considerations apply to cc-HTM, which spends almost 95% of time in the commit phase starting at 32 threads, when the single background applier thread becomes the system's bottleneck.

Analyzing the data in Fig 4b, which reports the probability for a transaction to abort, commit in HTM or by using the SGL, we notice that the HTM-based solutions suffer from a non-negligible abort probability even when using a single thread. We verified that this is the case also for non-durable HTM. The reason is that the memory footprint of some transactions exceed the HTM capacity. As the thread count grows, though, Crafty and DudeTM experience a much higher abort rate than SPHT. In Crafty's case, rolling back the transaction and replaying it afterwards (and using a conservative mechanism to detect conflicts in between these phases [14]) leads to higher conflict rates than with the SPHT's variants. In DudeTM's case, the global serialization clock imposes spurious conflicts, which are amplified at high thread counts.

SPHT-FL and SPHT-BL remain the most competitive solutions at high thread count, although they impose an overhead of up to around 25% in VACATION_LOW as well as in SSCA2 w.r.t. the no linking version. It should be noted, however, that the overhead incurred by the linking mechanism is at most 5% in all the other benchmarks. In KMEAN_VLOW, though, the linking variants actually outperform SPHT-NL. The explanation for this behaviour is that the additional operations performed by the linking variants in the commit phase serve as a back-off mechanism, reducing the overall contention. Although not shown in Fig 4b for space constraints, the abort rate with KMEANS_VLOW at 64 threads is 79%, 45% and 43% for SPHT-NL, SPHT-FL and SPHT-BL, respectively.

Finally, at high thread count, the gains of the SPHT variants w.r.t. existing solutions tend to reduce in KMEANS_VLOW, as this benchmarks generates higher contention than VACATION_LOW and SSCA2. Still, at 64 thread the SPHT variants achieve ~30% higher throughput than the best baseline (NV-HTM) and ~5× speed-ups w.r.t. the remaining ones.

Contention-prone benchmarks. These benchmarks scalability is inherently limited by their contention prone nature: above a given number of threads the likelihood of conflicts between transactions grows close to 1 and throughput is severely

hampered in all solutions. Yet, it is worth noting that, in INTRUDER and KMEANS_LOW, all the SPHT variants do scale to a large number of threads and achieve significant speed-ups w.r.t. all other solutions: e.g., SPHT achieves a peak throughput that is ~30% higher than the most competitive baseline, i.e., NV-HTM, scaling up to 24 threads.

HTM-unfriendly workloads. Finally, in LABYRINTH and YADA, as expected, PSTM outperforms all the HTM-based solutions, including SPHT. That is not surprising given that these benchmarks generate large and contention prone transactions, which do not lend themselves to be effectively parallelized using HTM. It is also unsurprising that most of the HTM-based solutions achieve similar performance in these HTM-unfriendly workloads, where a significant fraction of the transactions has to be committed using the SGL (in which case all the tested solutions tend to follow a very similar behavior). The only exception being Crafty, which incurs a much larger overhead than the other HTM-based solutions, due to the large abort costs that it incurs in these workloads.

4.1.2 TPC-C

We implemented three transactions of the TPC-C benchmark, namely Payment, New-Order and Delivery, and report the results in Fig 5. All solutions suffer a throughput drop when they enter hyper-threading after 16 threads, which we do not observe in STAMP. After that drop, SPHT and its linking variants are the only solutions capable of scaling up to 48 threads. As in KMEANS, the backoff introduced by the linking mechanism allows SPHT-FL and SPHT-BL to reduce abort rates (bottom plot of Fig 5). NV-HTM stops scaling above 8 threads, although achieving abort rates that are comparable to or lower than SPHT's. This suggests that NV-HTM is being bottle-necked by its sequential commit mechanism. DudeTM exhibits the same issues as in VACATION_LOW: after 12 threads the global clock creates spurious aborts that hinder throughput (as shown, e.g., at 16 threads). cc-HTM's background thread limits its scalability beyond 8 threads. Crafty's non-destructive undo logging scheme also imposes higher abort rates than NV-HTM and SPHT.

4.2 Log replay

We evaluate the two main novel techniques at the basis of the proposed log replay scheme: (i) linking transactions in the log and (ii) using multiple parallel replayers. For space constraints, we cannot explicitly evaluate the gains deriving from our NUMA-aware design, which, however, we use in all the experiments discussed next. Overall, in the tested system, our NUMA-aware design doubles the bandwidth available to the replayer threads, which is key to increase scalability.

The efficiency of these mechanisms is affected by a number of variables including: (i) the heap size; (ii) the average number of writes per transaction; (iii) the use of filtering technique

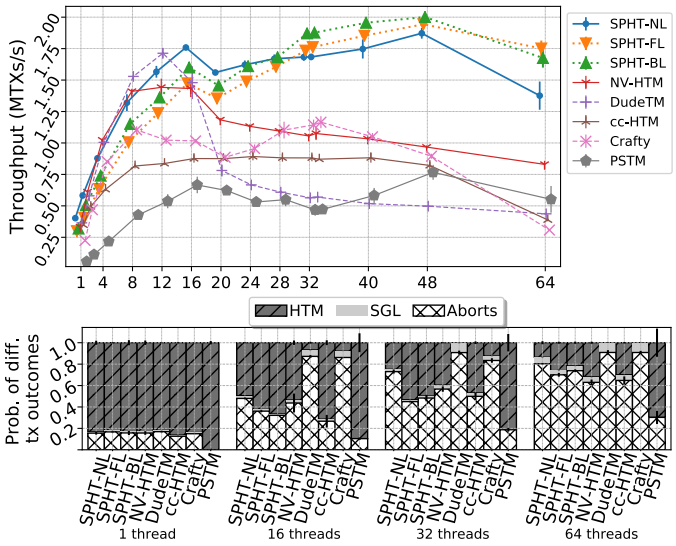


Figure 5: TPC-C using 32 warehouses, 95% Payment, 2% New Order, 3% Delivery transactions.

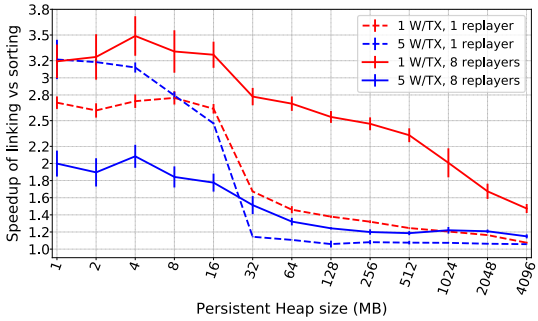


Figure 6: Performance benefits of linking.

and the level of duplicates in the log.

We explore those parameters with a synthetic benchmark in which transactions access the persistent heap uniformly at random, generating a configurable number of writes in each transaction. Once the benchmark completes, the LR fully replays the produced logs and we evaluate its throughput in terms of number of logged writes replayed per second. In the following, we set the number of worker threads to 64, each producing one log (i.e., total of 64 logs to replay).

Linking. Fig 6 shows the relative gain in log processing throughput stemming from linking with respect to a classical solution [4, 28], called *sorting*, where the replay order is established by analyzing all the per-thread logs. In this experiment, the logs contain a total of 10M transactions. We vary on the x-axis the heap size and consider 4 scenarios in which: (i) transactions issue either 1 or 5 writes; (ii) replayer uses either 1 or 8 threads. Linking provides the largest benefits for small heaps (up to 3.5× speed-ups below 4MB). For large heaps, the gains of linking tend to reduce, but remain still solid (~50%) with 8 parallel replayers and 1 W/TX.

These result can be explained by considering that the heap size affects the locality of the writes issued in the replay phase

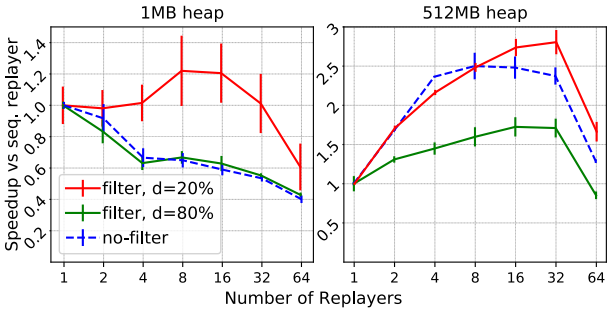


Figure 7: Speedup of parallel replay for 1MB and 512MB heap. The no-filter approach is compared with filtering for two levels of duplicates in the log: 20% and 80%.

and to what extent this write traffic can be served within the CPU cache (22MB in our case): if the writes can be replayed in cache, their relative cost decreases, amplifying the gains stemming from using an efficient mechanism to determine which transaction to replay next. Analogously, the number of writes per transaction affects the relative frequency of use linking and sorting. In fact, we see that generally the fewer the writes per transaction, the larger the gains of linking[5].

Parallel replay. Next, in Fig 7 we study how varying the degree of parallelism affects the speed-ups achievable w.r.t. sequential replay. We consider in this study also a version of the log replay that exploits the backward filtering technique [4], and use our synthetic benchmark to generate logs with 20% and 80% of duplicate writes.

The right plot, which considers a 512MB heap, shows peak gains of up to 2.8×. The use of filtering favours the scalability of the parallel replay technique and the maximum speed-ups are obtained for 20% of duplicates. This can be explained by considering that filtering reduces the write traffic towards PM, which represents the bottleneck in the no-filter scenario. For the case of 80% duplicates, though, filtering also reduces substantially the amount of writes that are effectively generated during the replay process. Accordingly, this reduces also the opportunities from benefiting from the proposed parallel log replay, which explains why the absolute speedups decrease as the duplicates' level grows from 20% to 80%.

With small heaps of 1MB (left plot), the efficiency of the parallel log replay degrades significantly. Only for the case of filtering with 20% of duplicates we observe speedups of ~20% (at 8 threads). In the other considered scenarios, parallelism ends up hindering performance. This can be explained by considering that writes to such a small heap are served entirely in the processor's cache and that the existence of a (possibly large) number of replayers intensively updating such a small working set is likely to generate strong contention and interference in the cache subsystem. Although this result pinpoints a limitation of the proposed technique, we argue that most applications that make use of large scale multicore machines and PM will likely adopt much larger heaps.

[5]Except for the case of 1 thread and heaps smaller than 8MB, arguably due to caching effects.

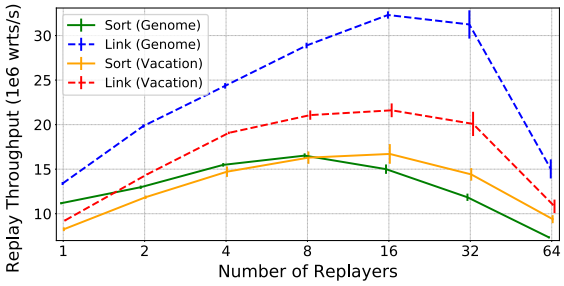


Figure 8: Log replay in VACATION_LOW and GENOME.

replayers	1	2	4	8	16
VACATION_LOW	8.64% (±0.03)	6.32% (±0.01)	5.81% (±0.03)	5.20% (±0.02)	4.75% (±0.01)
GENOME	8.85% (±0.04)	5.76% (±0.05)	4.99% (±0.06)	4.26% (±0.03)	3.82% (±0.05)

Table 2: L1 cache misses in the replay phase using linking.

Next we evaluate the joint use of parallel replay and linking, this time using realistic benchmarks, namely, VACATION_LOW and GENOME (shown in Fig 8). The proposed parallel log replay scheme has better throughput when compared to a conventional sorting approach, yielding ~1.3× and ~2.1× peak speedup, resp., for VACATION_LOW and GENOME at 16 threads. The joint use of linking further amplifies the speedups of parallel replay by an additional 35%, demonstrating how these two techniques can be effectively employed in synergy to accelerate the log replay process.

Finally in table Table 2 we report the L1 cache misses when varying the number of replayers from 1 to 16. We can observe that the cache misses decrease as the parallelism increases. This is expected, since all the replaying threads scan the whole log (i.e., generate the same stream of read accesses), confirming that this cost is amortized by an increase in the cache hits as the thread count increases.

5 Conclusions

This paper pinpointed several scalability limitations that affect existing PTM systems for off-the-shelf HTM. We tackled these limitations by proposing SPHT, a novel PTM system that integrates a number of innovative techniques targeting both the transaction processing and the log replay phases.

We evaluated SPHT in a system equipped with Intel Optane DC PM and compared it against other 5 state of the art PTM systems that had been so far only evaluated via emulation. SPHT achieves of up to 2.6× throughput gains during transaction processing, when compared to the most competitive baseline, accelerating log replay by up to 2.8×.

Acknowledgments

This work was partially supported by FCT (UIDB /50021/2020), FAPESP (2018/15519-5, 2019/10471-7) and EU's H2020 R&I programme (EPEEC project, GA 801051).

References

[1] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proceedings of the VLDB Endowment*, 10:409–420, Nov 2016.

[2] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'16, page 121–132, New York, NY, USA, 2016. Association for Computing Machinery.

[3] M. Cai, C. C. Coats, and J. Huang. Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596, 2020.

[4] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79, 2019.

[5] Daniel Castro, Paolo Romano, João Barreto, and Alexandro Baldassin. Scalable persistent hardware transactions. Technical Report 1, INESC-ID, January 2021.

[6] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, Seattle, WA, USA, 2008. IEEE.

[7] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European Conference on Computer Systems*, pages 27–40, Paris,France, 2010. ACM.

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS'11*, 47(4):105–118, jun 2011.

[9] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures - SPAA'18*, pages 271–282, Vienna, Austria, 2018. ACM Press.

[10] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, 41(1):381–394, March 2013.

[12] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, June 1984.

[13] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 3–14, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2020*, London, UK, 2020. ACM.

[15] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of htm transactions in nvm. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management - ISMM'17*, pages 70–81, Barcelona, Spain, 2017. ACM Press.

[16] Ellis Giles, Kshitij Doshi, and Peter Varman. Hardware Transactional Persistent Memory. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS'18, pages 190–205, Alexandria, Virginia, USA, October 2018. ACM.

[17] Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A. McKee, and Per Stenstrom. Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell. In IEEE, editor, *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 615–624, Phoenix, AZ, may 2014. IEEE.

[18] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, and V. Vlassov. Bandwidth-aware page placement in numa. In *34th*

IEEE International Parallel & Distributed Processing Symposium (IPDPS 2020), IPDPS'20, New Orleans, Louisiana USA, 2020. IEEE.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[20] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware logging in transaction systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.

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

[22] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, pages 313–327, Paris, France, 2016. Springer.

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[24] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *45th Annual International Symposium on Computer Architecture - ISCA'18*, pages 452–465, Los Angeles, CA, USA, June 2018. ACM.

[25] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. In *Proceedings of the twenty first international conference on Architectural support for programming languages and operating systems - ASPLOS'16*, volume 51, pages 399–411, Atlanta, Georgia, USA, 2016. ACM.

[26] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with TimeStone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 335–349, New York, NY, USA, 2020. Association for Computing Machinery.

[27] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 335–349, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Mengxing Liu, Mingxing Zhang, Kang Chen, and Xuehai Qian. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'17*, pages 329–343, Xi'an, China, 2017. ACM Press.

[29] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence in transactional persistent memory. *IEEE Symposium on Mass Storage Systems and Technologies*, 2015-August(1), 2015.

[30] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys'17, page 499–512, New York, NY, USA, 2017. Association for Computing Machinery.

[31] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[32] Peter Bergner, Alon Shalev Houfater, Madhusudnanan Kandeasamy, David Wendt, Suresh Warrier, Julian Wang, Bernhard King Smith, Will Schmidt, Bill Schmidt, Steve Munroe, Tulio Magno, Alex Mericas, Mauricio Oliveira, and Brian Hall. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2015.

[33] Peter Bergner, Alon Shalev Houfater, Madhusudnanan Kandeasamy, David Wendt, Suresh Warrier, Julian Wang, Bernhard King Smith, Will Schmidt, Bill Schmidt, Steve Munroe, Tulio Magno, Alex Mericas, Mauricio Oliveira, and Brian Hall. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, IBM, 2015.

[34] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? *SIGPLAN Not.*, 45(5):47–56, January 2010.

[35] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *Proceedings of the 2015 International*

Conference on Parallel Architecture and Compilation (PACT), PACT '15, page 445–456, USA, 2015. IEEE Computer Society.

[36] Storage Networking Industry Association (SNIA) Technical Position. NVM Programming Model Version 1.2, jun 2017.

[37] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11.0.

[38] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS'11*, pages 91–104, Newport Beach, California, USA, 2011. ACM Press.

[39] Darius Šidlauskas, Simonas Šaltenis, and Christian S. Jensen. Processing of extreme moving-object update and query workloads in main memory. *The VLDB Journal*, 23(5):817–841, October 2014.

[40] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen. Persistent transactional memory. *IEEE Computer Architecture Letters*, 14(1):58–61, Jan 2015.

[41] Zhenwei Wu, Kai Lu, Andy Nisbet, Wenzhe Zhang, and Mikel Luján. Pmthreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20)*, US, June 2020. ACM.

[42] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.