FUR: Fast and Unlimited Reads on Persistent Memory Transactions

João Barreto
INESC-ID, IST, Universidade de Lisboa, Portugal
joao.barreto@tecnico.ulisboa.pt

Daniel Castro
INESC-ID, IST, Universidade de Lisboa, Portugal
daniel.castro@tecnico.ulisboa.pt

Paolo Romano
INESC-ID, IST, Universidade de Lisboa, Portugal
paolo.romano@tecnico.ulisboa.pt

Alexandro Baldassin
São Paulo State University (Unesp), Brazil
alexandro.baldassin@unesp.br

Abstract

Despite the recent improvements in supporting Persistent Hardware Transactions (PHTs) on emerging persistent memories (PM), they have largely overlooked the poor performance of Read-Only (RO) transactions, which suffer from two crucial bottlenecks: i) the considerable post-commit delays required to ensure consistency with concurrent update transactions; and ii) the well-known tight read capacity limits of the commercially available HTM implementations.

We propose FUR, a new design for PHT that eliminates the two most crucial bottlenecks that hinder RO transactions in state-of-the-art PHT. At its core, FUR exploits advanced instructions that some contemporary HTMs provide to suspend (and resume) transactional access tracking.

With the exception of workloads comprised of 100% update transactions, our experimental evaluation with an IBM POWER9 system using the TPC-C benchmark shows that FUR can outperform the state of the art designs for persistent hardware (SPHT) and software memory transactions (Pisces or SpecPMT-based) by up to 6.17×.

CCS Concepts: • **Computer systems organization** → *Multicore architectures*; *Processors and memory architectures*; • **General and reference** → General conference proceedings; • **Information systems** → *Information storage systems*; • **Hardware** → *Memory and dense storage.*

Keywords: Hardware transactional memory, persistent memory transactions, persistent memory

ACM Reference Format:
João Barreto, Daniel Castro, Paolo Romano, and Alexandro Baldassin. 2026. FUR: Fast and Unlimited Reads on Persistent Memory Transactions. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3767295.3769343

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769343

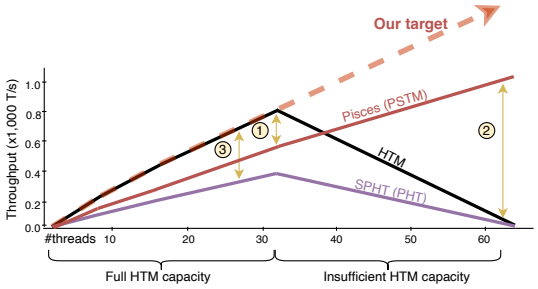

Figure 1. Throughput of RO transactions in a read-dominated TPC-C workload (*Payment* transactions on 1 thread + multiple threads running *Order status* transactions).

1 Introduction

The emergence of the Compute Express Link (CXL) technology [22] has sparked a profound shift towards high-capacity and high-throughput Persistent Memory (PM) devices. With CXL 2.0, the first commercial PM devices are already under development [45, 53, 57] and, in a near future, disaggregated CXL 3.0-based PM [28] will become a reality. The rise of byte-addressable PM allows bridging the gap between volatile and scarce memory vs. persistent and large disk storage, which has guided decades of software design.

At first glance, the low synchronization overheads of Hardware Transactional Memory (HTM) [40] make this technology a attractive solution for handling concurrent transactions accessing shared data that is durably stored in PM. However, existing HTM implementations do not guarantee that updates generated by a committed transaction are atomically transposed from the (volatile) CPU cache to PM [9, 13, 29, 30]. A number of proposals of Persistent Hardware Transactions (PHT) extend off-the-shelf HTM implementations with software-based schemes, which ensure that the effects of hardware transactions on PM remain consistent in the presence of system crashes [12, 13, 29, 30, 47].

Read-only (RO) transactions are dominant in many real-world workloads [11, 17, 20, 21, 23, 48, 52, 55, 60, 67]. Therefore, the design of PHT should prioritize their performance. However, while the state of the art in PHT has achieved

important improvements on the scalability of update transactions [12], it has largely overlooked the poor performance of RO transactions.

To support this claim, we set out a simple experiment where a variable number of threads concurrently execute *order status* RO transactions from the TPC-C benchmark [1], while a single thread continuously runs *payment* update transactions. We use this workload to study three representative systems: SPHT [12], a PHT; Pisces [32], a Persistent Software TM (PSTM); and plain HTM, as a non-durable baseline. We use an IBM POWER9 machine with a total of 32 cores, supporting HTM and Simultaneous Multithreading (SMT) (for detailed specifications, see §4). Up to 32 threads, each thread runs alone on a dedicated core; beyond that, thread pairs are co-located in the same core, which roughly halves the per-thread transactional capacity.

Figure 1 shows the RO transaction throughput of each solution, as we increase the number of RO threads. Despite the promises of HTM-based solutions, their performance is disappointing, while PSTM is the most competitive. This can be explained by three main observations, depicted in the plot. (1) HTM is faster than STM when running with enough capacity (up to 32 threads). (2) As soon as capacity drops, read sets no longer fit in HTM; thus, HTM-based solutions start thrashing with capacity aborts and falling back to the single global lock path, a well-known limitation [15]. (3) Even in the HTM-friendly zone (up to 32 threads), SPHT imposes a substantial overhead, which makes it less competitive than STM. In fact, to guarantee consistency in the presence of crashes [58], any RO transaction, *R*, that completes its execution must wait until any write that *R* might have observed is already durable. In SPHT, this is enforced by a *durability wait* that only terminates after any transaction that either executed before or concurrently with *R* has become durable.

The main goal of this paper is to eliminate the fundamental bottlenecks of RO transactions in PHT, thereby achieving "our target" in Figure 1. We take advantage of instructions for suspending (and resuming) access tracking inside hardware transactions, which are offered by commercial HTM implementations. We draw inspiration from existing proposals that exploit such instructions to relieve hardware transactions from the read capacity bounds of HTM [27]. Their approach is to suspend load tracking, which grants the transaction unlimited read capacity. To prevent a potential breach of isolation, update transactions perform an *isolation wait* before they can safely commit in HTM.

As a first contribution, we empirically demonstrate that naively combining existing proposals in PHT designs and unlimited reads techniques incurs prohibitive overheads.

These findings motivate our **second contribution**, the design of *Fast and Unlimited Reads (FUR)*. FUR relies on two main building blocks: a novel *RO durability wait* that spares RO transactions from the need to wait for concurrent update transactions to become durable; and, a set of new techniques (*opportunistic redo log flushing* and *partially-ordered durability markers*) that exploit the *isolation wait* as an opportunity to rethink the expensive durability stages of update transactions. These building blocks together with the unlimited reads feature allows FUR to overcome the two key limitations depicted in Figure 1: *HTM capacity* and *durability overheads*.

FUR can ensure either opacity [33] (the established correctness criteria in transactional memory literature) or the weaker Snapshot Isolation (SI) [10, 16, 24]. This choice entails a trade-off: with opacity, only RO transactions benefit from unlimited reads; with SI, all transactions do.

As a third contribution, we implement and experimentally evaluate FUR on an IBM POWER9 system. Using diverse workloads from the TPC-C benchmark [1], we compare FUR with systems that, to our knowledge, are among the most competitive proposals for durable memory transactions on PM: SPHT [12] (PHT), Pisces [32] (reader-optimized PSTM), and a PSTM based on the recent SpecPMT logging method [66]. Our results show that, with the exception of workloads comprised of 100% update transactions, FUR consistently surpasses all the state-of-the-art systems at each workload, outperforming SPHT, Pisces and SpecPMT at peak throughput by up to 3.68×, 3.99× and 6.17× (resp.).

We believe that, by achieving these gains, FUR repositions HTM as a key building block for the new generation of applications to embrace the upcoming CXL-based PM technologies. Our results also highlight the power of suspend/resume access tracking in HTM. In a hardware/software co-design perspective, we believe that our results constitute solid evidence that the recent trend led by Intel of introducing instructions for suspending load tracking in its latest processors should be generalized to suspension of both load and store tracking, as well as be adopted by other microprocessor manufacturers. Further, we also discuss how a subset of FUR's features can be adapted to today's off-the-shelf Intel HTMs.

The paper is organized as follows. §2 presents background and related work, as well as a preliminary study that highlights the challenges that lie ahead. §3 describes the design of FUR. §4 experimentally evaluates FUR. Finally, §5 discusses our work and §6 concludes it.

2 Background and Related Work

This section provides background on HTM and surveys recent proposals to extend HTM with durability and with unlimited reads. We start by describing existing HTM systems in §2.1. Then, we survey the state of the art on improving hardware transactions with durability (§2.3) and unlimited reads (§2.2). Finally, in §2.4 we show that, although techniques from the two previous sections can be directly combined, such combination yields disappointing performance.

2.1 Hardware transactional memory

HTM provides a hardware-based implementation of the familiar abstraction of atomic transactions. As of today, HTM implementations are offered by major CPU manufacturers such as Intel [69], ARM [5] and IBM [44]. Despite their differences, all commercially available HTM implementations keep track of the transactions' accesses within per-core caches. This implies two fundamental limitations.

A **first limitation** stemming from the cache-centric design of current HTM systems is that current HTM systems abort when a transaction's footprint exceeds the transactional cache capacity. As such, applications must rely on an inefficient Single Global Lock (SGL) fallback mechanism.

The **second limitation** is that, upon the commit of a transaction, its updates are not atomically flushed to PM. This becomes problematic if hardware transactions access persistent objects residing on a PM device. In fact, if the system crashes after a hardware transaction has committed, some of its writes on PM locations may not have yet persisted. Hence, they may be lost when the system later recovers.

Besides the basic primitives for transaction demarcation (`htmBegin`, `htmCommit`, `htmAbort`) some HTM implementations provide extensions that let the program temporarily suspend the tracking of a certain kind of memory accesses, and later resume it. During a suspend-resume window, the *untracked* memory accesses are not added to the local transactional cache, hence they neither take up the capacity of such cache, nor are they considered for conflict detection. Evidently, inadvertently suspending access tracking can lead to isolation anomalies.

Among today's landscape of commercial HTM implementations, IBM POWER CPUs provide the richest set of primitives. The program can, at any point in a transaction's lifetime, suspend tracking of *any access* and later resume. Furthermore, tracking suspension of loads (but not stores) can be requested for the whole duration of a transaction by passing a specific flag to `htmBegin`. Transactions of this kind are also called *Rollback-Only Transactions* (ROTs) [44].

As for Intel, the latest version of its Transactional Synchronization Extensions (TSX) has introduced the possibility of suspending load tracking (with the new `XSUSLDTRK` and `XRESLDTRK` instructions [2]). On the other extreme of the spectrum, ARM's Transactional Memory Extension (TME) does not currently allow any control over access tracking.

Our work exploits access tracking suspension primitives in novel ways. From here on, we consider the richest set of primitives, as offered by IBM POWER. Later on, in §5, we also discuss how our proposal can be adapted to Intel's HTM.

2.2 Unlimited reads with HTM

Recent proposals extend unmodified HTM implementations with a software layer that is able to stretch or even eliminate the *read* capacity limitations of HTM [26, 27, 36, 37]. To our knowledge, these proposals focus on volatile hardware transactions (not PHT). SI-HTM [27] currently stands out as the only solution that is able (i) to grant unlimited reads to both RO and update (hardware) transactions, while (ii) enabling multiple update transactions to run concurrently. SI-HTM relies on the primitives for suspending access tracking. In SI-HTM, update transactions are executed entirely with no load tracking, which grants them unlimited reads. RO transactions run outside any transactional context, thus also benefit from unlimited reads, as well as no HTM overhead.

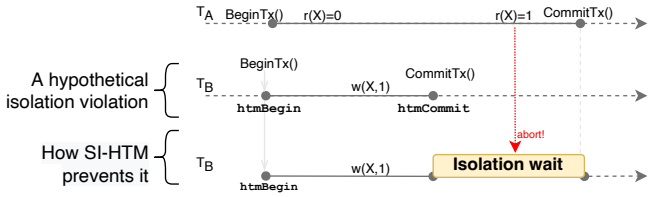

Figure 2. How SI-HTM prevents isolation violations.

This raises a consistency challenge: concurrent transactions with untracked loads may lead to isolation anomalies. In the example in Figure 2, transaction T_A reads twice from the same memory location, X, whereas transaction T_B concurrently modifies X and commits. Since T_A's first read happened before the concurrent write and the second read was after the write became committed, T_A observes two different values from the same location, which constitutes an undesirable *non-repeatable read* anomaly [10]. (Note that, if these reads were tracked by the HTM, it would abort T_A, thus preventing the anomaly.)

SI-HTM prevents this anomaly by making update transactions that reach `CommitTx()` wait until every concurrent transaction that is active at that point is no longer active; i.e., has performed its last transactional access, or aborted. We call this an *isolation wait*. Only after completing the isolation wait can an update transaction commit in HTM.

The bottom of Figure 2 illustrates how the isolation wait prevents the isolation anomaly in the previous example. The isolation wait ensures that, if a waiting update transaction (e.g., T_B) has performed any write that is subsequently read by a concurrent transaction (e.g., T_A), a read-after-write conflict will occur while the former is waiting. The HTM detects the conflict (since the write is tracked) and aborts one of the conflicting transactions. This holds in all contemporary HTM implementations that support access tracking suspension (IBM POWER and Intel). If the reader is an RO transaction (thus, running outside HTM), then the writer is always the victim, since an RO transaction does not run in HTM.

To put the above recipe into practice, each thread in SI-HTM shares a variable that advertises the thread's current state, which can be: *inactive*, not executing a transaction; *active*, executing a transaction; or *waiting*, performed the

last memory access but still waiting for safety. For update transactions, the transition from *active* to *waiting* happens before the transaction has committed in HTM. To ensure that such state transition is immediately observable by other threads (which might also be in their isolation wait), the thread first suspends all access tracking, the new (*waiting*) state of the transaction is externalized and access tracking is then resumed again. Finally, the thread scans the state of every other thread and spins until any thread initially found in the active state has transitioned to a different state. As soon as that condition is met, the isolation wait is over and the thread can commit in HTM.

It can be proven [27] that this isolation wait guarantees a fundamental property:

Property 1. *Given any pair of concurrent transactions (i.e., with overlapping* beginTx *invocation to* commitTx *response intervals), neither one can read from the writes performed by the other.*

Filipe et al. [27] show that this property is sufficient to provide SI. However, it does not ensure that concurrent update transactions are opaque [33] among each other. A simple workaround to achieve opacity is to run update transactions as full HTM transactions (i.e., tracking loads). A drawback is that update transactions loose unlimited reads – only RO transactions retain that advantage.

2.3 Persistent hardware transactions

Expanding the abstraction of TM to interoperate with PM in architectures where CPU caches remain volatile – as in the ADR persistence domain [56] – requires additional mechanisms. The usual requirement is to ensure that opacity [33] (the established correctness criteria in transactional memory) is preserved even if the system crashes and recovers, which is captured by the formal notion of durable opacity [58].

A traditional approach is to use Write Ahead Logging (WAL) techniques [42], in which a transaction logs all modifications to PM before actually modifying them. The first works on persistent TM were based on software (PSTM), namely Mnemosyne [59] and NVHeaps [19]. These early solutions catalyzed many subsequent improved PSTM designs [3, 8, 42, 43, 47, 63, 65, 66]. Among recent proposals, Pisces [32] is the most related to our paper, since its design also prioritizes the performance of RO transactions. Pisces achieves this by leveraging the weaker semantics of Snapshot Isolation (SI) [10, 16, 24].

Let us now focus on HTM. Since HTMs rely on cache coherency protocols and internal hardware buffers to handle synchronization, one cannot repurpose such mechanisms to use WAL without modifying their implementation [6, 7, 31, 38, 39, 61, 62]. The reason is that existing HTM implementations disallow flushing caches from within an active hardware transaction – the idea at the basis of WAL.

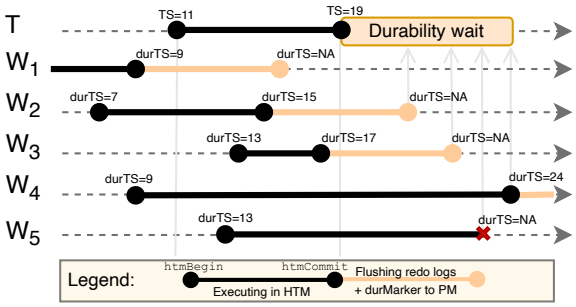

Figure 3. Example execution with SPHT.

The first systems to reconcile off-the-shelf HTMs with durability guarantees were proposed by cc-HTM [30], DudeTM [47], NV-HTM [14] and Crafty [29]. More recently, SPHT [12] revised such designs to achieve enhanced scalability. All these proposals (except for Crafty) share a common design backbone. They rely on the OS's Copy-on-Write (CoW) support to transparently create a volatile working snapshot. Transactions operate on such a snapshot. Transactional writes to the volatile snapshot are described in durable per-thread redo logs maintained in PM. Such redo logs are replayed on the original persistent heap in PM, typically in background, in order to bound the log's growth. Upon recovery, the durable redo logs are also replayed to reconstruct the durable contents of the persistent heap.

The life cycle of a transaction is illustrated in Figure 3. It comprises two phases. First, the *non-durable phase* occurs when a transaction, T, executes in HTM. During execution, for each write (performed on the volatile snapshot), an entry describing the write is added to a per-thread redo log in PM. Once T has performed its last access, T obtains a physical timestamp, also known as *durability timestamp (durTS)*, before committing in HTM. This timestamp determines the order in which T's redo log will be replayed, relative to other transactions' redo logs. Since *durTS* is obtained from within the hardware transaction, it is guaranteed that the *durTS* order is consistent with the transaction serialization order determined by the HTM [14].

Each thread's *durTS* is a shared variable that can be observed by other threads. Since assigning a physical clock to *durTS* from within the hardware transaction may lead to spurious aborts, recent PHT solutions adopt the following optimization. Before committing in HTM, T reads the physical clock to a private variable. Only after T has committed in HTM does T advertise the private timestamp in T's *durTS*. For the sake of correctness (as we discuss next), a committed transaction cannot hold a null *durTS*. Therefore, each thread that is about to begin a transaction starts by assigning an initial, conservatively low value to its *durTS*.

After executing, T commits in HTM and, if successful, enters the *durability phase*, which ensures that its writes

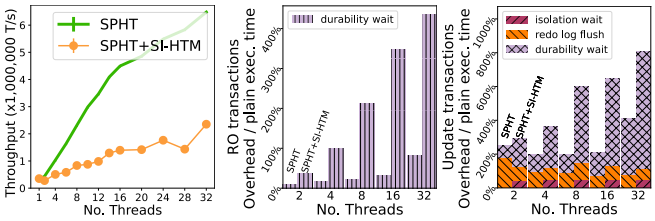

Figure 4. Comparing the throughput of SPHT and SPHT+SI-HTM, in a mix of 95% *order status* and 5% *payment* transactions from TPC-C, with disjoint warehouse accesses

are durable – i.e., will be visible upon recovery if the system crashes – before the thread returns to the application. Firstly, T flushes each redo log entry. Each transaction is only considered durable after it has persisted a *durable* marker (*durMarker*) that includes *durTS*. However, flushing the *durMarker* to PM requires careful coordination with other threads, in order to ensure that an update transaction T does not become durable (i.e., flushes its *durMarker*) before any update transaction from which T has read.

A possible approach to ensure the above property is to have T first carry out a *durability wait*, in which a transaction waits until every transaction with a lower *durTS* has either become durable or aborted. Only then can the transaction flush its *durMarker* to PM and return to the application.

The durability wait is a well-studied bottleneck [12] and Figure 3 illustrates why. Let us consider the durability wait conducted by transaction T. Since W_1, W_2 and W_3 committed before T, T may have observed their writes. Hence, T must wait until these transactions become durable. Still, there are less intuitive scenarios that force T to wait also for transactions whose writes it never observes. This is the case with transactions W_4 and W_5: W_4 will commit after T, whereas W_5 will simply abort. However, when T checks their *durTS*, it sees a lower value, which was conservatively set before they began. Therefore, T must (spuriously) wait until their *durTS* are updated (either upon commit or abort).

Since RO transactions produce no persistent effects, they skip the redo log and *durMarker* flush logic. However, they still have to go through the durability wait. Otherwise, an RO transaction could externalize (to the application program) the effects of transactions that are not yet durable. As Figure 1 has shown, the RO durability wait represents a strong performance penalty that, to our knowledge, no prior work in PHT has addressed.

2.4 Why not just combining durability with unlimited reads in PHT?

To the best of our knowledge, no prior work has unified durability and unlimited reads in hardware transactions. Yet, such unification is relatively straightforward to achieve. We start by adopting SPHT's architecture and design. Further, we inject the main features of SI-HTM into update transactions, namely: run without load tracking and perform SI-HTM's isolation wait (before `htmCommit`). Finally, we replace the `htmBegin`/`htmCommit` instructions from RO transactions by routines that implement the *active*/*inactive* state transitions.

As a first exploratory step, we have implemented this no-frills SPHT+SI-HTM combination and experimentally compared its performance with SPHT. Unfortunately, this revealed a prohibitive side-effect of SPHT+SI-HTM.

To understand it, Figure 4 presents results obtained when running both systems with a mix of 95% *order status* RO transactions and 5% *payment* update transactions. We disabled SMT and restrict each thread to access a disjoint warehouse. The former setting grants large-enough transactional caches to prevent capacity aborts. The latter ensures negligible aborts due to transactional conflicts. Hence, we can compare the durability overheads without noise incurred by different capacity/conflict abort rates of each solution.

The left plot (in Figure 4) shows that SPHT+SI-HTM exhibits considerably lower throughput than SPHT. This disadvantage can be explained by two decisive differences that stand out when we analyze the latency profile of each type of transaction. First, the middle plot shows that RO transactions spend up to 7× more time in their durability wait in SPHT+SI-HTM (than in SPHT). This is explained by a cascade effect: by injecting an isolation wait in update transactions, they take longer to commit in HTM (see the right plot); consequently, the durability wait conducted by RO transactions also takes longer to complete.

Moreover, update transactions are also strongly penalized, as the right plot shows. Not only do they need to include the isolation wait in their critical path, but also the post-HTM durability routines of SPHT now take considerably longer – for the same reason as above.

3 FUR

The previous preliminary study shows that, to effectively address the evident performance bottlenecks that state-of-the-art PHTs inflict on RO transactions (recall §1), we need much more than a no-frills combination of techniques for durability and unlimited reads.

To accomplish this main objective, the design of FUR combines some core mechanisms of SPHT and SI-HTM, and deeply rebuilds them by employing three innovative techniques. First, we propose a *pruned RO durability wait* that spares RO transactions from the need to wait for concurrent update transactions. In practice, this optimization reduces the RO durability overheads to a negligible level. We also introduce two techniques, *opportunistic redo log flushing* and *partially-ordered durability markers*, that substantially optimize the durability steps of update transactions in FUR, when compared to SPHT. This compensates for the additional latency that FUR's *isolation wait* incurs on update transactions,

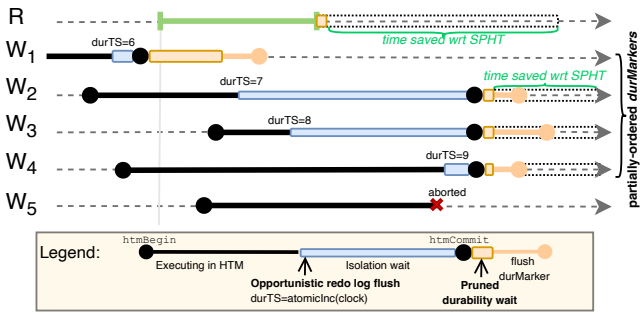

Figure 5. An example execution of FUR, highlighting its main optimizations

and even outweighs it in several update-dominated workloads, as we show in Section 4.3.

FUR adopts the base architecture of the state of the art in durable HTM (§2.3), in which transactions execute on a DRAM-hosted shadow copy of the persistent heap. FUR comprises two main processes: the *working process* and the *log replayer*. The working process has parallel working threads that are responsible for executing the application (transactional and non-transactional) code. When created, the working process receives an argument that specifies which isolation criterion it must enforce. FUR supports both SI and opacity. This choice entails a trade-off: with SI, every transaction benefits from unlimited reads; whereas, with opacity, only RO transactions do.

Transactions can access persistent locations that are originally stored in a data file residing in PM. The working process maps this PM file as a persistent heap in CoW mode (private mode in `mmap`). This mechanism transparently duplicates in DRAM any page that is altered. The working process also maps an additional persistent heap in shared mode, where it allocates the redo logs. Updates to this persistent heap are directly propagated to the PM, bypassing the OS page cache.

The log replayer process replays the durable logs produced by transactions into the persistent heap. It is activated upon recovery, but can also be executed during transaction processing either in background (to prune the logs) or synchronously (when the logs' capacity is exhausted).

Figure 5 showcases FUR's working process in action, highlighting the main optimizations that it employs. Its pseudocode is reported in Algorithm 1. In the following sections, we describe it in detail. We start with its non-durable phase (§3.1) and then proceed to the durability phase (§3.2). Finally, we address FUR's log replayer (§3.3).

3.1 Non-durable execution phase

We start by focusing on the *non-durable execution phase*, which operates on the volatile snapshot of the persistent heap. Each thread has a reserved entry in a shared `state` array (as in SI-HTM), which announces the thread's current

Algorithm 1 FUR

```
1: durMarkerArray, redoLog[N]                          ▷ Shared persistent variables (N is number of threads)
2: volatileRedoLog[N], state[N], durTS[N]              ▷ Shared volatile variables
3: isReadOnly, beginTime                               ▷ Thread local volatile variables
4: function BeginTx(isolationLevel)
5:     beginTime ← getTime()
6:     state[myTid] ← {active, beginTime}
7:     durTS[myTid] ← −1
8:     isReadOnly ← false
9:     memfence
10:    if isolationLevel=SI then
11:        htmBegin(noLoadTracking)
12:    else
13:        htmBegin(trackAnyAccess)
14: function BeginTxReadOnly()
15:    beginTime ← getTime()
16:    state[myTid] ← {active, beginTime}
17:    isReadOnly ← true
18:    memfence
19: function write(addr, val)
20:    logWrite(addr, val)                             ▷ add redo log entry (without flushing it)
21:    *addr← val                                      ▷ execute write
22: function CommitTx()
23:    if isReadOnly then
24:        state[myTid] ← {inactive}
25:        DurabilityWait()
26:    else
27:        htmSuspendAnyAccessTracking
28:        state[myTid] ← {inactive}
29:        memfence
30:        flushRedoLog(volatileRedoLog[myTid], redoLog[myTid])   ▷ Flush redo log
       opportunistically
31:        durTS[myTid] ← atomicInc(globalOrderTs)     ▷ Acquire the next durTS
32:        IsolationWait()                             ▷ Similar to SI-HTM
33:        state[myTid] ← {non-durable,getTime()}
34:        htmResumeAccessTracking
35:        htmCommit
36:        memfence
37:        DurabilityWait()
38:        FlushDurMarker(redoLog[myTid], durTS[myTid], durMarkerArray)
39:        state[myTid] ← {inactive}
40: function IsolationWait()
41:    state_snapshot[0..N-1] ← state[0..N-1]
42:    for c ∈ [0..N-1] : c ≠ myTid do
43:        if state_snapshot[c].isActive then
44:            wait while state[c] = state_snapshot[c]
45: function DurabilityWait()                          ▷ Pruned durability wait
46:    state_snapshot[0..N-1] ← state[0..N-1]
47:    for c ∈ [0..N-1] : c ≠ myTid do
48:        if state_snapshot[c].isNonDurable and state_snapshot[c].time < beginTime then
49:            wait while state[c] = state_snapshot[c]
50: function abortHandler()
51:    state_ts[myTid] ← [INACTIVE]
52:    if durTS[myTid] ≠ −1 then
53:        writeAbortMarker()                          ▷ Writes abort marker (asynch.)
```

state. Besides the original *inactive* and *active* states, we add a *non-durable* state, which we detail in the next section. The *active* and *non-durable* states are coupled with a physical timestamp, obtained from the CPU clock just before the thread enters such states. The physical timestamps serve two purposes in FUR, which we explain shortly.

When the application calls the *BeginTx* routine, the thread starts by setting its `state` to *active* (ln.6). In the case of RO transactions, *BeginTx* returns without starting any HTM transaction. For update transactions, a hardware transaction is started and *BeginTx* returns (as in SI-HTM). If FUR is configured to enforce opacity, a new hardware transaction is created as a regular HTM transaction. Otherwise, the hardware transaction starts without load tracking (as in SI-HTM).

Shared persistent location can be written via the *write* routine,which adds a new entry to the thread's redo log, describing the updated address and value, before performing the write. By definition, an RO transaction is not allowed to call the *write* routine, which can be enforced by adding an assertion. Still, RO transactions can execute *store* instructions to private volatile locations. This is analogous to prior PHT proposals (§2.3).

When the application intends to commit a transaction, it invokes the *CommitTx* routine. In the case of an RO transaction, the thread simply changes its state to *inactive* (ln.24). Otherwise, the thread takes the following steps to guarantee Property 1: it suspends any access tracking in the encompassing hardware transaction (ln.27), then it announces the *inactive* state (ln.28), before it calls the *IsolationWait* routine (ln.32). This routine waits until every other transaction that was active at the beginning of the isolation wait is no longer in that state (as in SI-HTM). An important detail in this routine is that the state timestamp lets the waiting thread disambiguate between an *active* thread that is still running a concurrent transaction (hence must be waited for), and the same thread after having completed the previous transaction and started a new one (at a higher timestamp). Finally, the thread changes its state to *non-durable* and resumes access tracking, before it finally commits in HTM.

Whereas SI-HTM only had one state transition in the beginning of its isolation wait, FUR requires an additional state transition just after the isolation wait's end. Therefore, FUR stretches the *suspended* window to cover both transitions. We have empirically confirmed that this strategy is more efficient than suspending/resuming twice.

3.2 Durability phase

As soon as a transaction becomes non-durably committed, it enters the *durability phase*. Recalling §2.3, this is a potentially cumbersome phase: RO transactions must undergo a potentially expensive durability wait; while update transactions execute even more costly steps (flushing the redo log, running the durability wait, and flushing the *durMarker*).

In FUR, we rethink each durability step, by proposing three novel optimizations. The first one, *pruned RO durability wait*, is able to practically hide the length of the durability wait of RO transactions. By combining this optimization with the unlimited reads feature (as described in the previous section), FUR is able to fulfill the main goal of our paper – to address the two bottlenecks that hinder RO transactions' performance in existing PHT systems (recall §1).

Since update transactions in FUR are penalized with a new time-consuming coordination step (the isolation wait), we compensate it with two optimizations to their durability phase, *opportunistic redo log flushing* and *partially-ordered durability markers*. The next sections describe these optimizations, focusing on RO transactions first (§3.2.1), then update transactions (§3.2.2 and §3.2.3).

3.2.1 Pruned RO durability wait.

Recall from §2.3 that an RO transaction, R, cannot return to the application before every transaction from which R read is durable. In SPHT, R conservatively waits for any update transaction that had already committed before R began, as well as any update transaction that executed concurrently with R.

Fortunately, we can take advantage of Property 1. A corollary of this property is that R will never read from update transactions that executed concurrently with R. Therefore, we can safely *prune* the RO durability wait to bypass any transactions that had not yet HTM-committed when R began. Concretely, in the *DurabilityWait* routine, a thread scans the state of all other threads and, whenever it finds any thread in *non-durable* state with a timestamp smaller than its own *active* timestamp, it spin-waits until that state changes.For simplicity, we assume perfectly synchronized hardware timestamp counters. As in prior research [14], this waiting step can be generalized to account for the maximum deviation offsets between any two clocks.

This optimization has a twofold advantage, which Figure 5 illustrates by revisiting the example from Figure 3. As a first advantage, by drastically reducing the number of transactions R needs to wait for, it returns much earlier to the application. In this example, R only needs to wait for W_1 to become durable, since every other (concurrent) transaction is bypassed. Furthermore, since the pruned set only comprises transactions that have committed earlier, it is likely that these are already durable by the time R enters the durability wait. This is the case with W_1 in our example.

In RO-dominated workloads, most accesses to the `state` array are writes by RO transactions (as they change state). These state transitions are read by the durability wait but have no effect on its logic. If left unaddressed, this interleaving of write/read accesses from different cores to the same cache lines would trigger many cache invalidations. We prevent this thrashing condition by unfolding the per-thread state into two arrays (not presented in Algorithm 1, for simplicity). The first array specifies whether a transaction is *active* or not; the second array tells whether a transaction is in the *non-durable* state or not. The *DurabilityWait* routine scans the second array, which is only changed by update transactions. Therefore, in RO-dominated workloads, cache invalidations on the second array become rare and *DurabilityWait* serves most of its reads from L1 cache.

3.2.2 Opportunistic log flushing.

Next, we focus on optimizations that reduce the overheads of the durability steps of update transactions by exploiting the isolation wait. We start by focusing on the redo log flushing step.

In every HTM implementation that we are aware of, hardware transactions cannot flush the cache lines they has written to (as that would externalize the writes). However, while a transaction is in (full) suspended mode, it is allowed to

perform (untracked) writes and to write-back (without invalidating) their cache lines, as long as such cache lines had not been accessed by the transaction before it suspended access tracking. Specifically, this is the behavior of IBM POWER.

FUR's isolation wait, which is performed in *suspended* mode, is an opportunity to exploit the above feature to *opportunistically* anticipate the flushing of the cache lines of its redo log entries. Two implementation details are crucial to accomplish this goal. First, the redo log entries that are written by the *write* routine are part of the transaction's write-set (in the transactional cache). Therefore, they cannot be flushed in suspended mode. To work around this restriction, each thread uses a volatile redo log (ln. 2) while executing transactionallyand, in the *suspended* window, copies each volatile redo log entry to a the persistent redo log, and only flushes the latter. Note that, in practice, the volatile log only needs to store the addresses of each write. Second, the flush instructions are issued asynchronously. Therefore, the thread can immediately proceed to the isolation wait while, in background, the CPU is flushing the cache lines to PM. After committing in HTM, the thread executes a memory fence (ln.36). This ensures that the transaction's *durMarker* will only be persisted after the redo-log entries are. Hence, if a crash occurs before the memory fence completes, no redo-log entries of this transaction will be replayed upon recovery.

In §4, we show that, in most cases, such flushes have already completed *before* this point. When this happens, the latency of redo log flushing is effectively hidden.

3.2.3 Partially-ordered durability markers. The two last steps of the durability phase of update transactions in FUR are notably simple. An update transaction performs the same pruned durability wait as RO transactions (see §3.2.1). Then, it can immediately flush its *durMarker* (ln.38).

Hence, FUR trades the total order among *durMarker*s – which, to our knowledge, underpins all prior PHT proposals – for a lightweight *partial order*. To illustrate, let us revisit the example in Figure 5. The *durMarker* of the concurrent transactions $\{W_2, W_3, W_4\}$ can be flushed in any order; they must only be ordered after W_1's. This obviates any need for coordination across concurrent transactions (e.g., SPHT's intricate group commit scheme [12]) at this stage.

The same advantages that we get by pruning the durability wait of RO transactions (see §3.2.1) are also granted to update transactions. Moreover, the same correctness arguments that we provide in §3.2.1 also apply to update transactions. Concretely, once an update transaction, T, that has gone through its durability wait, it is guaranteed that any transaction from which T might have read is already durable, so T can also safely become durable. This is a corollary of Property 1. In fact, if a set of concurrent update transactions is performing their durability phase and, due to a crash, only an arbitrary subset of them is durable upon recovery, we know that no write performed by the lost (concurrent) transactions can have been read by any durable transaction that survived.

3.3 Efficiently replaying partially-ordered logs

In prior PHTs, each logged transaction contains a *durMarker*. Hence, the log replayer (LR) needs to scan through every per-thread's log to determine the next update transaction to replay. This is a well-studied bottleneck [12]. To our knowledge, the only technique to avoid this scanning phase – SPHT's log linking scheme [12] – relies on totally-ordered *durMarker*s. Hence it is not compatible with FUR.

To achieve a similar scalability to SPHT while supporting partially-ordered *durMarker*, FUR organizes every *durMarker* in a global *durMarker* array (implemented as a circular array). The size of the array determines how many update transactions can be durable in the *durMarker* array before having their writes replayed on the persistent heap.

FUR uses logical timestamps as *durTS*, which serve as indexes to the *durMarker* array. As soon as a transaction, T, acquires its unique *durTS*, T becomes the owner of the corresponding entry in a global *durMarker* array. Each *durMarker* entry in the array includes: the start address of T's redo log, the number of entries, as well as T's *durTS*.

The *durMarker* global array fulfills our scalability requirement, since the LR thread can just traverse the *durMarker* array and sequentially replay the redo log pointed to by each *durMarker* entry that it reads. After a batch of entries is replayed, the tail of the circular array is advanced, which effectively frees such entries to new transactions.

The choice of logical timestamps is fundamental to enable the solution described so far. To ensure that the *durTS* obtained by a transaction is consistent with the isolation order determined by the HTM, the *durTS* needs to be acquired before the transaction HTM-commits. The straightforward solution of reading and incrementing the global clock using transactional loads and stores is well-known to harm scalability under contention [12, 47].

To avoid this shortcoming, FUR takes advantage of the *suspended* window of its isolation wait. Concretely, the logical timestamp is acquired via an atomic increment instruction executed from such window (ln.31). Since the atomic increment is not tracked by the HTM, it does not introduce transactional conflicts. Moreover, similarly to opportunistic redo log flushing (§3.2.2), the latency of the atomic instruction is normally hidden as it is overlapped with the (typically longer) isolation wait.

A transaction that has acquired a *durTS* and then aborts will produce a *hole* in the logical timestamp sequence. Flushing an *abort marker* in the abort handler of such transactions to the corresponding entry in the global *durMarker* array (ln.53) fixes the *hole*. A system crash can also create holes in the *durMarker* array. To illustrate this, let us recall the example in Figure 5, where transactions W_2 to W_4 run concurrently and commit in HTM. Transaction W_3 took longer

than W_2 and W_4 to flush its *durMarker* (this is possible due to FUR's partially-ordered *durMarker*s). Assume that a crash happens just before W_3 persists its *durMarker*. Hence, upon recovery, the array will hold a *durMarker* in the entries corresponding to the *durTS* of W_2 and W_4, and a hole between them. A crash-induced hole can be detected since it either has a null entry or an entry with an expired *durTS* (i.e., from a previous epoch of the circular array). Let us designate these as *unmarked* holes.

When the LR detects a hole, it jumps to the next valid entry. It is easy to show that there cannot be more than $n-1$ (crash-induced) *unmarked* holes before the last valid *durMarker* in the array. Therefore, as soon as the LR has found *n unmarked* holes, it is guaranteed that there are no more valid entries to replay and the LR can stop.

4 Evaluation

Our evaluation aims at answering the following questions:

1. *In RO workloads of different read footprints, what advantage do FUR's unlimited reads provide?* (§4.2)
2. *In update-only workloads, given the isolation wait penalty and the new optimizations to the durability phase of FUR, what is the net performance outcome of both factors*? (§4.3)
3. *In mixed workloads (combining RO and update transactions) do the main observations taken from the previous scenarios still hold?* (§4.4)
4. *What is the performance of FUR's LR?* (§4.5)

4.1 Experimental settings

System. We used a dual-socket IBM POWER9 machine equipped with DD2.3 CPUs at 2.3-3.8GHz and 1TB DRAM. Each CPU has 16 cores with simultaneous multi-threading (SMT). POWER9's HTM relies on a transaction tracking structure that is distributed across the L1 and L2 caches, whose architectural details can be obtained in [34]. In this architecture, HTM can only be used from a virtual machine (VM) [49], so we set up a QEMU/KVM VM with 64 virtual cores (mapped to the 32 physical cores, each with 2 SMT hardware threads) and 8GB DRAM. The suspend (and resume) access tracking instructions trigger a trap to activate KVM hypervisor, which implements (in software) the logic associated with these instructions [41]. This incurs a considerable penalty on such instructions. Executing both instructions in sequence takes from 350ns on a single thread, to 1500ns on 64 threads.

Since PM is not yet available for IBM POWER systems, we emulate the write latency of an Optane-like PM device connected via CXL each time a cache flush is requested. Using access latency figures from the literature [51, 64], we inject a spin loop of 310 ns on each cache line flush. This approach is analogous to previous works in the literature (e.g., [13, 47]), whenever PM is not available.

Table 1. Read and write footprints per transaction type

Benchmark	Transaction type	Read footprint	Write footprint
TPC-C	stocklevel	Very high (avg 122K)	none
	orderstatus	Moderate (avg 650)	none
	delivery	Very high (avg 86K)	Moderate (avg 30)
	payment	Low (avg 97)	Low (avg 5)
	neworder	High (avg 7.5K)	Moderate (avg 141)
Red-black tree	lookup	Low (avg 28)	none
	insert/remove	Low (avg 44)	Low (avg 6)

Evaluated solutions. We used a full implementation of both variants of FUR (FUR-opa and FUR-SI). They are implemented in C as a library that can be linked to any C/C++ application. Instrumentation of reads and writes is manual in our prototype but could be automated by compiler support.

The source code of our FUR implementation is publicly available at https://github.com/inesc-id/FUR.

We compare FUR against three systems that, to our knowledge, are the most competitive proposals for persistent memory transactions, touching diverse points in the design space:

- SPHT [12], a durably opaque PHT;
- Pisces [32], a read-optimized PSTM, with durable SI semantics;
- SpecPMT's speculative logging [66] coupled with TinySTM [25], which together form a durably opaque PSTM. (By design, SpecPMT only ensures durability and requires an additional concurrency control mechanism to provide isolation [66].) For simplicity, we refer to this combined solution as SpecPMT.

We used the full-fledged SPHT variant, with forward log linking enabled [12]. We also considered pure HTM (with SGL fallback), as a reference of the raw throughput of HTM.

For fairness, we implemented all systems in a common framework. The log replayer is disabled during transaction processing for all solutions that support asynchronous log replay (FUR and SPHT), which is coherent with previous papers in this category [12, 13]. Every solution based on HTM falls back to SGL after 10 retries. The results are an average of 3 runs, each taking 5 seconds.

Workloads. As benchmarks, we use a full implementation of the TPC-C OLTP benchmark [1]. It uses a B-tree implementation that is exempt from SI's consistency anomalies (e.g., write skews [16, 24]).

We also study a red-black tree implementing an *IntSet*, a widely used benchmark in TM literature [50]. The tree is initialized with 10K elements.

As Table 1 shows, the transaction types of both benchmarks cover a diverse read and write footprints mix.

We exclude STAMP [54], another established TM benchmark, from our evaluation, since its applications do not have transactions that are known to be read-only at begin time, and the percentage of effectively read-only transactions is relatively low.

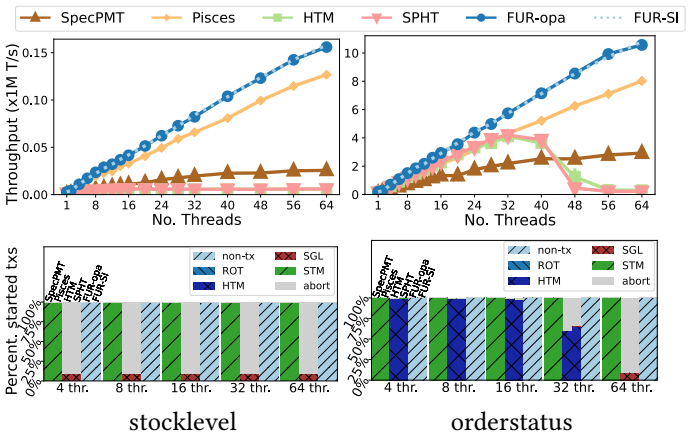

Figure 6. Throughput and transaction outcomes with RO TPC-C workloads

4.2 Processing RO workloads

Figure 6 presents the throughput and transaction outcomes when exclusively running *stocklevel* and *orderstatus* transactions. In both, the performance of FUR stand out when compared to their persistent competitors.

Since *stocklevel* transactions have large read footprints, they always capacity-abort when executed inside full HTM hardware transactions. This holds for SPHT as well as HTM. In contrast, the PSTM systems scale up to 64 threads, even though SpecPMT experiences very marginal gains with higher thread counts. In FUR, this is an expected consequence of the fact that RO transactions run without HTM. As in most STM implementations, Pisces and SpecPMT impose no limits on the read (or write) footprints. While FUR does not add any instrumentation to read accesses, Pisces/SpecPMT instrument each read to a shared location in PM with a software routine that checks the lock table to determine the latest version to read from. This explains the 18% and 84% performance gap between FUR and, resp., Pisces and SpecPMT in *stocklevel* at 64 threads.

In the *orderstatus* workload, the read footprints fit a full per-core transactional cache. Therefore, as long each thread runs in its own core (up to 32 threads), capacity aborts are negligible, thus the unlimited reads feature of FUR has no benefit when compared to SPHT/HTM. Yet, we can still observe a performance advantage of 30% due to the fact that SPHT/HTM need to issue `htmBegin`/`htmCommit` to activate the HTM support. Beyond 32 cores, SMT co-locates pairs of threads on a single core, hence the per-thread transactional caches no longer suffice for most transactions and we see the same trends of the *stocklevel* workload.

4.3 Processing update-only workloads

Figure 7 presents the results for the update-only *payment* and *delivery* workloads. In contrast to the previous section, we now include a third graph (bottom of Figure 7), which reports the overhead that committed transactions incur (on average) on the different steps that occur beyond the plain execution (i.e., the interval between the invocations to *BeginTx* and *CommitTx*), relatively to time spent executing the latter. For instance, an overhead of 200% for a given step means committed transactions spend 2× more time performing that step than their plain execution.

The *payment* workload has a small footprint and negligible capacity aborts. It is a worst-case scenario for FUR, since its *unlimited reads* feature brings no benefit, whereas the costs of its isolation wait stand out. Not surprisingly, SpecPMT outperforms the best FUR variant by 1.31x (at peak throughput). This can be explained by SpecPMT's efficient logging scheme (tailored to update transactions) and the scalability of the underlying TinySTM scheme (which scales relatively well under update-dominated workloads [25]).

It is interesting to notice that FUR-opa and FUR-SI outperform SPHT by 15% and 13% on average, respectively. The bottom plot in Figure 7 shows that, in SPHT, the overhead of the durability steps increases steeply after 8 threads, growing up to ≈150× the plain execution time at 64 threads. The durability wait is its major contributor.

In contrast, FUR's main source of overhead is the isolation wait, whereas the durability overheads are substantially reduced (up to 5× lower than SPHT's at 64 threads). This reduction stems from the two optimizations in FUR's durability phase. First, *opportunistic redo log flushing* reduces the critical-path waiting time to negligible (at 8 threads, which is peak throughput: 1% redo log flush overhead in FUR vs. 55% in SPHT). Second, the *partially-ordered* durMarkers are dramatically faster than SPHT's totally-ordered durability wait (at peak throughput, durability wait overheads of 60%-76% in FUR and 174% in SPHT). Overall, the savings attained by FUR's durability optimizations prevail over the performance penalty of the isolation wait.

When comparing the two FUR variants, it is perhaps surprising that FUR-opa achieves a higher throughput than FUR-SI, despite having higher abort rates. Analyzing abort codes and time spent on rolled-back transactions (available in our extended technical report [4]) shows that FUR-opa, by detecting conflicts earlier, spends considerably less time on rolled-back transactions. These savings outweigh the costs associated with increases in SGL acquisitions, suggesting that FUR-opa may be more competitive in high-contention workloads with small transactions.

Regarding Pisces, its reader-optimized design is not able to cope with update-dominated workloads, which explains why it is the worst-performing persistent alternative.

The next workload, *delivery*, is extremely prone to capacity aborts on HTM (moderately high write footprints and very high read footprints). In fact, with 1 thread, all solutions that run update transactions in full HTM (FUR-Opa, SPHT, HTM) exhibit a severe capacity abort rate (81%). At higher thread

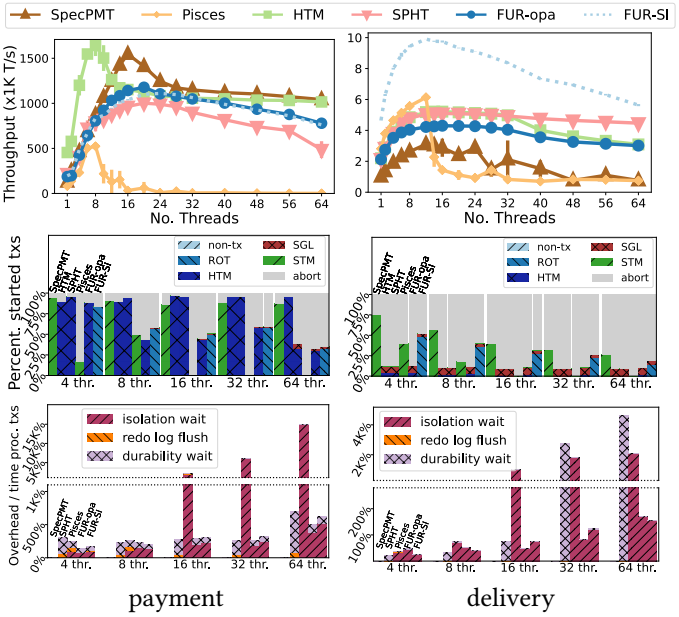

Figure 7. Throughput, transaction outcomes and overhead breakdown with update-only TPC-C workloads

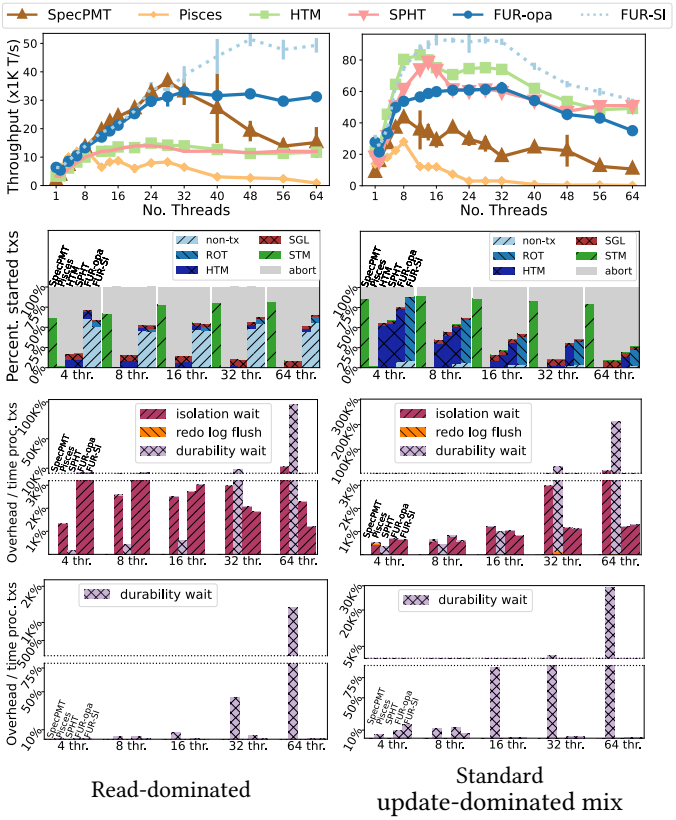

Figure 8. Throughput, transaction outcomes and overhead breakdown with mixed TPC-C workloads

counts, the scenario is worsened since contention induces transactional conflicts.

In contrast, since FUR-SI provides update transactions with unlimited reads, it exhibits substantially lower capacity abort rates (it still aborts nearly half of transactions due to write capacity). Therefore, FUR-SI attains dramatic throughput gains when compared to the HTM-based alternatives. Such gains even conceal the costs of FUR-SI's isolation wait.

Pisces and SpecPMT are the only solutions with both unlimited reads *and writes*. This allows them to outperform the solutions based on full HTM up to 12 threads. However, the costly access instrumentation of these PSTM approaches make them loose to FUR-SI by a substantial margin.

We omit the results for the *neworder* workload, as it would reveal similar observations to the ones drawn for *payment*. We consider *neworder* next, when we study mixed workloads.

4.4 Processing mixed workloads

Figure 8 presents our results with mixed workloads. We start by considering a read-dominated workload in Figure 8, in which 85% of transactions are RO (uniformly selected among *stocklevel* or *orderstatus*), and the remaining ones are from update types (*delivery*, *payment* or *neworder*). For the first time, we have RO transactions that, in their durability wait, actually need to wait for update transactions. RO transactions in SPHT spend a substantial portion of time in the durability wait (up to ≈10× the latency of the plain execution) until concurrent update transactions become durable (or abort). In contrast, the *pruned RO durability wait* of FUR is practically cost-less. A typical call to the corresponding routine finds the state array cached in L1 and does not find any transaction to wait for.

A collateral effect can be seen in the isolation wait latency of FUR. Since the minority of update transactions now have to wait for a majority of RO transactions, the relative cost of the isolation wait grows so much that it overcomes the durability overheads of SPHT. Even though the durability optimizations of FUR reduce the durability overheads to marginal levels, the net balance is that update transactions in FUR incur a higher overhead than in SPHT. Still, from the perspective of global throughput, this shortcoming is a means to achieve a greater good.

Pisces keeps up with FUR's pace up to 8 threads and achieves throughput that is competitive with SPHT, before contention effects start manifesting more eminently on update transactions, hindering the overall scalability. As for SpecPMT, it is able to scale better than Pisces (up to 28 threads) or even than FUR-opa, which shows that the underlying TinySTM concurrency control scheme is better at taming higher contention levels. Yet, SpecPMT/TinySTM cannot scale beyond 28 threads, whereas FUR can.

Finally, we look at the opposite extreme, an update-dominated mixed workload where 85% transactions are either *payment* or *neworder*, and the remaining 15% are from

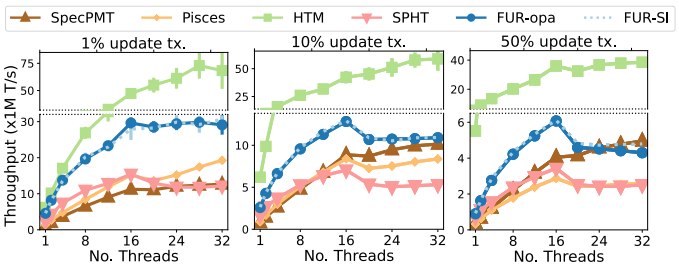

Figure 9. Red-black tree throughput with 1%, 10% and 50% update transactions (resp. left, middle and right plot)

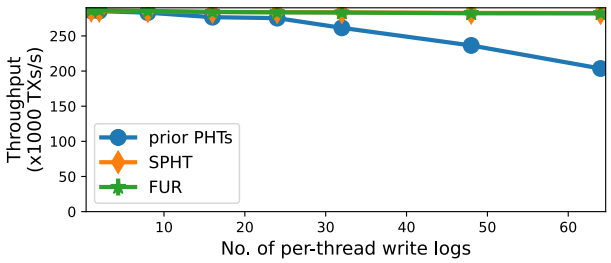

Figure 10. Log replay throughput of PHT alternatives

the other transaction types. This is the standard mix mentioned in the TPC-C specifications [1]. This workload is not favorable to FUR, not just because RO transactions only represent 10%, but also since the transactions prone to capacity aborts only account to 15%. Still, FUR-SI manages to be the most competitive solution. Relatively to SPHT, Pisces and SpecPMT, this advantage is explained by the same factors that we have observed with *payment* (§4.3), which also affect *neworder* – i.e., the two dominant transactions. Furthermore, FUR's RO durability wait yields similar savings as in the read-dominated mix. Comparing FUR-SI with FUR-opa, the less frequent capacity aborts in FUR-SI (which grants unlimited reads to both kinds of transactions) is the decisive factor behind its throughput advantage over FUR-opa.

These results enable us to breakdown the performance impact of the novel techniques that FUR employs (Section 3.2). In the read-dominated workload, the *pruned durability wait*, being the only optimization benefiting RO transactions, has a dominating impact. In update-dominated workload, both *opportunistic log flushing* and *partially-ordered durability markers* are highly effective optimizations; yet, the latter achieves the highest absolute reductions. This can be confirmed by comparing the redo log flushing and durability wait overheads of FUR-* to SPHT, in Figure 8.

Finally, we consider the red-black tree benchmark. Figure 9 presents results for mixed workloads where 1%, 10% and 50% (resp.) of transactions are insertions or removals, while the remaining operations are lookups (RO transactions). The relatively small tree height implies that capacity aborts are negligible, therefore the unlimited reads benefit of FUR is not evaluated here. Overall, both FUR variants achieve comparable peak throughput values in the three workloads, and are considerably more competitive than all persistent baselines. The instrumentation overhead of the STM solutions is evident, especially at low thread counts. At higher thread counts, Pisces' reader-optimized design scales better than SpecPMT in the read-dominated scenarios, but not in the 50% update case. SPHT performs akin of STM solutions due to the overhead of the durability wait.

4.5 Log replay

We now compare the performance of the Log Replayer (LR) of FUR with SPHT and the LR of prior PHTs (cc-HTM [30], DudeTM [47], NV-HTM [13]). We use the same methodology as in SPHT's paper [12]. First, we run a synthetic application with 100% update transactions to prefill the per-thread logs. Transactions issue between 1 and 20 writes (the number of writes and their accessed locations are chosen uniformly at random), the log size and heap size is 128MB, and we vary the number of working threads.

Next, we halt the application and run the LR to fully replay the logs and measure the replay throughput. For space limitations, we only study the case where the LR runs in a single thread, and we disable filtering of duplicated writes [13]. Using multi-threaded LR and duplicate filtering techniques is compatible with FUR's LR, and does not change the main conclusions that we draw next.

Figure 10 presents the throughput of the different replay schemes. The LR of prior PHTs struggles with an increasingly larger amount of working threads. This is caused by the need to scan, after replaying one transaction, every per-thread log to find the next transaction to replay. In contrast, SPHT and FUR prevent the scanning bottleneck, albeit by different means. In SPHT, this is accomplished with its *log linking* technique [12]; in FUR, via our novel LR scheme based on the global *durMarker* array. As shown in Figure 10, both LR schemes perform comparably and, most importantly, remain efficient even at high (worker) threads counts.

5 Discussion

Limitations. Similarly to any prior work based on off-the-shelf HTM, FUR does not circumvent the limited write capacity of HTM, so it is not suited to large write footprints. In opacity mode (FUR-opa), unlimited reads are only provided to those transactions that the program explicitly flags as RO when invoking `beginTx`, which can be an issue with legacy programs. In contrast, FUR-SI transparently grants unlimited reads to any kind of transaction.

On the power of suspending access tracking in HTM.

Supporting suspend/resume access tracking in HTM is well-known to incur higher complexity and cost in the design of the cache coherence protocol of a processor [18]. From a hardware-software co-design perspective, the hardware manufacturers will consider high-cost features only if they observe a clear demand by software developers. However, among academic research, we are only aware of a handful of works that exploit advanced suspend/resume features of HTM. To our knowledge, among software practitioners, the adoption of such advanced features is also negligible. We believe that our results contribute to a better awareness of the potential of suspend/resume among the research community and the microprocessor manufacturers. Our results also provide evidence that kernel-assisted suspend/resume implementations, while reducing hardware complexity, can still enable important gains, even in workloads that make intense use of such instructions.

Another dimension is how suspend/resume support can be implemented. Encapsulating the entire suspend/resume logic at the hardware level (e.g., IBM POWER8 [35] or Intel TSX [2]) reduces suspend/resume overheads at the cost of hardware complexity. An alternative to hardware-only implementations is to have the HTM issue a trap that activates the operating system kernel to assist in the suspend/resume handling. Our results, obtained in a system based on such alternative, provide evidence that kernel-assisted suspend/resume can still be a competitive sweet spot in the hardware-software co-design space.

Adapting our techniques to Intel TSX. FUR's design is incompatible with the latest generations of Intel TSX, which only enable suspending load tracking, but not *any-access* tracking. More precisely, the optimizations that we propose for the durability phase of update transactions (see §3.2) intrinsically depend on any-access tracking suspension.

However, the modules of FUR that benefit RO transactions (with unlimited reads and the pruned durability wait) can be adapted to an HTM that only supports load tracking suspension, as Intel TSX. To accomplish this, we can adapt the isolation wait to only scan (and spin on) the state of RO transactions. Consequently, update transactions no longer need to externalize their state transitions from within the suspend-resume window; therefore, only load tracking needs to be suspended in that window. Update transactions still need to advertise that they are entering a non-durable state, but this can be done just after access tracking is resumed (i.e., just before the HTM commit). This solution would retain the two major benefits that FUR provides to RO transactions. Understanding the design and performance trade-offs of this approach in a real Intel TSX system is left for future work.

Beyond the ADR persistence domain. In theory, there are stronger persistence domains than the one that we consider in this paper. A notable example is eADR [68], in which the CPU caches are also part of the persistence domain. Like a large body of literature on PM computing [8], our work targets systems where eADR is not available.

On the persist latency of future CXL-based PM. The emulated persist latency in our experiments (extra 310 ns) is likely to under-estimate the latency of some future PM-based CXL devices. In fact, recent empirical results [46] show that, although the latency of today's real CXL *volatile* memory expansion varies considerably, some devices can easily exceed our emulated latency. When one considers the future PM-equipped CXL devices, latency is naturally higher than the volatile alternatives. For instance, persistent CXL devices based on NAND flash (also known as memory-semantic devices [57]) have estimated persist latencies of 600 ns [45]. Since a crucial benefit of FUR is its ability to hide the indirect impact of persist latency on RO transactions (namely, by FUR's pruned RO durability wait), higher emulated latencies represent more favorable environments for FUR. The exploration of other latencies is left for future work.

6 Concluding remarks

We proposed FUR, a new design that eliminates two fundamental bottlenecks of durable RO transactions on HTM. At its core, FUR exploits instructions that some contemporary HTMs provide to suspend (and resume) transactional access tracking. Our experimental results show that FUR can outperform state of the art alternatives in both PHT and PSTM, by up to 6.17× in read-dominated workloads.

Acknowledgements

We thank the valuable feedback provided by the anonymous reviewers, our shepherd, Shuai Mu, as well as IBM and the Oregon State University Open Source Lab, in the name of Lance Albertson, for granting us access to the IBM Power machine.

This work was developed within the scope of the project no.62—"Responsible AI", financed by European Funds, namely "Recovery and Resilience Plan"—Component 5: "Agendas Mobilizadoras para a Inovação Empresarial", included in the NextGenerationEU funding program. The work was partially supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI:10.54499/UIDB/50021/2020), the European Commission through the Horizon Europe Programme, with the Grant Agreement GAP-101189689, and by FAPESP (2018/15519-5, 2019/10471-7).

References

[1] Transaction Processing Performance Council: TPC-C Benchmark Revision 5.11.0. http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, February 2010.

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3*, 2023.

[3] Mohammad Alshboul, James Tuck, and Yan Solihin. Lazy persistency: A high-performing and write-efficient software persistency technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 439–451, 2018.

[4] anonymous. Technical report to be disclosed once the paper is published. Technical report, anonymous, 2025.

[5] ARM. Arm c language extensions (2021q2). Online, july 2021.

[6] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proceedings of the VLDB Endowment*, 10(4):409–420, nov 2016.

[7] Hillel Avni, Eliezer Levy, and Mendelson Avi. Hardware Transactions in Nonvolatile Memory. In *LNCS 9363, 29th International Symposium on Distributed Computing*, volume 9363, pages 617–630. Springer, 2015.

[8] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. Persistent Memory: A Survey of Programming Support and Implementations. In *ACM Computing Surveys Vol. 54, No. 7*, pages 1–37, July 2021.

[9] Alexandro Baldassin, Rafael Murari, João P.L. de Carvalho, Guido Araujo, Daniel Castro, João Barreto, and Paolo Romano. Nv-phtm: An efficient phase-based transactional system for non-volatile memory. volume 12247 LNCS, 2020.

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95*, pages 1–10, June 1995.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 49–60, USA, 2013. USENIX Association.

[12] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. SPHT: Scalable persistent hardware transactions. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 155–169. USENIX Association, February 2021.

[13] Daniel Castro, Paolo Romano, and João Barreto. Hardware Transactional Memory Meets Memory Persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 368–377, 2018.

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

[15] Daniel Castro, Paolo Romano, Diego Didona, and Willy Zwaenepoel. An analytical model of hardware transactional memory. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 221–231, 2017.

[16] Andrea Cerone and Alexey Gotsman. Analysing Snapshot Isolation. In *PODC '16: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 55–64, July 2016.

[17] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *SIGMOD Rec.*, 39(3):5–10, February 2011.

[18] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. Asf: Amd64 extension for lock-free data structures and transactional memory. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 39–50, 2010.

[19] 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. *SIGARCH Comput. Archit. News*, 39(1):105–118, mar 2011.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[21] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 251–264, USA, 2012. USENIX Association.

[22] Intel Corporation. From intel® optane™ persistent memory to cxl. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory-to-cxl-attached-memory.html, 2022. Accessed: 15 May, 2025.

[23] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: an extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013.

[24] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. In *ACM Transactions on Database Systems, Vol. 30, No. 2*, pages 492–528, June 2015.

[25] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 237–246, New York, NY, USA, 2008. Association for Computing Machinery.

[26] Pascal Felber, Paolo Romano, Alexander Matveev, and Shady Issa. Hardware Read-Write Lock Elision. In *EuroSys '16: Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, April 2016.

[27] Ricardo Filipe, Shady Issa, João Barreto, and Paolo Romano. Stretching the capacity of Hardware Transactional Memory in IBM POWER architectures. In *PPoPP '19: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 107—-119, February 2019.

[28] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. Cxl memory as persistent memory for disaggregated hpc: A practical approach. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 983–994, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: efficient, HTM-compatible persistent transactions. In *PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–74, June 2020.

[30] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous Checkpointing of HTM Transactions in NVM. *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management – ISMM 2017*, pages 70–81, 2017.

[31] Ellis Giles, Kshitij Doshi, and Peter Varman. Hardware transactional persistent memory. In *Proceedings of the International Symposium on Memory Systems*, page 190–205, 2018.

[32] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 913–928, Renton, WA, July 2019. USENIX Association.

[33] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 175–184, 2008.

[34] IBM. POWER9 Processor User's Manual (Version 2.0), April 2018.

[35] IBM. Transactional memory. https://www.ibm.com/docs/en/xffbg/121.141?topic=fortran-transactional-memory, Last accessed 14 Jan 2022.

[36] Shady Issa, Pascal Felber, Alexander Matveev, and Paolo Romano. Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[37] Shady Issa, Paolo Romano, and Tiago Lopes. Speculative read write locks. In *Proceedings of the 19th International Middleware Conference*, page 214–226, 2018.

[38] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. DHTM: Durable Hardware Transactional Memory. In *ISCA'18*, pages 452–465, June 2018.

[39] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, 2017.

[40] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving in-memory database index performance with intel® transactional synchronization extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 476–487, 2014.

[41] The Linux kernel development community. Transactional memory support. https://www.kernel.org/doc/html/v5.12/powerpc/transactional_memory.html, Last accessed 15 May, 2025.

[42] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-Performance Transactions for Persistent Memories. *Proceedings of the twenty first international conference on Architectural support for programming languages and operating systems – ASPLOS'16*, pages 399–411, 2016.

[43] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with TimeStone. In *ASPLOS'20*, pages 335–349, March 2020.

[44] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, January 2015.

[45] Shaobo Li, Yirui (Eric) Zhou, Hao Ren, and Jian Huang. Bytefs: System support for (cxl-based) memory-semantic solid-state drives. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 116–132, New York, NY, USA, 2025. Association for Computing Machinery.

[46] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1203–1217, New York, NY, USA, 2025. Association for Computing Machinery.

[47] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *ASPLOS '17: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329––343, April 2017.

[48] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 135–150, USA, 2016. USENIX Association.

[49] Paul Mackerras. KVM: PPC: Book3S HV: Work around transactional memory bugs in POWER9. https://github.com/torvalds/linux/commit/4bb3c7a0208fc13ca70598efd109901a7cd45ae7, Last accessed 15 May, 2024.

[50] Virendra Marathe, Michael Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William Scherer, and Michael Scott. Lowering the overhead of nonblocking software transactional memory. *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[51] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[52] Paul E. McKenney. A critical rcu safety property is... ease of use! In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 132–143, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Chris Mellor. Numemory reinvents optane storage-class memory. *Blocks and Files*, 7 Oct, 2024.

[54] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.

[55] Hasso Plattner. The impact of columnar in-memory databases on enterprise systems: implications of eliminating transaction-maintained aggregates. *Proc. VLDB Endow.*, 7(13):1722–1729, August 2014.

[56] Andy Rudoff. Persistent Memory Programming. *;login:*, 42(2), 2017.

[57] Samsung. Samsung cxl solutions – cmm-h. https://semiconductor.samsung.com/news-events/tech-blog/samsung-cxl-solutions-cmm-h/, Last accessed 15 May, 2025.

[58] Eleni Vafeiadi Bila, Simon Doherty, Brijesh Dongol, John Derrick, Gerhard Schellhorn, and Heike Wehrheim. *Defining and Verifying Durable Opacity: Correctness for Persistent Software Transactional Memory*, pages 39–58. Springer, 06 2020.

[59] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS XVI: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 91–104, March 2011.

[60] Junchang Wang and Manos Athanassoulis. Cubit: Concurrent updatable bitmap indexing. *Proc. VLDB Endow.*, 18(2):399–412, October 2024.

[61] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. Persistent Transactional Memory. *IEEE Comput. Archit. Lett.*, 14(1):58–61, January 2015.

[62] Xueliang Wei, Dan Feng, Wei Tong, Jingning Liu, and Liuqing Ye. MorLog: Morphable hardware logging for atomic persistence in non-volatile main memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 610–623, 2020.

[63] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. ArchTM: Architecture-Aware, high performance transaction for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 141–153. USENIX Association, February 2021.

[64] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: Log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 346—-359, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Yan Sha, Xiaofei Liao, Hai Jin, and Yan Solihin. Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 762–777, New York, NY, USA, 2023. Association for Computing Machinery.

[67] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. Polaris: Enabling transaction priority in optimistic concurrency control. *Proc. ACM Manag. Data*, 1(1), May 2023.

[68] Chongnan Ye, Meng Chen, Qisheng Jiang, and Chundong Wang. Hercules: Enabling atomic durability for persistent memory with transient persistence domain. *ACM Trans. Embed. Comput. Syst.*, 23(6), September 2024.

[69] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, November 2013.