Unbounded Hardware Transactional Memory for a Hybrid DRAM/NVM Memory System

Jungi Jeong*§, Jaewan Hong†, Seungryoul Maeng†, Changhee Jung*, Youngjin Kwon†
*Department of Computer Science, Purdue University
{jungijeong,chjung}@purdue.edu
†School of Computing, KAIST
{jaewanhong,maeng,yjkwon}@kaist.ac.kr

***Abstract*—Persistent memory programming requires failure atomicity. To achieve this in an efficient manner, recent proposals use hardware-based logging for atomic-durable updates and hardware transactional memory (HTM) for isolation. Although the unbounded HTMs are promising for both performance and programmability reasons, none of the previous studies satisfies the practical requirements. They either require unrealistic hardware overheads or do not allow transactions to exceed on-chip cache boundaries. Furthermore, it has never been possible to use both DRAM and NVM in HTM, though it is becoming a popular persistency model.**

To this end, this study proposes UHTM, unbounded hardware transactional memory for DRAM and NVM hybrid memory systems. UHTM combines the cache coherence protocol and address-signatures to detect conflicts in the entire memory space. This approach improves concurrency by significantly reducing the false-positive rates of previous studies. More importantly, UHTM allows both DRAM and NVM data to interact with each other in transactions without compromising the consistency guarantee. This is rendered possible by UHTM's hybrid version management that provides an undo-based log for DRAM and a redo-based log for NVM. The experimental results show that UHTM outperforms the state-of-the-art durable HTM, which is LLC-bounded, by 56% on average and up to 818%.

I. Introduction

Emerging Non-Volatile Memory (NVM) has gained a massive amount of attention from both researchers and practitioners for the last decade [15], [16], [37], [44], [58], [59], [67]. They have found killer applications, e.g., persistent key-value stores [23], [41], [63] and persistent database systems [33], [34], [52], to exploit both DRAM-like performance and durability as in SSD.

However, the benefits of NVM do not come for free. First, in-memory states of NVM must remain consistent across software crashes and machine failures, which makes *failure-atomicity* of NVM updates a hard requirement. Due to the 8-byte granularity of atomic stores in modern CPUs, programmers should deal with all the complexities to support the failure-atomicity of large stores ($>$ 8 bytes) and consistency between multiple objects in the presence of power failure. Second, in-memory states must remain consistent even when they are concurrently updated. Thus, the existing failure-atomic systems mediate concurrent updates on the shared data using synchronization primitives such as locks [13], [22], [36], [40] or transactional memory [39], [59].

Given these requirements, the ACID transaction guarantees what persistent programming requires. With that in mind, researchers have proposed `durable transactions` that support both atomic-durable updates and isolation in software [13], [14], [22], [36], [39], [59]. Since these proposals incur significant performance overheads, others take advantage of hardware support to accelerate a durable transaction. For example, they equip atomic-durable updates with hardware-based logging [17], [28], [31], [46], [53], [66] and offer isolation using hardware transactional memory (HTM) [3], [10], [30], [61].

However, the hardware support limits the total size of transactions, raising practical concerns. When a transaction goes over the hardware limitation (i.e., `capacity overflows`), it aborts and restarts, which significantly degrades the performance and the forward progress due to repeated overflows. In reality, this happens for NVM's killer applications whose transaction footprints scale from a few hundred KBs to MBs[1]. Although the capacity overflows can be addressed by *serializing* a problematic transaction using the programmer-defined slow path [26], programmers still have to reason the footprint of the transaction to minimize the serialization. The dynamic nature of data growth makes the reasoning a hard problem. In particular, it becomes even more challenging when applications run on virtualized or containerized environments—because several outstanding transactions compete for hardware resources. Given that unbounded HTMs can address the capacity overflow problem, it would be worth adapting them to persistent programming.

Unfortunately, merely extending the state-of-the-art unbounded HTMs is not sufficient to support full-fledged persistent programming in practice. First, previous unbounded HTMs either rely on unrealistic assumptions or suffer from an unacceptably large amount of false conflicts. For example, while LogTM assumes the fully-mapped directory coherence protocol [42], this assumption no longer holds for the hybrid DRAM/NVM memory system that can scale to Peta-bytes. LTM and VTM realize the unbounded HTM systems by

§This work is mostly done when the author was a Ph.D. student at KAIST.

[1]*Long-running* and *read-only* transactions, which are common in real-world applications [19], [48] and research papers [18], [38], [57], [62], exacerbate the capacity overflow.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00051

virtualizing cache eviction and managing overflow states in DRAM [1], [50]. However, they require searching DRAM to detect conflicts beyond the on-chip cache capacity, thereby causing prohibitively large overheads. On the other hand, the address signature used by Bulk and LogTM-SE ends up with a very high false-positive rate since they check all coherence traffics [12], [64]. According to our evaluation, more than 99% of transactions experience a repetitive false conflict and make no forward progress without serializing them.

More importantly, no prior study supports the interaction between non-persistent and persistent objects within a transaction. Their interplay is driven by two different trends. First, the ease of programming and its flexibility leads to the advent of new persistency models [20], [56] where DRAM and NVM objects are seamlessly managed in a transaction. For example, AutoPersist [56] can navigate non-persistent data structures and persist them on the fly. Similarly, Go-pmem [20] allows pointers across persistent objects and non-persistent ones inside a transaction. Second, the demand for performant transactional applications has them manipulate both persistent and non-persistent objects as well [5], [23], [34], [41], [63]. Since pointers between DRAM and NVM exhibit non-trivial inter-dependencies in the applications, their HTM acceleration must deal with the dependencies to provide consistency between persistent and non-persistent objects. The takeaway is that when a transaction aborts, intermediate changes made to both persistent and non-persistent objects must be discarded consistently. However, prior studies [3], [10], [30], [61] focus only on persistent objects, leaving questions on how to offer consistency for the new persistency models unanswered.

To overcome the aforementioned challenges, we present UHTM, our hardware support for **U**nboundedness in the DRAM/NVM **H**ybrid **T**ransactional **M**emory. To efficiently isolate transactions, UHTM introduces `staged conflict detection` that combines two detection schemes and selectively uses one depending on the transactional object location being changed. For objects in on-chip caches, UHTM leverages the *cache-coherence protocol* that renders conflict detection accurate but is limited within on-chip caches. On the other hand, when transactional objects are evicted to off-chip memory, UHTM uses *address signatures* based on hardware bloom filters that can cover the unlimited address space. The staged detection scheme significantly reduces the abort rates of durable transactions compared to using the address signature only since the cache-coherence protocol filters the traffic to signatures, resulting in fewer false positives.

Moreover, UHTM further reduces the abort rate in virtualized or containerized environments. Even if two persistent applications do not share data, LLC miss requests of one can be found in the address signature of another (i.e., a false positive), which causes a transaction unnecessarily to abort. UHTM prevents this false abortion between different persistent applications by confining the conflict domain. LLC miss requests are only checked against the signatures in the same conflict domain. Isolating transactions per conflict domain achieves an additional reduction on the false positive rate, making UHTM practical in consolidated environments.

To minimize the commit latency, which is common and performance-critical, UHTM proposes `hybrid logging` that brings two different loggings into the design. It performs *undo logging* for non-persistent objects (in DRAM) while *redo logging* for persistent objects (in NVM). Employing the undo logging for DRAM data achieves two advantages. First, it commits faster since the redo logging suffers from copying overheads of new values in the log to commit transactions. Second, the undo logging does not require searching the logs to locate new values while redo logging must do. On the other hand, for NVM data, UHTM adopts a recent study of an efficient hardware failure-atomic update based on redo logging [28], which minimizes the latency and the bandwidth consumption of failure-atomic updates.

The implication of transactions having both DRAM and NVM data is to guarantee consistency between non-persistent and persistent data structures. UHTM commits them in parallel and atomically. On aborts, UHTM reverts intermediate changes on DRAM and NVM using undo and redo logs, respectively, preserving consistency between DRAM and NVM data.

The contributions of this paper are as follow:

- We design UHTM, the unbounded HTM for a hybrid DRAM and NVM memory system. This is the first proposal that achieves the unboundedness and the support for the DRAM/NVM hybrid memory in HTMs.
- For the first time, UHTM allows both DRAM and NVM data in a transaction and ensures their consistency, taking into account a common practice in recent persistency programming models.
- UHTM's novel conflict detection scheme reduces the abort rate of durable transactions from 99% to 9% by removing most of false positives of address signatures.
- Our evaluation shows that UHTM outperforms the state-of-the-art durable hardware transactional memory [30] by 56% on average and up to 818%.

The rest of the paper is organized as follows. We compare our design to previous studies in Section II. Then, we motivate the urgent need for unbounded HTMs for the DRAM and NVM hybrid memory system in Section III. Our design goals and implementation are explained in Section IV. We evaluate our design in Section V and Section VI. Finally, we conclude our paper in Section VII.

II. Related Work

HTM must provide version management and conflict detection. *Version management* keeps both unmodified and modified data within a transaction and uses a modified data (e.g., new value) when committing and an unmodified version (e.g., old value) when aborting. On the other hand, *conflict detection* identifies concurrent accesses on the shared data and performs proper means to serialize them if necessary. For HTMs to be unbounded, they must support the unboundedness in both version management and conflict detection.

In this section, we introduce previous solutions for unbounded and durable transactions. Our discussion here focuses

	Transaction Boundary		Conflict Detection		Version Management	
	DRAM	NVM	On-chip	Off-chip	DRAM	NVM
LogTM [42]	Unbounded	Not support	Cache-coherence	Sticky-Bit in Directory	Undo	None
LTM [1]&VTM [50]	Unbounded	Not support	Cache-coherence	Overflow States in DRAM	Undo	None
LogTM-SE [64]	Unbounded	Not support	Signature in L1-cache	Signature in L1-cache	Undo	None
Bulk [12]	Unbounded	Not support	Signature in L1-cache	Signature in L1-cache	Redo	None
PTM [61]	Not support	L1-cache	Cache-coherence (L1-bounded)	None	None	Undo
PHyTM [3]	Not support	L1-cache	Cache-coherence (L1-bounded)	None	None	Redo
NV-HTM [10]	Not support	L1-cache	Cache-coherence (L1-bounded)	None	None	Redo
DHTM [30]	Not support	LLC	Cache-coherence	None	None	Redo
UHTM	Unbounded	Unbounded	Cache-coherence	Signature in DRAM/NVM (+Signature Isolation)	Undo (Overflow only)	Redo

TABLE I: Comparison of UHTM with previous studies.

on conflict detection because it hinders previous solutions not suitable for NVM. We discuss version management in Section III-A. Table I compares previous approaches and summarizes how they differ.

A. *Limitations of Prior Unbounded HTMs*

Although there have been many proposals for unbounded HTMs, they do not suffice to satisfy the requirements of durable transactions. These studies aimed to unshackle the boundary of hardware transactional memory only with concerning DRAM. Consequently, they do not fit for NVM. In fact, their conflict detection methods have several limitations in applying to large NVM.

First, they require non-negligible modifications to existing hardware and software. For example, LogTM extends a directory-based cache coherence protocol with a sticky-bit to detect conflicts [42]. That is, LogTM sets a sticky-bit in the directory entry on the cache eviction and determines access as a conflict if it requests the block with the sticky-bit. However, this approach relies on the fully-mapped directory protocol, which is unrealistic under the hybrid DRAM/NVM memory system that can scale to Pete-bytes. Although both LTM and VTM take a different approach that realizes unbounded transactions by virtualizing cache evictions and moving them to software tables [1], [50], the both suffer from significantly long latency by scanning the software tables to detect conflicts. *Due to hardware/software complexities and overheads, these approaches are not practical for durable transactions.*

Moreover, a common technique for boundless detection is to use address signatures that represent the read- and write-sets of transactions [12], [64]. The address signatures are managed based on hardware bloom filters and can encode unlimited addresses, thus making transactions unbounded. Unfortunately, this approach ends up with a lot of false positives, leading to many unnecessary aborts. Given the nature of durable transactions whose data footprint easily exceeds a few hundreds of KBs[2], the signature bloom filters are quickly saturated generating many false positives. Thus, the address signature-based approach is vulnerable to false aborts. In our experiments, we observed that transactions could not make forward progress without serialization if address signatures are only used to detect conflicts (see Section VI). Therefore, *reducing the false positives of address signatures is critical to make unbounded HTMs efficient.*

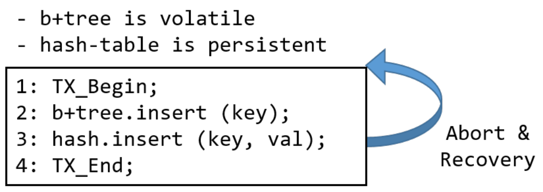
[2]We measured the footprints of transactions in SQLite using Mobibench [29]. The sizes of read- and write-sets are about 160KB and 100KB, respectively.

Finally, OneTM eases the management of capacity overflows by limiting a single overflow transaction at a time [7]. However, running a single overflow transaction will cause performance bottleneck as transaction sizes increase. On the other hand, TokenTM detects conflict using tokens allocated for every memory block [8]. Although TokenTM can support unbounded transactions, token management is costly in the hybrid DRAM/NVM memory system. Releasing tokens on commit or abort requires to walk the log in DRAM, whose cost is linear to transaction sizes.

B. *Limitations of Previous Durable HTMs*

With demanding requirements of persistent programming, there have been a variety of studies proposing durable transactions in software [14], [15], [36], [39], [59]. Unfortunately, the software-based approaches cause a significant performance degradation due to ordering stores in logging [53], [54] and identifying conflicts (e.g., validation) when committing [9].

To overcome the limitations of software-based approaches, researchers proposed the hardware support to extend HTM for durability support with hardware-based logging [17], [28], [31], [46], [53]. Since they use Intel RTM-like HTM, which relies on the cache-coherence protocol to detect conflicts within the L1 cache only, PTM [61], NV-HTM [10], and PHyTM [3] all restrict durable transactions so that they fit in the L1 cache. If a transaction footprint exceeds the L1 cache boundary, it either restarts or falls back to the slow-path provided by programmers—which eventually serializes transactions. Recently, DHTM extends the transactional boundary to the last-level cache to reduce the serialization due to frequent L1-overflow [30]. However, this is not a safeguard to prevent capacity overflow fundamentally, e.g., for *long-running* and *read-only* transactions in real-world applications [18], [19], [38], [48], [57], [62], DHTM inevitably generates capacity overflows. Consequently, *none of the previous studies, that propose durable HTM, supports unbounded transactions.*



```
- b+tree is volatile
- hash-table is persistent
1: TX_Begin;
2: b+tree.insert (key);
3: hash.insert (key, val);
4: TX_End;
```

Fig. 1: An example of a durable transaction that manipulates both DRAM and NVM data.

C. *UHTM's Approaches*

UHTM uniquely positions itself, as shown in the last row of Table I. For conflict detection, this paper proposes to use the staged detection technique that can minimize the false-positive rate of address signatures. That is, UHTM integrates the cache-coherence protocol for removing false positives in on-chip caches and the signature isolation technique for reducing false conflicts in off-chip memories. For version management, UHTM allows both DRAM and NVM data to be managed in a transaction to respond to a compelling need for such a seamless interplay (see Section III-A). The novelty of the UHTM design is two-fold: undo logging only for cache-evicted DRAM data and applying different logging for DRAM and NVM data.

III. Motivation

A. *Using DRAM in Durable Transactions*

Two trends lead to the advent of a new persistency model *where DRAM and NVM objects are managed in a transaction.* First, the need for ease of programming and the flexibility demand drive the removal of the restriction that forces to manage either non-persistent or persistent objects only [20], [56]. Recent persistent programming models such as AutoPersist [56] and Go-pmem [20] allow persistent and non-persistent data to seamlessly interact with each other in a transaction.

Second, the demand for high performance puts significant pressure on the data management, pushing transactions to manipulate non-persistent data structures therein to improve the performance [5], [23], [34], [41], [63]. For example, Figure 1 represents the code fragment of inserting a new key-value pair into a hybrid persistent kv-store [63][3] that maintains two indexes, b+tree and hash-table. The b+tree is placed in DRAM to accelerate a *scan* operation while others such as *put/get/update/delete* use the hash-table in NVM. Since these two indexes have to be synchronized, a transaction includes both indexes and updates them atomically. Furthermore, other proposals employ a hybrid memory design using DRAM to offset the performance of slow NVM [5], [23], [34]. If transactions do not allow manipulating both memory types, persistent programming becomes increasingly tricky.

The takeaway is that HTM acceleration of such persistent programming models must ensure consistency for both between persistent and non-persistent objects and handle their inter-dependencies. In this respect, an important implication is that not only NVM stores but *DRAM stores also have to be logged* since transaction aborts are more common than power failures. In other words, commit and abort protocols must preserve the consistency of both memory types. Similar to persistent data, where failure-atomic updates guarantee its atomicity, volatile data also needs to be handled atomically.

[3]We replaced mutex-based critical sections to transactions.

Unfortunately, supporting both volatile and persistent memory requires a significant redesign of existing systems. DHTM, the start-of-the-art durable HTM proposal, leaves this challenge unanswered and handles persistent data only within transactions [30]. Other durable HTM solutions have the same limitation as well [3], [10], [61]. On the other hand, prior unbounded HTMs for volatile data are vulnerable to a high abort rate even if they are modified to support failure-atomicity. Given all this, we design UHTM to address the unbounded hardware transactional memory for both DRAM and NVM data. To the best of our knowledge, no prior study supports their interaction within a transaction.

B. *Handling Transaction Overflows*

Commercial HTMs do not guarantee forward progress [26], [47], [60]. In addition to its concurrency control mechanism contributing to this, the `capacity overflow` makes forward progress more difficult by aborting transactions that grow beyond the hardware limitation. Previous works rely on the cache coherence to detect conflicts and, hence, limit the transaction boundary within the on-chip cache size (e.g., L1-cache [10], [61] and LLC [30]). If footprints of transactions exceed the boundary, transactions abort since HTM can no longer support correctness. According to our experiments, capacity overflows slow down applications by up to 6.2x (See Section III-C). This limitation hinders a wide adoption of HTM to existing applications [11], [32].

For handling capacity overflows, a common practice requires a programmer to provide the slow-path that *serializes* transactions. Algorithm 1 shows the example code of handling transactions for Intel RTM [26]. This practice applies similarly to other commercial HTMs as well [47], [60]. When the transaction starts (line 3) and the lock is available (line 4), transactions execute concurrently in a `fast-path`, indicated by ❶. Otherwise, the transaction aborts the execution (line 8) and waits until the lock becomes available (line 12). If capacity overflows are detected (line 15), the transaction jumps to the `slow-path` (indicated by ❷) without retrying because capacity overflows tend to happen repeatedly even after restarts. Hence, the slow-path *serializes* transactions to avoid wasting more cycles for retrying. In the absence of capacity overflows, the execution also falls back to the slow-path after the maximum number of retries (line 18).

This practice significantly degrades the concurrency as well as forces programmers to reason about hardware limitations and optimize transactions in order to minimize the overflows. This paper tackles this problem by proposing the `unbounded` hardware transactional memory.

Algorithm 1 Hardware Transactional Memory Programming.

```
function $run\_transaction$
    while 1 do
        if $\_xbegin() = \_XBEGIN\_STARTED$ then
            if $!lock.isLocked()$ then
                fast-path                                  ▷ ❶
                $\_xend(); return$
            end if
            $\_xabort()$
        end if
        if $\_XABORT\_EXPLICIT$ then
            while $lock.isLocked()$ do
                $pause()$
            end while
        end if
        if $\_XABORT\_CAPACITY$ then
            $break$
        end if
        if $N\_retries \geq Max\_retries$ then
            $break$
        end if
    end while
    $lock.acquire()$
    slow-path                                              ▷ ❷
    $lock.release()$
end function
```

C. Why Overflows Matter?

Previous work extends the transaction boundary beyond L1 cache up to the last-level cache because the sizes of transactions in modern applications exceed the L1 cache capacity [11], [30], [32]. However, LLC is no longer the safety line to avoid such costly capacity overflows. First, the software and hardware trends encourage the consolidation of workloads, bearing heavy contention in LLC. Given that cloud infrastructures already have adopted persistent memory [21], the rapid spread of lightweight container platforms is to increase the consolidation level to overcome heavyweight virtualization. All these trends imply that the number of applications in the system and thereby the contention on the LLC will increase, which renders LLC overflows the real problem. We found that even a single memory-intensive application (e.g., graph500) could consume all of the shared LLC, preventing other processes from occupying LLC during 52% of its execution time. Therefore, if such applications run with failure-atomic transactions, *the overflow from the private caches leads to overflow in LLC in a cascade.*

Second, even if most transactions are small size, there are large transactions that exceed the on-chip cache capacity: long-running, read-only transactions are often used in both real-world applications and research proposals [19], [35], [38], [48], [57], [62]. Although they are not frequent (e.g., less than 1% of the total transactions), the resulting throughput is substantially decreased—because they are prone to capacity overflows. When the capacity overflow occurs, not only the long-running, read-only transaction but other transactions—that are short and prevalent—also should be blocked.

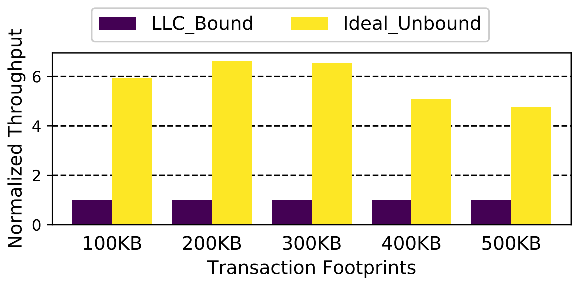


Fig. 2: Comparing throughput of `LLC-Bounded` to `Ideal` unbounded HTM with running 16 threads.

Furthermore, capacity overflows are hard to prevent. Although programmers design transactions to fit in the hardware limitation, transactions still may be overflowed due to contentions in the shared resources [27], [30], [32], [60]. Also, once the capacity overflow occurs, a transaction would repeatedly encounter the same capacity overflow even though it restarts—since the contention in the shared resources may still exist. Hence, it is an urgent need to unshackle the size limitation of hardware transactional memory to eliminate the performance slowdown due to the overflows.

To quantify the impact of capacity overflows, we compared the throughput of two HTM systems. The `LLC-bounded` HTM system extends the cache coherence protocol to detect conflicts within the last-level cache, similar to the recent study, i.e., DHTM [30]. For the `unbounded` HTM, we assume that it is capable of `perfect` conflict detection for unbounded memory spaces (having no false positive). Section V provides detailed configurations. As shown in Figure 2, we observed that the LLC-bounded HTM is up to 6.2x slower than the ideal unbounded HTM.

D. Need for Unboundedness

In summary, the advantages of supporting the unboundedness in HTM is two-fold; (1) high performance and (2) convenient programming. First, transactions no longer experience capacity overflows and, hence, do not execute the slow, serialized path used to guarantee forward progress in the current best-efforts HTM, achieving a significant performance boost as elaborated in Section III-C. Second, the unboundedness allows programmers to have less burden in designing transaction boundaries by hiding hardware limitations. This guarantee not only makes the programming simple but reduces the development cost. For example, the best-effort, bounded HTM even requires redesigning index structures to accommodate HTM into database systems [32]. Thus, the unbounded HTM remedies the complexity and, as a consequence, catalyzes the adoption of HTM into the wide range of existing applications.

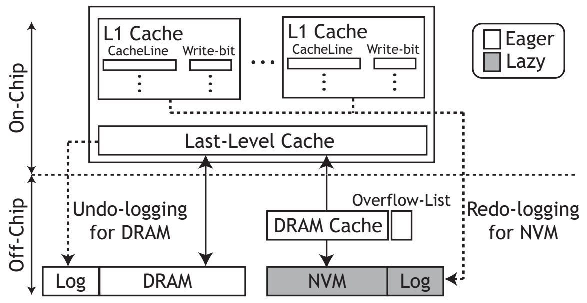


Fig. 3: The hybrid version management scheme of UHTM.

IV. UHTM

A. Overview

To this end, we propose **UHTM** that provides unbounded hardware transactions for hybrid DRAM and NVM memory system. UHTM satisfies all ACID properties. The design of UHTM is summarized as follows:

- For **atomic-durability**, UHTM proposes the hybrid hardware logging that minimizes the commit latency. For non-persistent data, UHTM uses undo logging while performing the redo logging for persistent data.
- To preserve non-persistent and persistent data of the same transaction **consistent**, UHTM performs different commit and abort protocols for non-persistent or persistent data since it offers different logging techniques.
- For **isolation**, UHTM leverages the cache coherence protocol to detect conflicts within the on-chip cache and relies on address signatures to detect conflicts beyond on-chip caches. By placing the address signatures on the memory bus and checking conflicts only for LLC-overflowed blocks, UHTM substantially reduces the false-positive rate.

B. Atomic-Durability

In this section, we explain how UHTM guarantees atomicity and durability. Atomicity applies for both DRAM and NVM data and requires to make modifications visible at once when commits. UHTM supports atomicity via logging for both DRAM and NVM data. On the other hand, durability only correlates with NVM data and requires to flush all modifications to NVM before a transaction commits. Based on this observation, UHTM proposes the `hybrid` scheme that uses the `eager` version management for data in on-chip caches and DRAM while applying the `lazy` policy for NVM data. Figure 3 illustrates the hybrid version management scheme. *All* NVM updates within transactions are logged with new value and flushed to NVM for durability. On the other hand, for DRAM data, logging is only used for *overflowed* blocks. We explain the rationale behind this design decision and how to handle data in on-chip caches.

UHTM reserves the part of the DRAM and NVM regions for the log area. The log area is only accessible to the memory controllers upon committing or aborting transactions. The memory controllers serialize and append concurrent log writes to the end of the log area.

On-chip Caches. UHTM allows the changes to overwrite data in on-chip caches (known as the `eager` version management) and does not store its copies in speculative buffers, such as logs. Instead, if a transaction attempts to overwrite dirty cache blocks that belong to DRAM, UHTM write-backs them to DRAM before overwriting. UHTM invalidates modified cache blocks when the transaction aborts (See Section IV-C). On a transaction commit, UHTM makes transactional DRAM data visible in their coherence states without flushing them to DRAM. In this way, the abort and commit in UHTM are fast for volatile transactions.

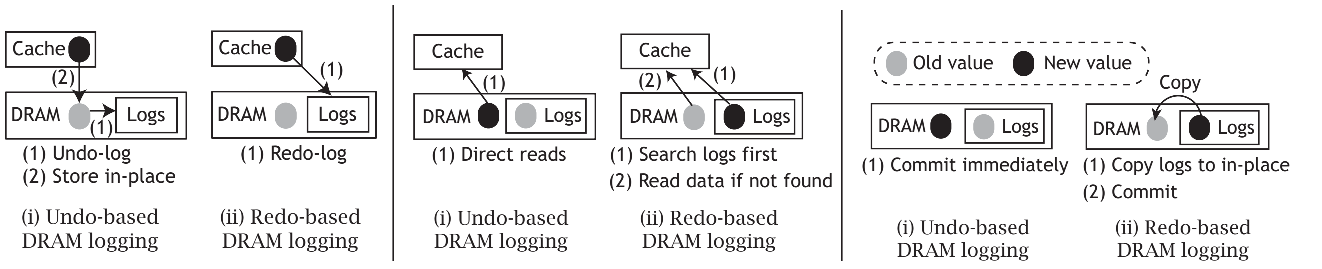
DRAM Data. On the other hand, DRAM data evicted from on-chip caches require logging for correct recovery. Figure 4a shows two different logging strategies. Undo-based logging copies an original value in DRAM to a log, and an overflowed block overwrites in-place location (as the `eager` version management). In contrast, the redo approach keeps new values of overflowed blocks in the log while in-place data remain unmodified (as the `lazy` version management).

For this reason, UHTM employs `eager` version management. Whenever evicting DRAM data from LLC, the memory controller copies data in DRAM (e.g., old versions) to the log before handling the cache eviction. Since the cache eviction is not in the critical path, the undo logging can happen asynchronously without stalling the transaction.

The eager policy better suits for the LLC-evicted DRAM data for two reasons. First, it provides faster commits than lazy version management. As shown in Figure 4c, undo logging can finalize the commit protocol immediately by placing the commit mark on the log because all changes are already applied. In contrast, redo logging suffers from the copying of new values in logs, making them commit slow. Although eager version management has to recover overflowed blocks by using undo-logs when aborting the transaction, UHTM prioritizes commits, which is more common than aborts.

Second, the eager policy does not suffer from a read-indirection problem. Once the LLC-overflow happens, redo logging stores new values in logs while leaving in-place data unmodified. Therefore, to locate new values, DRAM read must query the log before accessing its address in DRAM (shown in the right figure of Figure 4b). Indexing the log area often necessitates multiple DRAM accesses, slowing down the DRAM read performance. On the other hand, undo logging is free from this limitation. As shown in the left figure of Figure 4b, DRAM read can get data in its address directly without log lookup.

NVM Data. Unlike DRAM data, durable transactions must write-back NVM data from caches for durability when they commit. UHTM uses the previous study that provides hardware-based logging for persistent data [28]. The prior work proposed a commit protocol based on redo-logging. It flushes modified cache blocks that belong to NVM to a DRAM cache, which is placed between LLC and NVM, while not deleting redo log entries. The DRAM cache buffers LLC-overflowed blocks



(a) DRAM logging when overflows. (b) DRAM reads after overflows. (c) Commits of DRAM logging.

Fig. 4: Comparison of undo and redo logging for DRAM data of LLC-overflowed blocks.

(referred to as early-evicted blocks in [28]) and replaces the NVM log searches with faster DRAM searches. Later, it updates in-place data through eviction from the DRAM cache without copying from the log. It provides fast commits and saving NVM bandwidth. Note that UHTM does not bound to the prior redo logging [28], but other designs for durable logging for persistent data can also be used [17], [30], [31], [46], [53].

As other failure-atomic updates do, UHTM also flushes modified cache blocks to NVM when it commits a transaction. To find a write-set in L1-cache, UHTM refers to a write-bit in L1-resident cache blocks. However, locating the write-set in the shared LLC and the DRAM cache is not trivial. Scanning the shared LLC and DRAM cache whenever committing a transaction increases the commit latency and would hamper other transactions to access them. To this end, UHTM uses the `overflow list` to store addresses of L1-evicted blocks to reduce the latency of locating the write-set in LLC and DRAM cache, similar to the previous study [30]. UHTM avoids scanning the entire LLC or the DRAM cache by referring to the overflow list when committing or aborting transactions. UHTM stores the overflow list in the DRAM cache.

UHTM starts a commit protocol to DRAM and NVM in parallel. First, the commit protocol for NVM data waits until all redo-logs become durable in the NVM logs. Then, UHTM accesses to the overflow list stored in the DRAM cache and performs the following actions depending on whether the address belongs to DRAM or NVM. Non-persistent DRAM data are marked visible in their coherence states while persistent NVM data are flushed to the DRAM cache. Meanwhile, the commit for LLC-overflowed DRAM data is done after writing the commit-mark on the DRAM log. When the commit protocols for DRAM and NVM are completed, UHTM finalizes the commit procedure.

C. Consistency and Data Recovery

This section explains how UHTM preserves consistency when aborting a transaction that manipulated both DRAM and NVM data due to a conflict or restarting one from a failure. *Transaction ID*, a monotonically increasing global counter, is stored in a register on each core and uniquely identifies a transaction, defining the boundary of preserving consistency.

UHTM restores the program state from a power failure with NVM data only. UHTM replays the committed redo entries in the NVM log area and disregards the uncommitted one, as same as the recovery of redo-logging in the conventional database logging. The programmers' responsibility is to place data structures in NVM if they are necessary for data recovery.

On the other hand, UHTM guarantees the consistency of both DRAM and NVM data in conflicted transactions. When a conflict is detected (Section IV-D describes conflict detection of UHTM), UHTM aborts the conflicted transaction by rolling back to the previous consistent state of DRAM and NVM. The abort protocol of UHTM consists of two stages: aborting on-chip states and off-chip states. Abort protocols for all stages begin in parallel.

On-chip states. On aborts, UHTM flushes all pipeline states of a core at first and invalidates all cache blocks modified by the aborting transaction. For cache blocks in the private cache, UHTM identifies cache blocks with a write-bit set and invalidates them while UHTM accesses the overflow list to locate modified blocks in LLC. Then, UHTM sends an invalidation request for blocks stored in the overflow list.

DRAM. Since UHTM keeps old versions in the log area, it needs to restore in-place data with old values in the log when aborting the transaction. Specifically, UHTM finds log entries of the aborted transaction in the log area and copies the content to in-place locations. Although the abort process is expensive in exchange for fast commits, UHTM optimizes commits instead of aborts.

NVM. Aborting persistent data in UHTM consists of two steps; the DRAM cache and redo logs in NVM. First, UHTM obtains addresses of uncommitted blocks in the DRAM cache by accessing the overflow list and invalidates them by setting an invalidate bit. Next, UHTM needs to delete logs of aborted transactions. However, searching logs involves scanning the NVM log area, making the abort procedure slow. Hence, to reduce the abort latency, UHTM defers log deletion to the background and completes the abort protocol after marking an abort flag. Later, the log reclaiming policy deletes logs in a similar way that the prior work has proposed [28].

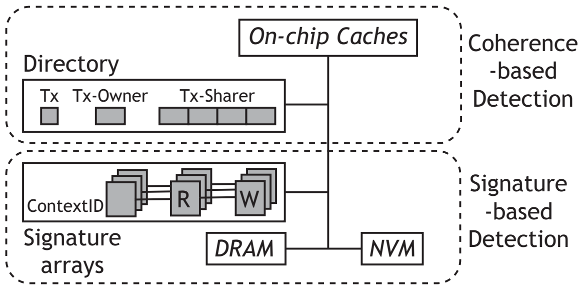


Fig. 5: The conflict detection mechanism of UHTM.

D. Isolation

UHTM guarantees serializability, where the concurrent and serial executions produce the same output. This guarantee is identical to what other hardware transactional memory systems guarantee both in commercial [26], [47], [60] and literature [3], [10], [30], [42], [61]. UHTM proposes the `staged` conflict detection mechanism that uses different methods in on-chip caches and off-chip memory. Figure 5 illustrates the conflict detection mechanism for UHTM. UHTM extends the directory-based cache coherence protocol to detect conflicts in on-chip caches, which is accurate, while it uses address signatures for overflowed blocks in off-chip memory, which provide an unlimited range. UHTM uses the `eager` conflict detection policy for both on- and off-chip memory but differs in how to resolve conflicts whether detected in on-chip caches or off-chip memory. We explain the conflict resolution policy in Section IV-E.

On-chip caches. UHTM extends the directory-based cache coherence protocol without changing coherence states. UHTM introduces new fields in the directory entry; `Tx`-bit, `Tx-Owner`, and `Tx-Sharer`. When a core has read or written blocks in the transaction context, UHTM sets the `Tx`-bit of the directory entry. The `Tx-Owner` and `Tx-Sharer` fields identify the owner transaction that modifies the block and sharers that read the block, respectively. These fields store the transaction IDs, instead of core IDs to handle a context switch (see Section IV-E). UHTM clears the Tx-Set, Tx-Owner, and Tx-Sharer in the directory entry and the address signatures when transactions commit or abort.

When the directory receives the coherence requests that demand blocks with the `Tx`-bit set, UHTM starts checking conflict conditions such as read-after-write, write-after-write, or write-after-read. For example, if it receives an exclusive request (e.g., GetM), UHTM determines it as either a read-after-write conflict if the block has a sharer in `Tx-Sharer` or a write-after-write conflict if its `Tx-Owner` exists. When a shared request (e.g., GetS) arrives, and its `Tx-Owner` set, then it is a write-after-read conflict. The conflict detection mechanism of UHTM is similar to the one of LogTM [42] but is different in the case of cache overflow. UHTM uses the address signatures, explained below, for cache overflow instead of sticky states in LogTM.

	Overflowed?	Action
On-chip Cache	One	Abort non-overflowed Tx
	None or both	Requester-Wins
Off-chip Memory	One	Abort non-overflowed Tx
	None or both	Requester-Aborts

TABLE II: The conflict resolution policy of UHTM.

Overflowed blocks. The cache coherence is not sufficient to detect conflicts on overflowed blocks. The naive approach of identifying conflicts on overflowed blocks is to walk the log area and match an address in the log. Since this approach is infeasible due to long latency, UHTM needs to detect conflicts on overflowed blocks without accessing memory.

For this purpose, UHTM uses the address signatures proposed in previous studies [12], [30], [55], [64]. The signatures are implemented in the HW bloom filter and encode addresses of overflowed blocks. Each transaction has separate read- and write-signature. They can determine conflicts in unlimited space but may produce a false-positive. Previous studies place the signature in L1-cache and check all incoming coherence messages [10], [12], [55], [61], [64] or L1-missed requests [30]. Unfortunately, this design results in a very high false-positive rate and aborts non-conflicting transactions. We observed that abort rates increase beyond 99% even if using large signatures (e.g., 16k-bit). Most aborts were due to false-positives. On the other hand, UHTM checks signatures with the address of the LLC-missed request only. Therefore, UHTM drastically reduces the false-positive rates from 99% to 26%.

Optimization. Using address signatures for cache-overflowed blocks arises a unique challenge that aborts transactions if they conflict with a *non-transactional* context. Signature arrays are vulnerable to such false-conflicts since all LLC-missed requests must be checked to provide correctness. For example, background processes, which are *completely unrelated* to durable transactions, could abort the transaction if signature arrays detect it as a conflict (but it is false-positive). We observed that such false-conflicts increased the abort rate further by 17%. To reduce the false-positive rates, UHTM proposes the signature isolation technique that confines address signatures by defining the conflict domain. The conflict domain denotes a group of transactions that share the address space and, therefore, potentially conflict with each other. We modified the pthread library to generate a transaction group ID shared by threads in the process. The signature isolation technique does not require modifications to applications. By grouping address signatures with the group ID and eliminating false conflicts between processes, UHTM further decreases the false-positive rates from 26% to 9%.

E. Discussion

Conflict detection and resolution. UHTM enforces the pessimistic (`eager`) detection strategy. In the conflicting cases, one of the transactions has to abort and restart with a random backoff delay to avoid subsequent aborts. If only one transaction overflowed between two conflicted transactions, UHTM aborts the non-overflowed transaction to prioritize the overflowed one.

Table II summarizes the conflict resolution policy of UHTM. UHTM maintains the transaction status structure (TSS) to track the status of all running transactions, whose entry consists of the transaction ID, abortion flag, and the overflow bit. If UHTM detects a conflict, it marks the abortion flag of the aborted transaction in the TSS and signals to the thread to abort the execution. In the case of capacity overflow, UHTM sets its overflow bit in the TSS. If two overflowed (or non-overflowed) transactions encountered a conflict with each other, UHTM uses the different strategies whether the conflict occurs in on-chip caches or off-chip memory. For conflicts found within on-chip caches, UHTM uses the requester-wins policy [2], [26], which prioritizes the transaction that requests later. On the other hand, for conflicts of overflowed blocks, UHTM aborts the transaction that requests later by the `requester-loses` policy because the policy does not require extra communication between processors. The requester-wins/-loses policy may produce a cyclic abortion [2], [4], [51], [65], leading to livelock. UHTM leaves this problem to the future work.

Context switch. UHTM preserves contexts of transactions for conflict detection and version management across context switches by virtualizing the transaction ID. First, the directory and address signatures use the transaction ID instead of the CPU ID to locate the correct thread running the transaction after context switches. For example, when a transaction aborts, UHTM broadcasts the aborted transaction ID to all CPUs and aborts the transaction if it matches its transaction ID of a received CPU. Furthermore, UHTM offers conflict detection during context switches as well.

If the aborted transaction belongs to a suspended thread, the broadcast will not receive its acknowledge. In this case, UHTM sets the abortion flag in the TSS of the aborted transaction. Then, it proceeds an abort protocol by invalidating cache blocks in the write-set of the aborted transaction in LLC and requesting an abort message to DRAM and NVM with its transaction ID. When the suspended thread resumes, it restarts by checking the abortion flag in the TSS.

Moreover, a transaction must find the write-set in the private cache of the CPU, which it runs on, whenever it either commits or aborts. Therefore, UHTM flushes modified data of both DRAM and NVM in the private cache to the LLC on context switch. The latency of flushing the private cache can be reduced by the hardware support [49]. Later, UHTM correctly locates these blocks in the LLC without asking the other CPUs and flushes to NVM.

Hardware overheads. UHTM adds the hardware overheads on top of Intel RTM-like HTM and includes hardware overheads of the previous hardware-logging design [28], such as a DRAM cache. Also, UHTM adds registers for the transaction ID, the transaction status structure (TSS), and the start address of the overflow list to each core. Moreover, UHTM extends the directory for cache-coherence with additional fields on each entry; Tx-Set, Tx-Owner, and Tx-Sharer. It also requires separate read-/write-address signatures for each core. The log areas of DRAM and NVM are reserved during system initialization and not accessible by software. If the log is out of free space, UHTM traps the operating system to expand the log area, similar to previous studies [28], [31].

Processor	16-core, 2GHz, in-order
L1 I/D Cache	Private 32KB, 8-way
L1 Latency	1.5ns
L2 Cache	Shared 16MB, 16-way
L2 Latency	15ns
DRAM Latency	Read/Write = 82ns
NVM Latency	Read = 175ns, Write = 94ns

TABLE III: Simulation configuration.

Benchmarks	Description
HashMap [25]	Insert/update entries in hash table
B-Tree [25]	Insert/update nodes in b-tree
RB-Tree [25]	Insert/update nodes in red-black tree
SkipList [25]	Insert/update entries in skip-list
Hybrid-Index KV-Store [63]	Insert/update in KV-store with two indexes in DRAM and in NVM
Dual KV-Store [23]	Insert/update in KV-store with two data structures in DRAM and NVM
Echo [5]	Insert/update KV-pairs to persistent hash table

TABLE IV: List of benchmarks used in our evaluation.

V. Methodology

Our evaluation uses the system-call emulation mode of the gem5 simulator [6]. Table III summarizes the simulation configuration. We configured our system to have a 16-core multicore and two-level cache hierarchy. Each core has a 32KB private L1 I-cache and D-cache and shares 16MB of the last-level cache. We measured latency from the real persistent memory module [24] and set 175ns and 94ns for read and write latency of NVM, respectively. The write latency is faster than read latency since the NVM write finishes when the memory controller accepts the request in the write-pending queue. The durability of pending write requests in the memory controller is guaranteed by asynchronous DRAM refresh (ADR).

Table IV presents the benchmarks used in our evaluation. We use micro-benchmarks such as `HashMap`, `B-Tree`, `RB-Tree`, and `SkipList` data structures provided in the Intel PMDK library [25]. We evaluated these benchmarks with persistent and volatile versions. The volatile versions keep all data in DRAM while the persistent versions store all data in NVM. We also evaluated UHTM with in-house hybrid key-value stores. They resemble the recent studies of persistent key-value stores [23], [41], [63]. The `Hybrid-Index` key-value store maintains two separate indexes, one for DRAM (e.g., B-Tree) and another for NVM (e.g., HashMap) while data are only stored in NVM [63]. Another hybrid key-value store we evaluated is referred to as `Dual` key-value store, which maintains two identical data structures (e.g., HashMap) and stores one in DRAM and another in NVM [23]. The foreground threads handle user requests and deal with the DRAM data structure. The foreground and background threads communicate through cross-referencing logs that operate similar to a producer-consumer model. The backend threads keep data

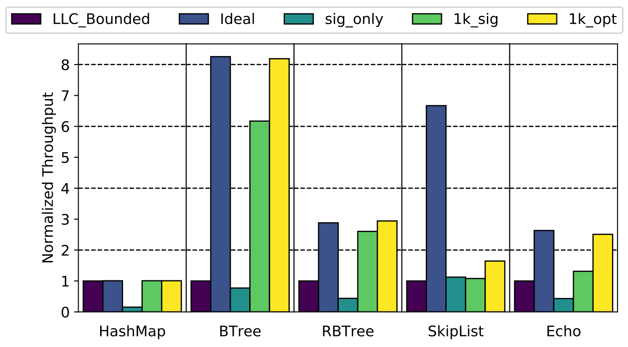


Fig. 6: Throughput of PMDK benchmarks and Echo key-value store running transactions of 100KB footprints. Plots are normalized to LLC-Bounded HTM.

structures in DRAM and NVM consistent. These hybrid key-value stores represent transactions manipulating both DRAM and NVM data. For the last class of benchmarks, we use the `Echo` key-value store from the WHISPER benchmark suite [43]. The master thread of the Echo key-value store manages a persistent hash table while clients threads batch and send updates to the master.

For all experiments, we consolidated four benchmarks with four threads. We evaluated our design with different footprints of transactions, ranging from 100KB to 500KB, which we controlled with the number of operations in a single batch. We decided to use a few hundred KBs footprints based on our measurements of SQLite transactions using Mobibench [29]. We found that they have about 160KB for read-set and 100KB for a write-set, respectively, when inserting a single entry to a database table. Furthermore, the sizes of durable transactions are expected to grow due to the failure-atomicity, which involves multiple data structures to be consistently updated [45].

To evaluate UHTM, we compare the following designs.

- **LLC-Bounded HTM.** The baseline hardware transactional memory system which provides durability through hardware-supported redo-based logging. This design limits the transaction boundary to on-chip caches since it uses the cache coherence to detect conflicts. This implementation is similar to the previous study [30].
- **Signature-Only HTM.** This represents a naive implementation of unbounded HTM system and is an extension of previous HTM systems to non-volatile memory [12], [64]. This system detects conflicts with only address signatures by checking all coherence traffics.
- **UHTM.** Proposed hardware transaction memory implementation that supports both durability and the unboundedness. This design provides unbounded transactions by incorporating address signatures for overflowed blocks and hybrid logging for DRAM and NVM data. We denote UHTM in two labels; `xxx_sig` and `xxx_opt`, which represents UHTM with and without the optimization of confining conflict domains, respectively, while `xxx` is the size of signatures.
- **Ideal Unbounded HTM.** The ideal hardware transactional memory system which provides unboundedness and durability. The conflict detection for overflowed blocks never produces a false-positive.

For `LLC-Bounded` HTM, a transaction does not attempt to retry if the transaction has overflowed and executes the slow-path right away.

VI. Evaluation

A. *Persistent Hardware Transactions*

In this section, we report the evaluation of UHTM with durable transactions having NVM data only. Figure 6 shows the throughput of PMDK benchmarks such as `HashMap`, `B-Tree`, `RB-Tree`, and `SkipList`, and the real-world application, `KV-Echo`. Figure 7 demonstrates the abort rates with the cause of aborting transactions. Each benchmark either inserts or updates the persistent data structure with the value size of 100KB. We also run two memory-intensive applications to emulate contention in LLC. Note that they also can cause transactions to abort by false-positivies in address signatures.

First, the capacity overflow significantly hurts throughput of durable transactions. `HashMap` does not experience the capacity overflows, and hence, the LLC-bounded HTM and UHTM do not show the difference in the transaction throughput. Since `HashMap` has a shorter transaction latency, the capacity overflow does not happen in this benchmark while it happens in others. Nevertheless, the capacity overflow severely degrades the throughput. For example, `B-Tree` and `SkipList` benchmarks incur 8.2x and 6.7x slowdown while `RB-Tree` and `Echo` show 2.7x and 2.5x throughput degradation, respectively. In particular, UHTM shows a substantial speed-up for `B-Tree`, `RB-Tree`, and `Echo` benchmarks, making them comparable to the ideal performance. On the other hand, UHTM underperforms in `SkipList`, showing less than 2x speed-up compared to the LLC-Bounded baseline while the ideal one achieves 7x speed-up. It turns out that UHTM ends up with many false-positives while `SkipList` traverse the list, which significantly diminishes throughput by false conflicts.

Second, reducing the false-positive rate of the address signatures is critical. Signature-only HTM systems produce the very high abort rates due to false-positives of address signatures. They under-perform even compared to LLC-bounded systems. Therefore, the key to designing efficient unbounded HTM is to maintain the false-positive rates of address signatures low. Figure 7 shows the abort rates of each signature configuration of UHTM and decomposes the cause of the abortion (e.g., true or false conflicts, and overflows). The x-axis of the figure represents the footprint of each transaction while labels on the figure, `xxx_sig`, represents UHTM with the address signatures of size `xxx`-bit. As the size of transaction increases, so do the abort rates. Although UHTM provides the performance gain over the LLC-bounded baseline as the size of signatures increases (from 512_sig to 4k_sig), it requires more space overheads for large signatures.

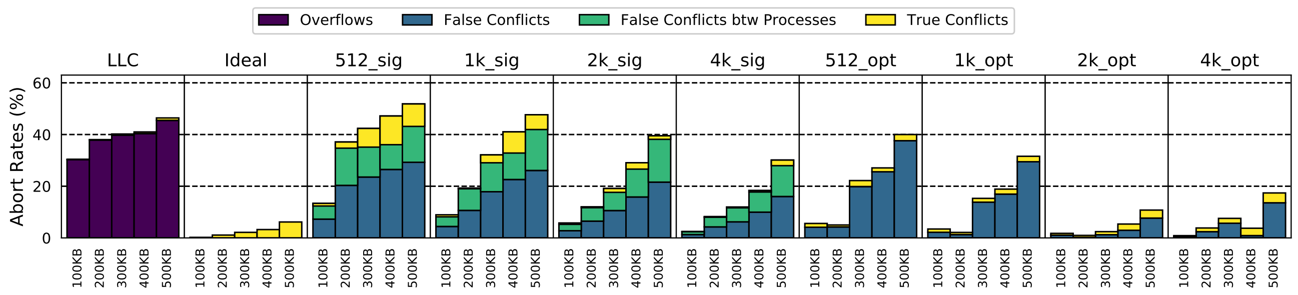


Fig. 7: Abort rates of UHTM when running PMDK benchmarks.

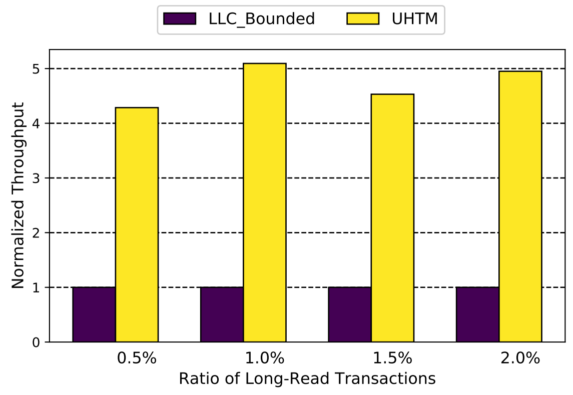
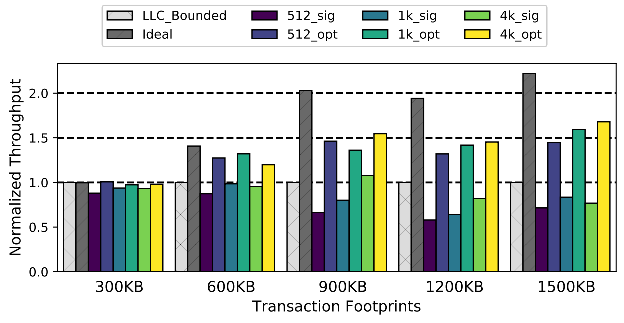
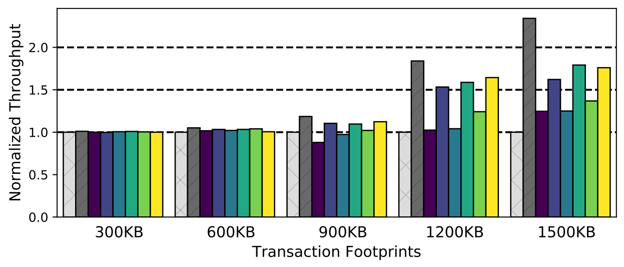


Fig. 8: Transaction throughput of Echo key-value stores with long-running read-only transactions.



(a) Hybrid-Index KV-store.



(b) Dual KV-store.

Fig. 9: Evaluation of the hybrid key-value stores [23], [63].

Lastly, when consolidating multiple persistent applications, UHTM's optimization that confines conflict domains of hardware transactions eliminates false aborts between different conflict domains, which contributes to a large portion of false-positives. By grouping address signatures with conflict domains, UHTM substantially improves throughput since applications less abort and restart. Furthemore, the signature isolation technique removes the false conflicts with the memory-intensive applications. Hence, confining conflict domains is essential to reduce false-positives of signature-based approaches.

B. Long-running Read-only Transactions

In this section, we demonstrate the need for unboundedness with the long-running and read-only transactions in the `Echo` KV-store [5] in Figure 8. Long-running and read-only transactions are rare but have significant implications on transaction throughput by causing the capacity overflow since their footprints far exceed the on-chip cache size. We did not run the memory-intensive applications in this experiment and configured long-running read-only transactions to be from 0.5% to 2.0% of the total operations. Each long-running and read-only transaction performs a batch of get-operations for randomly selected KV-pairs for the size of between 8MB and 32MB. Meanwhile, other transactions have a single put-operation with a value size of 1KB. As shown in Figure 8, the occurrence of long-running and read-only transactions drastically degrades the throughput of the LLC-bounded system. As a result, UHTM achieves 4.2x throughput improvements when having 0.5% of long-running read-only transactions. The existence of such long-running transactions makes bounded HTMs challenging to offer sustainable throughput. However, UHTM can support any size of transactions without harming programmability or performance.

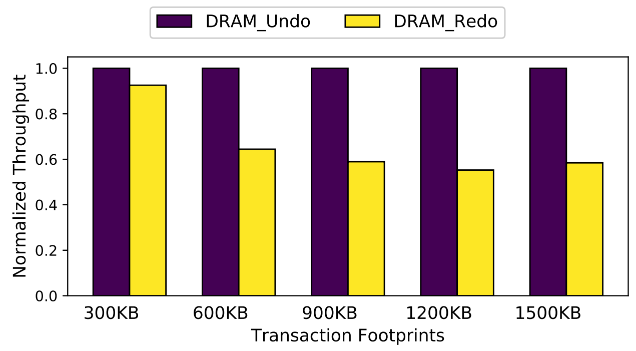


Fig. 10: Transaction throughput of volatile transactions when either logging with undo or redo when overflowed.

C. Hybrid Key-Value Stores

Figure 9 shows the results of hybrid-key value stores [23], [63]. We varied the size of address signatures from 512-bit to 4k-bit. We increased the footprints of transactions and did not run LLC-hungry applications. The `Hybrid-Index` key-value store, shown in Figure 9a, handles both DRAM and NVM indexes when inserting a new key-value pair but only stores data in NVM. The naive UHTM design (noted as `xxx_sig`) has poor performance since high false-positive rates hamper transactions to progress. The false-positive happens more frequently in hybrid-index key-value stores since transactions manipulate both DRAM and NVM, increasing the possibility of overflows. However, the optimization to confine the conflict domain within the process significantly boosts the performance by eliminating false-positives between different conflict domains. This optimization increases the throughput by 30.5% even with transactions with 600KB footprints, and the improvement reaches up to 64% with 1.5MB footprints.

On the other hand, the `Dual` key-value store has lower overflow rates than other benchmarks. This is because the communication between foreground and background threads are out-of-transactions, resulting in low aggregated footprints of active transactions. However, when overflows have occurred, the throughput of the LLC-bounded system drops substantially. Even the naive UHTM shows better performance by 15.6% and 32.6% when the footprints are 1.2MB and 1.5MB, respectively. The optimization improves performance by 62.3% and 74.7%, respectively.

D. Volatile Hardware Transactions

In volatile transactions, where all data are non-persistent (DRAM), the overflowed blocks can be handled in either undo or redo logging. The debate on which approach is superior to another depends on the abort rate if not considered capacity overflows. However, in the presence of overflows, the undo approach outperforms redo logging.

Figure 10 compares the results of the undo or redo logging for overflowed DRAM blocks. We averaged the results of UHTM having 512-bit, 1k-bit, and 4k-bit signatures with the optimization but only differing logging techniques for DRAM. When committing transactions, the undo approach ends by marking the commit mark on the log, offering the fast commit. On the other hand, the redo log needs to copy new values to in-place locations, making the transaction commit slow. When aborting transactions, the trade-off becomes the opposite way. Not only this trade-off but slow read performance of the redo approach contribute to low throughput compared to the undo log. The slow down becomes more significant as overflows are frequent. Our experimental result shows that the undo approach outperforms the redo log by 7.5% when footprints are 300KB (e.g., low overflow rate). When overflows are more frequent, the performance gap grows further up to 44.7%.

VII. Conclusion

To answer the calls for simple yet performant persistent programming, we presented UHTM supporting unbounded hardware transactions for DRAM and NVM hybrid memory systems. UHTM proposed two techniques; the staged conflict detection and hybrid hardware logging schemes. Experimental results show that UHTM significantly outperforms the state-of-the-art that limits the transaction to the size of on-chip caches and supports NVM data only therein.

Acknowledgment

This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-00118, Research and Development on Memory-Centric OS Technologies of Unified Data Model for Next-Generation Shared/Hybrid Memory). At Purdue University, this work was supported by NSF grants 1750503 (CAREER) and 1814430.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, 2005, pp. 316–327.

[2] A. Armejach, R. Titos-Gil, A. Negi, O. S. Unsal, and A. Cristal, "Techniques to improve performance in requester-wins hardware transactional memory," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 42:1–42:25, 2013.

[3] H. Avni and T. Brown, "Persistent hybrid transactional memory for databases," *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 409–420, 2016.

[4] U. Aydonat and T. S. Abdelrahman, "Hardware support for relaxed concurrency control in transactional memory," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 15–26.

[5] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring storage class memory with key value stores," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)*, 2013, pp. 4:1–4:8.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, pp. 1–7, 2011.

[7] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 24–34.

[8] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, "Tokentm: Efficient execution of large transactions with hardware transactional memory," in *2008 International Symposium on Computer Architecture (ISCA)*, 2008, pp. 127–138.

[9] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *Queue*, vol. 6, no. 5, pp. 40:46–40:58, 2008.

[10] D. Castro, P. Romano, and J. Barreto, "Hardware transactional memory meets memory persistency," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 368–377.

[11] D. Cervini, D. Porobic, P. Tözün, and A. Ailamaki, "Applying htm to an oltp system: No free lunch," in *Proceedings of the 11th International Workshop on Data Management on New Hardware (DaMoN)*, 2015, pp. 7:1–7:7.

[12] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006, pp. 227–238.

[13] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OPPSLA)*, 2014, pp. 433–452.

[14] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *Proc. VLDB Endow.*, vol. 8, no. 5, pp. 497–508, 2015.

[15] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 105–118.

[16] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 133–146.

[17] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 77–89.

[18] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1190–1201, 2015.

[19] "Firebirdsql," FirebirdSQL. [Online]. Available: https://firebirdsql.org/file/documentation/reference_manuals/fbdevgd-en/html/fbdg30-firedac-transactions.html

[20] J. S. George, M. Verma, R. Venkatasubramanian, and P. Subrahmanyam, "go-pmem: Native support for programming persistent memory in go," in *2020 USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 859–872.

[21] "Available first on google cloud: Intel optane dc persistent memory," https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory, Google.

[22] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017, pp. 468–482.

[23] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and S. Byan, "Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs," in *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 967–979.

[24] "Intel optane dc persistent memory module," Intel. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[25] "Intel persistent memory development kit (pmdk)," https://github.com/pmem/pmdk, Intel.

[26] Intel, "Programming with intel® transactional synchronization extensions," *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 1, p. Chapter 16, 2019.

[27] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for ibm system z," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 25–36.

[28] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 520–532.

[29] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won, "Androstep: Android storage performance analysis tool," in *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik*, 2013, pp. 327–340.

[30] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 452–465.

[31] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 361–372.

[32] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner, "Improving in-memory database index performance with intel®transactional synchronization extensions," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 476–487.

[33] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: Exploiting nvram in write-ahead logging," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 385–398.

[34] H. Kimura, "Foedus: Oltp engine for a thousand cores and nvram," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015, pp. 691–706.

[35] M. J. Kishi, S. Peluso, H. Korth, and R. Palmieri, "SSS: scalable key-value store with external consistent and abort-free read-only transactions," *CoRR*, vol. abs/1901.03772, 2019.

[36] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. 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)*, 2016, pp. 399–411.

[37] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009, pp. 2–13.

[38] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, 2014, pp. 580–591.

[39] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 329–343.

[40] Q. Liu, J. Lzraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "ido: Compiler-directed failure atomicity for nonvolatile memory," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 258–270.

[41] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris, "Persistent memcached: Bringing legacy code to byte-addressable persistent memory," in *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2017, pp. 4–4.

[42] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: log-based transactional memory," in *The Twelfth International Symposium on High-Performance Computer Architecture (HPCA)*, 2006, pp. 254–265.

[43] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 2, pp. 135–148, 2017.

[44] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 401–410.

[45] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 385–398.

[46] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 336–349.

[47] M. Ohmacht, A. Wang, T. Gooding, B. Nathanson, I. Nair, G. Janssen, M. Schaal, and B. Steinmacher-Burow, "Ibm blue gene/q memory subsystem with speculative execution and transactional memory," *IBM Journal of Research and Development*, vol. 57, no. 1/2, pp. 7:1–7:12, 2013.

[48] "Oracle databases," Oracle. [Online]. Available: https://docs.oracle.com/cd/B12037_01/server.101/b10743/consist.htm

[49] B. Pham, D. Hower, A. Bhattacharjee, and T. Cain, "Tlb shootdown mitigation for low-power many-core servers with l1 virtual caches," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 17–20, 2018.

[50] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture (ISCA)*, 2005, pp. 494–505.

[51] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008, pp. 246–257.

[52] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 91–104.

[53] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 178–190.

[54] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 175–186.

[55] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008, pp. 139–150.

[56] T. Shull, J. Huang, and J. Torrellas, "Autopersist: An easy-to-use java nvm framework based on reachability," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019, pp. 316–332.

[57] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 18–32.

[58] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[59] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 91–104.

[60] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 127–136.

[61] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent transactional memory," in *IEEE Computer Architecture Letters*, vol. 14, no. 1, 2015, pp. 58–61.

[62] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014, pp. 26:1–26:15.

[63] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017, pp. 349–362.

[64] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007, pp. 261–272.

[65] G. Zhang, V. Chiu, and D. Sanchez, "Exploiting semantic commutativity in hardware speculation," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[66] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 421–432.

[67] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009, pp. 14–23.