Haste Makes Waste: Accelerating Persistent HTM with Lazy Logging

***Abstract*—hardware transactional memory (HTM);**
atomic, durable - version management? - intra-transaction;
isolation? - concurrency control; - inter-transaction;
durability - logging / version management / memory persistency...;
in this paper, we present FlyHTM, a fast ...; with optimization of both;
***log/data ordering on the fly, explicit log persist instructions and sfence barriers;**
hardware-only?
software support?
processor modifications?
present two acceleration techniques, for logging and concurrency control ...;

I. Introduction

NVM random access (more than 300 ns) is 10× slower than DRAM [27]

==========

...;

HTM idea proposal [15];

Several NVRAM storage technologies promise performance and byte-addressability comparable to DRAM with the persistence of disk and flash memory [21], [29].

non-volatile memory, as known as persistent memory...;

Among these, both Intel [38] (Intel TSX) and IBM [16] already released commodity processors incorporating HTM.

commercial HTM [10];

==Kiln [39] and LAD [14] assumes additional persistent components closer to the CPU, — Kiln - LLC, LAD - MC queues; *not generalized as traditional work; — requires hardware modification to the CPU architecture and its cache coherence protocol [cite-hoop].*

HOOP [7]: similar to LAD and Kiln?

==In this paper, we ...;

In summary, we make the following contributions to mitigating conflict-based cache timing attacks.

===========

==PiCL [27]: Multi-undo logging allows multiple logical commits to be in-flight while still maintaining a single central undo log, removing the need for synchronous cache flushes.

By versioning individual cache lines, asynchronous cache scan further removes cache flushes from the critical path and asynchronously executes checkpoint phases to minimize performance overhead.

cache-driven logging. By preemptively sourcing undo data directly from the on-chip cache, it is now possible to buffer these entries before writing them to NVM.

==ThyNVM [33]: ThyNVM is a redo-based WAL design where memory translation tables are used to maintain both the committed and the volatile execution versions of the data. It has a mixed checkpoint granularity of both block-size and page-size, which can lead to good NVM row buffer usage for workloads with high spatial locality. ThyNVM also overlaps the checkpoint phase with the execution phase to minimize stalling, although it is still subjected to a synchronous cache flush stall at every checkpoint, and requires a translation table which is not scalable to large multi-core systems.

==========

Committing a volatile transaction requires that the read-/write-set tracking structures be cleared and that the speculative state be made visible to other threads. In addition to the above steps, in order to commit a durable transaction (with redo logging), the redo-log entries must be written to persistent memory.

======================

why redo logging's memory barriers inefficient: Memory barriers stall subsequent data updates until the previous updates by the transaction complete, However, this write-order control prevents caches from optimizing system performance via coalescing and reordering writes. [28];

focus on HTM (e.g., [19]), many related work on software TM;

***list all HTM and Persistent HTM related papers;

Second, NVM has closeto-DRAM read latency, but about 10x write latency comparing with DRAM, For example, PCM's write latency is 150∼1000ns and ReRAM's is 500ns, while DRAM has only 60ns write latency [13].

=================

II. Problem

In this section, we review the basics of persistent HTM and its building block for version management—write-ahead logging. It enforces eager logging in that triggers an immediate log at the L1 cache upon every `store` instruction and has to persist the log before data. We identify such logging eagerness as an efficiency bottleneck and investigate how it hinders performance.

A. Persistent HTM

Emerging non-volatile memory (NVM) has brought ACID into practise for in-memory transaction processing. This is mainly attributed to its disk-like storage durability and DRAM-like access latency. Supporting durable storage, NVM enables transactional memory solutions with the durability property.

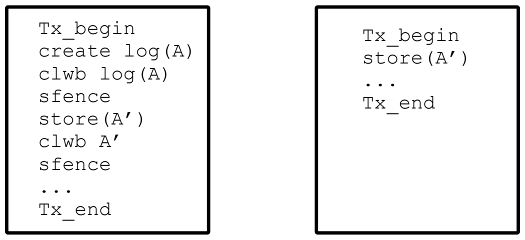
The CPU no longer has to use slow I/O processes to persist data into disks. Unlike disks, NVM offers fast byte-addressability [21], achieving ACID for in-memory transactions fairly fast. The state-of-the-art marries NVM with efficient HTM and is usually referred to as Persistent HTM [19].

Next, we review how traditional transactional memory partially fulfills ACID and how Persistent HTM bridges the gap.

Transactional Memory (TM). TM adopts in-memory transaction processing for concurrency control in multicore systems [15]. Multicore systems can perform tasks concurrently and thus provide better performance than single-core systems. The performance improvement relies on the parallelism of thread execution. This requires concurrency control to synchronize across threads such that they generate the same results as if they had executed on a single core. The major intricacy arises from orchestrating threads that access the same data. It is therefore modeled by TM after database transactions, which should satisfy the ACID properties.

Traditional TM solutions based on volatile DRAM, however, support only atomicity, consistency, and isolation without durability. Atomicity requires that a transaction appear to be having fully executed if it is successfully committed or having not executed at all otherwise. In the latter case, even if a noncommitted transaction has made modifications to certain data, its modifications cannot take effect. Consistency requires that a committed transaction cannot lead to a state of data values that can hardly be produced by any serial execution of transactions [15]. Isolation requires that the result of a transaction not be tampered with by concurrent transactions. It does not necessarily prohibit transactions from accessing the same data. Instead, transactions may be allowed to share data as long as they do not violate consistency. This requirement is also defined as serializability [4], [17], [30], [32], following which the result of concurrent transactions conforms to one obtained by their serial execution.

Persistent HTM. The advent of NVM augments TM with durability. Recent advances favor its efficient hardware implementation [6], [12], [19], [26], [28]. Traditionally, Software TM (STM) [20], [23] requires programmers to manually instrument memory references in programs [8]. As shown in Figure 1(a), STM starts and commits a transaction using instructions of `begin_transaction` and `end_transaction`, respectively. Inside a transaction, old or new data states should be logged prior to their in-place updates. This requires many more instructions for creating logs (e.g., `create`), persisting logs (e.g., `clwb`), and stalling memory accesses while persisting logs (i.e., memory barrier `sfence`). These extra instructions induce a heavy overhead to not only the programmers but also the [11], [23], [26], [35]. In contrast, Hardware TM (HTM) [6], [12], [19], [20], [26]–[28] reserves only the instructions for delineating transactions while pushing all other instructions for orchestrating memory references to hardware (Figure 1(b)). Persistent HTM thus promises a much efficient way for in-memory transaction processing [5], [19], [28].Problems: difference between STM and HTM



```
Tx_begin
create log(A)
clwb log(A)
sfence
store(A')
clwb A'
sfence
...
Tx_end
```

(a) transaction with software logging

```
Tx_begin
store(A')
...
Tx_end
```

(b) transaction with hardware logging

Fig. 1. comparison of transaction execution with SL and HL

Toward efficiently fulfilling the ACID properties, Persistent HTM needs to craft two orthogonal design policies—version management and conflict management [25]. First, version management guarantees crash consistency on an intra-transaction scale. If a crash occurs, the recovered data of a transaction should be consistent in that they entirely follow either the modified version or the original version. Second, conflict management detects and resolves inter-transaction conflicts to guarantee the isolation property. A potential conflict occurs when transactions concurrently access the same data and at least one of them tries to modify it. Version management is fundamental for guaranteeing transaction correctness while conflict management matters for exploring transaction parallelism. In this paper, we focus on version management and aim to enforce it in hardware with high efficiency.

B. Version Management: Write-Ahead Logging

Version management usually uses write-ahead logging (WAL) to guarantee crash consistency. Logging, as the key principle of WAL, maintains both original (old) and modified (new) versions of data in the memory hierarchy. It allocates a dedicated log area in NVM for recording certain versions of data. WAL ensures that logs are persisted into NVM prior to committing a transaction. Then new data can be updated in place upon commits. Once the system crashes during a commit, we can use the persisted logs to recover either the old or the new version of the entire data set of the crashed transaction. Such data recovery guarantees data consistency in that the recovered data set is not mixed with old and new values. WAL is widely adopted because of simplicity (e.g., in-place updates) in comparison with other alternatives. For example, shadow paging [7], [20] uses the log area directly for storing updates. Without in-place updates, subsequent accesses to the new data values should be redirected to the log area. This necessitates a redirection table that maps the address storing the old data to the address storing its new value. Further updates of the new value makes address redirection a costly, recursive operation.

According to whether the log area persists old, new, or both versions of modified data, WAL can be classified into undo logging, redo logging, and undo+redo logging, respectively (Figure 2).

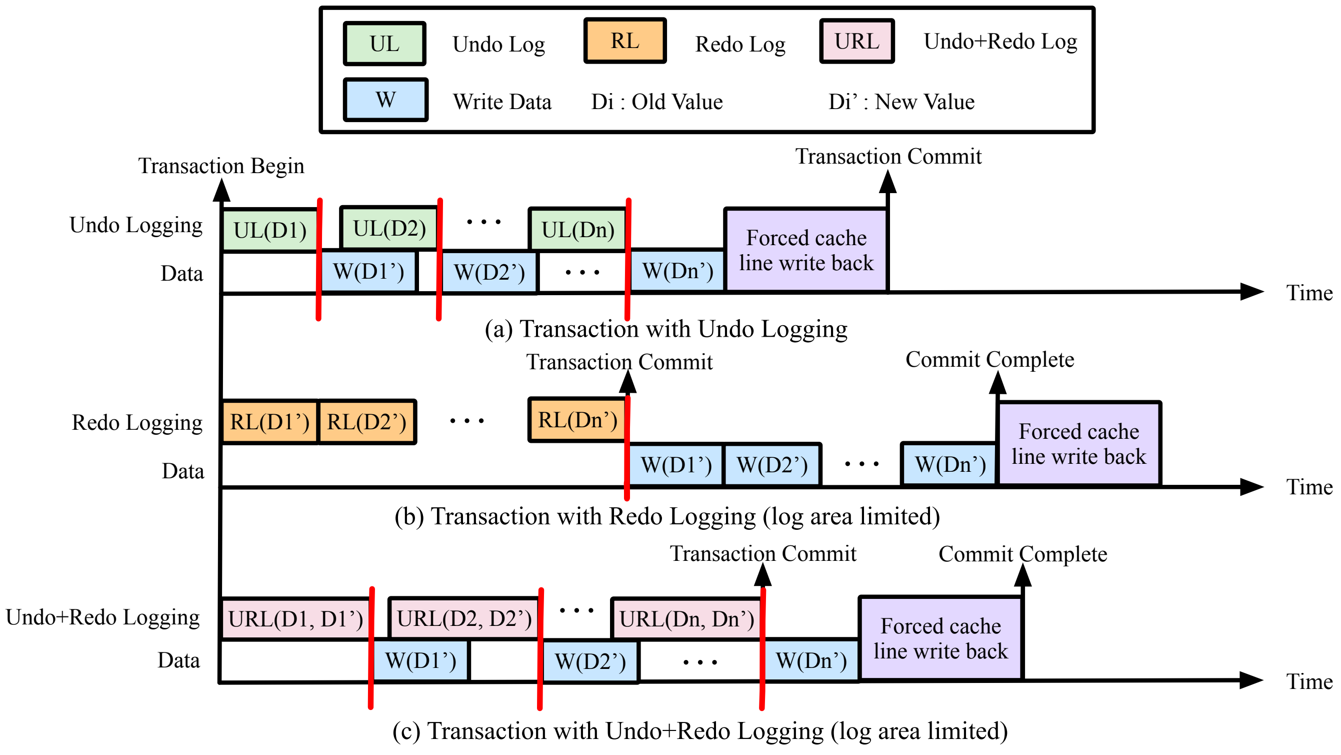


Fig. 2. Comparison of executing a transaction with persistent memory with (a) undo logging, (b) redo logging, (c) undo+redo logging

Undo logging logs the old data and supports in-place updates prior to transaction commits. As shown in Figure 2(a), log and data writes resemble a pipeline [28]. Once after an undo log (e.g., UL(D1)) is persisted, the corresponding write can take place over the data (e.g., W(D1') for updating D1 to D1'). The modified data can be directly written to NVM upon cache eviction. When the transaction is about to commit, some of its modified data might still be in caches. Such data need to be written to NVM as well before the transaction commits, using forced cache line write-back commands such as `clwb`. Prior to and during a transaction commit, the system may crash. If this happens, it is likely that some modified data have been written to NVM while some others are still in caches. We can use undo logs to revert modified data in NVM to their old values and guarantee consistency [20], [26], [27], [34].

Redo logging logs the new data and modifies data only after it persists all the corresponding redo logs. As shown in Figure 2(b), all the redo logs (i.e., RL(·)) recording new values of D1' and Dn' should be persisted prior to data writes (i.e., W(·)). In other words, redo logging persists logs and writes sequentially [11], [18], [19], [23], [24], [35]. Otherwise, data consistency might be violated upon system crashes. For example, if 1) we persist both RL(D1') and W(D1') prior to other logs and writes and 2) the system crashes before the transaction commits, the system contains only the new value for D1 while:

- only the new values for data whose logs and writes are also persisted as well;
- only the old values for data whose logs and writes are not persisted yet; or
- both old and new values for data whose logs are persisted while writes are not.

In this case, a system crash can easily leave the system mixed with data that cannot simultaneously recovered to old or new values. Redo logging imposes more complexities on reads than undo logging does. Because date are not updated in caches or NVM during log creation, read accesses need to go through the log area first in case cached or in-place data are already obsolete [14].

Albeit redo logging does not enforce cache line write-back prior to commits, it may require so after commits [28]. This requirement is for guaranteeing consistency when the log area is limited. In this case, some logs need to be evicted upon log overflow. If their logged new values are not written back to NVM in time, a system crash will make these new values vanish and lead to inconsistency.

Undo+redo logging reaps the benefits of both worlds. As shown in Figure 2(c), it logs both the old and new versions of modified data [28]. First, undo logs enable modified data to be updated in place before the transaction commits. Second, redo logs help remove the forced cache line write-back prior to commits. However, as an intrinsic limitation of redo logging, undo+redo logging still requires the forced cache line write-back upon log area overflow. Furthermore, recording both the old and new versions requires more logging time and a larger log area [18].

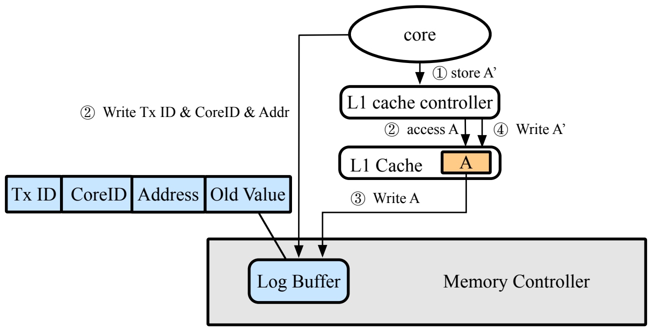


Fig. 3. Previous Hardware Undo Logging Procedure

C. Limitation: Logging Eagerness

Albeit various WAL improvements, we identify eager logging as its efficiency bottleneck. Specifically, eager logging triggers an immediate log at the L1 cache upon every `store` instruction [18]–[20], [28]. The log needs to be persisted before data according to the WAL principle (Figure 2). Therefore, previous solutions often generate logs before in-place updates in caches and write logs to the memory controller by bypassing the cache hierarchy to ensure that logs run faster than data do. Figure 3 showcases the logging eagerness through a typical undo logging scheme.

- Step ①: The L1 cache controller receives a write request from the CPU core to store A'. The `store` instruction triggers the creation of an undo log including the meta-log information and old value of A.
- Step ②: The core collects the meta-log information (e.g., the transaction ID and address) and writes it to the log buffer. Meanwhile, it accesses the L1 cache to fetch the old value of A.
- Step ③: The old value of A is written directly to the log buffer. Together with the meta-log information in Step ②, it completes the log entry.
- Step ④: After the log is ready, the new value of A' is updated to the cache line.

Obviously, eager logging introduces an inevitable delay between when the write request arrives and when the corresponding in-place cache update takes place. All the extra operations for logging (e.g., collecting the meta-log information, fetching the old value if undo logging or the new value if redo logging from caches and writing it to the log entry) cost additional clock cycles.

Furthermore, eager logging may induce wasted efforts when a transaction enforces multiple writes to the same data block. For undo logging, the log needs to track the old value. The first log of each modified data block suffices for recovery. Solutions such as ATOM [20] and PiCL [27] simply add indicator bits per cache line to track whether the data therein have been logged. However, for redo logging, the last update of the modified data block dominates. Since eager logging creates a log right after a write request arrives, a log can easily become obsolete when the logged data are modified again. The more frequent a data block is modified, the more overhead is induced by such obsolete logs. In other words, eager logging leads to more potential inefficiency for write-intensive transactions.

Finally, another inefficiency arising from eager logging is due to locality unfriendliness. As shown in Figure 3, writing logs bypasses the cache hierarchy and thus ignores the temporal and spatial locality therein. To regain the locality for efficiency, some solutions temporarily buffer logs between the cache and memory controller [19], [27], [28]. Buffered logs can be coalesced into larger chunks to improve bandwidth efficiency. However, this necessitates additional buffer space and complexities for processing logs therein.

III. Overview

In this section, we present lazy logging to address the inefficiency caused by eager logging in existing persistent HTM designs. It directly leverages LLC-cached data as undo logs and writes logs to the memory controller upon cache eviction. This way, we can immediately update L1-cached data without stalling the write until the log is created and written to the memory controller as in eager logging. We implement lazy logging through our faster persistent HTM called FlyHTM.

A. Motivation

We observe that a log is not immediately needed when a cache line is modified. Instead, it suffices if the log can become available in the persistent domain before data are written back. Modified data usually stay in caches and may not be written back for a while. This offers a relaxed time bound between the arrival of a write request and the writing of the log it triggers. If we revive the relaxed time bound by filling it with in-cache updates, we expect to speed up transactions. The later we delay the log creation and writing process, the faster we can complete write requests of transactions. This motivates us to explore lazy logging as a fundamental acceleration technique for persistent HTM.

B. Methodology: Lazy Logging

Lazy logging accelerates persistent HTM by decoupling log creation from the arrival of write requests. It directly leverages LLC-cached data for undo logging. In commonly used inclusive caches, the LLC-cached data are a superset of the L1-cached data. When a write request tends to modify a L1-cached data block, we do not have to write the L1-cached old value back to the memory controller for undo logging. This is because the old value is also in the LLC. Even if we immediately modify it in the L1 cache, we would not miss the old value. In a nutshell, lazy logging can perform in-cache updates upon receiving write requests and prepare undo logs nearly at the same time, without interrupting consecutive writes by log creation. The old values are written back to the memory controller as undo logs when they have to be evicted from the LLC. This mitigates the overhead of eager logging to the maximal extent. Furthermore, the delayed log creation also promises more chances of log coalescing.

We now investigate more about how lazy logging accelerates `store` instructions and optimizes log coalescing.

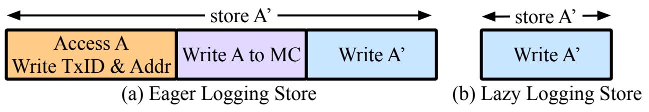


Fig. 4. Comparison the `store` instruction of eager logging and lazy logging

Shortened execution of `store` instructions. Figure 4 compares the execution timeline of a `store` instruction instance under traditional eager logging and our lazy logging. Let A denote the old value and A' denote the new value to be written. Under eager logging (Figure 4(a)), the `store` instruction is tightly coupled with log creation. Log creation enforces extra operations such as writing the meta-log information (i.e., transaction ID and address of A) to the log buffer, accessing caches to fetch the old value of A, writing A to the log buffer, and merging A with the meta-log information into a log entry. Only after all these extra operations can we modify the cache line with the new value of A'. In contrast, our lazy logging eschews all these extra operations prior to in-cache updates. The `store` instruction immediately modifies A to A' in caches as if it were a non-transactional access. Meanwhile, we track the meta-log information via the cache coherence directory (Section IV-D), which should be updated upon write requests anyway. We piggyback the meta-log information into the directory without much additional overhead.

Inherent support of log coalescing. Since lazy logging leverages the LLC-cached data as logs, it directly benefits from the inherent locality therein. In eager logging, previous solutions create a log immediately after a write request arrives. To coalesce logs, they usually introduce an extra buffer to temporarily hold logs such that they can be coalesced into larger blocks before being written to the memory controller or memory. In contrast, our lazy logging uses the LLC to hold logs and thus does not require an additional buffer. Furthermore, logs stay in the LLC until the cache lines they reside have to be evicted. Each LLC cache line intrinsically contains address-adjacent data. This saves lazy logging from much extra overhead to coalesce logs for transactions with high spatial locality.

C. Architecture: FlyHTM

System model. We use the lazy logging technique to develop a faster persistent HTM called FlyHTM. As with existing HTM solutions, the core of FlyHTM resides on the CPU chip. Figure 5 illustrates the architecture of FlyHTM. Before delving into the specific logging framework, we define the system model. We consider a multi-core system with a commonly used three-level cache hierarchy. Each core has a private L1 cache and a private L2 cache. All cores share the Last-Level Cache (LLC). The caches are inclusive and write-back with write allocate. To guarantee cache coherence and detect transaction conflicts, we use the MESI directory-based protocol [9] for tracking data states across private caches.

Motivated by Proteus [34] and ATOM [20], FlyHTM adopts a battery-backed persistent memory controller. The benefit is to persist logs directly in the memory controller without having to write them to the much slower NVM. Another motivation behind this design choice is that most logs persisted into the NVM are not really needed. HTM solutions keep logs persistent for recovering data to a consistent state when the system crashes. However, system crashes occur rarely and transactions can commit successfully more often. Therefore, writing unnecessary logs to the NVM wastes considerable memory bandwidth and induces high overhead. Yet a dilemma arises when we cannot make sure whether the system may crash before the ongoing transaction commits. In other words, we can hardly decide which logs can be omitted. Using a persistent memory controller helps to mitigate the dilemma because persisting logs into the memory controller is much faster than into the NVM. Only when logs overflow the memory controller need them be written to the NVM. Furthermore, once a transaction commits, its logs can be discarded to save space.

Logging framework. Figure 5 illustrates how FlyHTM performs lazy logging stepwise. At a high level, the core executes a transaction and the modified data during transaction execution remain in private caches. Meanwhile, the undo logs contain the old values of the modified data; they remain in the LLC until the corresponding cache lines are evicted upon replacement or flushed upon transaction commits. We walk through the logging steps in Figure 5 as follows.

- Step ①: The core issues a write request to the L1 controller. Instead of enforcing the L1 controller to create a log as in eager logging (Figure 3), FlyHTM directly uses the already cached old value in the LLC as the undo log. It can easily use the addresses maintained in the write set to track which data in the LLC are used as logs, without any extra operations on the LLC cache lines. Since no extra log creation is required, the write request can immediately update data in the L1 cache.
- Step ②: FlyHTM flushes undo logs from the LLC to the memory controller in two cases. The first is when the cache line containing undo logs is evicted due to replacement. The second is when the transaction commits and flushes non-persisted logs prior to data. In both cases, the old value is evicted from the LLC to the memory controller while the core writes the transaction ID and address to the memory controller. All these items form a complete undo log entry.
- Step ③: To maintain the inclusion property of inclusive caches, evicting an LLC cache line triggers back invalidation on the L2 and L1 cache lines containing the same data block. Specifically, we first invalidate both the cache lines in the L2 and L1 caches. Then we write the new value from the L1 cache to the memory controller, which further persists the data into the NVM. This ensures that logs are persisted prior to data, a requirement of WAL.

IV. Design And Implementation

In this section, we detail the FlyHTM design.

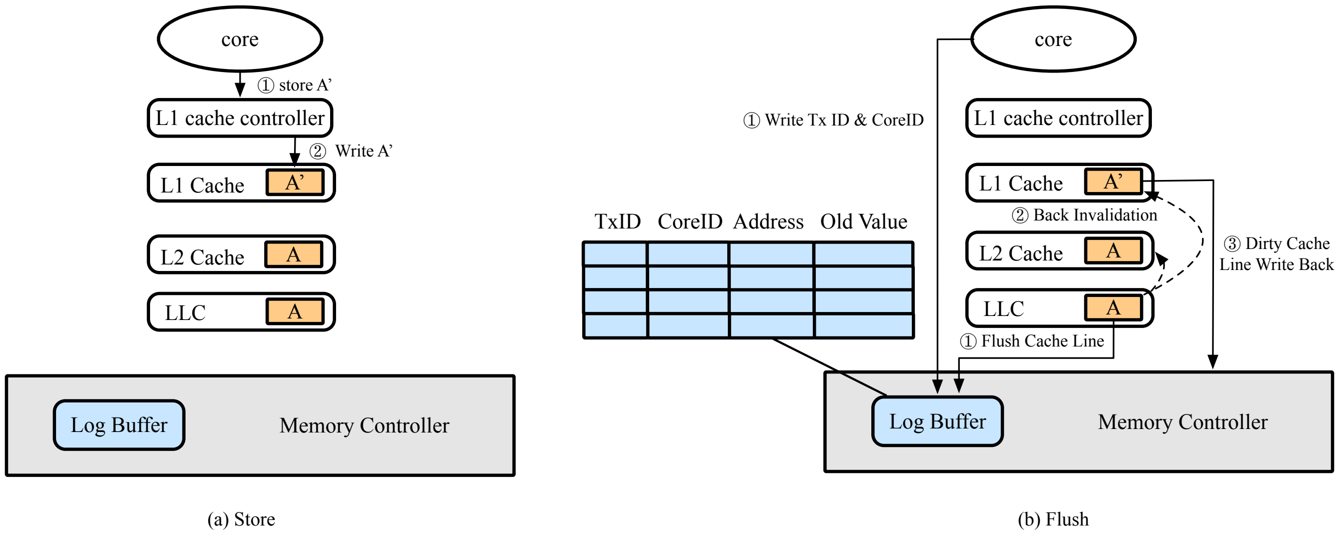


Fig. 5. FlyHTM architecture with lazy logging. related to Figure 3, need NVM unit?

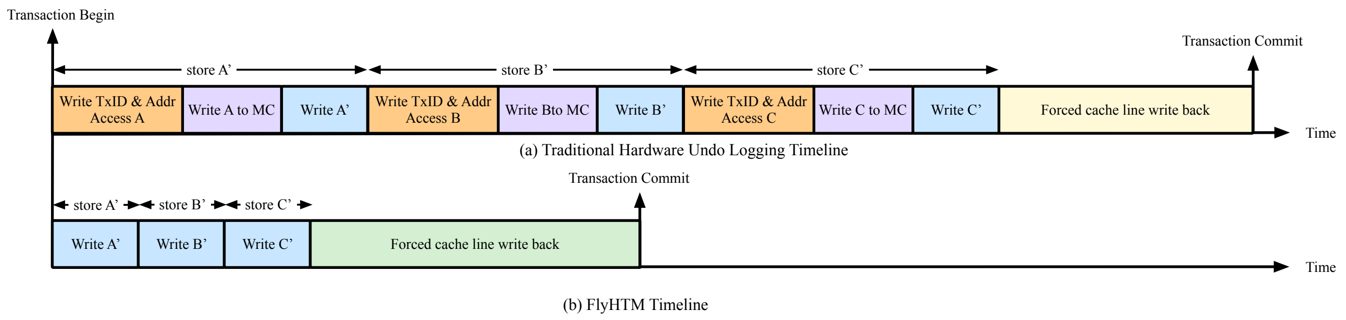


Fig. 6. Logging Timeline

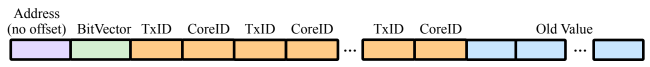


Fig. 7. FlyHTM log structure

A. Log Structure

FlyHTM uses undo logs that are originally the cached data in the LLC and, upon eviction, maintained in a centralized log buffer in the memory controller. Between undo logging and redo logging discussed in Section II-B, we choose undo logging in favor of its simplicity in log management and swiftness in in-place updates. First, it logs old values that need to be logged only once throughout the entire life cycle of a transaction. No matter how many times a data block is written, they originate from the same initial, old value. In contrast, redo logging generates a log each time the data block is written. This not only complicates log management but also expands log volume. In comparison with redo logging, undo logging can thus minimize the log volume to be proportional to the amount of updated data instead of the times they are updated. Second, undo logging allows in-place updates right after the associated undo logs are persisted. Such a swift in-place update saves undo logging from redirecting reads requests to logged new values as in redo logging. Furthermore, different from redo logging, undo logging does not have to persist logs until all the logs of a transaction are available. This greatly eases the pressure on the demand of log buffer area. Following existing work [20], [28], the centralized log buffer resides in the memory controller.

Traditionally, undo logging has a granularity of data block and each undo log entry consists of four fields—a transaction ID (TxID), a core ID, an address, and an old value. The transaction ID specifies the transaction the log entry belongs to. The core ID represents the core that executes the transaction and helps to distinguish different threads. The address and old value track the address of a modified data block and its original value, respectively. When a transaction is aborted, we first locate its log entries using the associative TxID and core ID. Then we locate the in-place updated data block and recover it to the old value.

FlyHTM, however, maintains variable-length undo logs at a cache-line granularity.

given all blocks ... in the same cache line...;

then common address

The log area may or may not store the unlogged blocks depending on space capacity.

Ideally, extra 16-bit vector, yet omit repeated address prefixes (tag and index) except offset...;

favor more compact ... logs

save common address from the log entries ...;

address / offset...;

address - bit vector - sequence of transaction id and core id

cache line based, more than one old values, piggyback more than one logs

should specify ...

==address from cpu to memory controller

then controller merges them into a more compact one...;

==synthesized ones ...: bit vector - log with 1, and unlog with 0

txid and core id

cache line of old values;

optimized ones later in Section IV-E...;

Log field parameterization.

txid

coreid

address - traditional

old value - one cache line, more discussion in Section IV-E...;

Then we need to decide the length of each log entry field. For the transaction ID, if it's too long, storage will be wasted; if it's too short, a new transaction may be allocated a ID the same as a previous uncommitted transaction. We think 16 bits is a proper size [28]. As for the Core ID, the field size depends on the number of cores. If the system has 16 cores, then the field size is 4 bits. Then, the address field takes 48 bits. The size of old value is the length of an LLC cache line. Here we apply 64 bytes.

To manage logs, there are two major tasks: log creation and log persistence. Log creation focuses on collecting log entry fields and organizing them to form a complete log entry while log persistence focuses on making logs persistent and controlling the ordering of logs and data. In brief, we couple log creation with the LLC cache line eviction and directly leverages inclusive's inclusion property to guarantee logs persisted prior to data. Next we will talk about how FlyHTM manages these two tasks in detail.

== creation to discuss how cpu and memory controller collaborate to ... these log structures...;

B. Log Creation

Log creation concentrates on two parts of content: sources of the log entry fields and the procedure to create a log entry. FlyHTM's log entry fields' sources resemble previous designs but in the procedure it delays log creation to until cache line eviction.

Sources of log entry fields. In FlyHTM, A log entry consists of four fields: undo value, address, coreID and TxID. Firstly, undo value is obtained by leveraging existing cache information. Traditionally there are two ways to get undo value: direct update which gets undo value from caches [18]–[20], [28], [34] and indirect update which get value by accessing NVM [11]. Given that indirect update requires additional NVM reads which wastes considerable time and bandwidth, FlyHTM chooses direct update. And for every update to a cache line in L1 cache, we can definitely find its original value resides in LLC because it uses inclusive cache. FlyHTM directly leverages this LLC information as undo value. Secondly, the address information is carried when a cache line is written back, which forms the address part of the log entry. As for the core ID, it can be obtained from the CPU ID register. Finally for the transaction ID, the core will allocate it for each transaction and a special register is added to record this information [36].

The procedure to create a log entry. FlyHTM couples log creation with cache line eviction. Upon cache line eviction, it collects log entry information from above sources and write them to the log buffer in the memory controller to form a complete undo log entry. Following shows the detailed procedure.

- The creation of log is triggered when a cache line is evicted from LLC by flushing transaction's write set.
- Upon eviction, the undo value together with its address is written back to the log buffer in MC.
- The cache line eviction will also trigger the transaction ID and core ID to be sent to MC. Thus, a complete log entry is created.

C. Log Persistence

After creating a log entry, the task turns to making it persistent. As we use write-ahead logging, to maintain crash consistency, a key problem is that logs should be persisted prior to data. Instead of using uncacheable logging that bypasses the cache hierarchy to save time [19], [28], [34], FlyHTM achieves inherent ordering control between log and data with the help of inclusive caches.

Inherent ordering between log and data. FlyHTM directly leverages the inclusion property of inclusive caches to enforce the write ordering of log and data. For inclusive caches, if the data in the LLC cache line to be evicted is also present in L1 cache, then in order to maintain the inclusion property, back invalidation is required to invalidate the cache line in the L2 and L1 caches. So, in FlyHTM, LLC cache line that is regarded as an undo value is written to the MC first, then comes to flushing modified value in higher cache hierarchies. Following illustrates concrete procedure.

1) FlyHTM evicts a cache line in LLC which is regarded as undo value to the log buffer in MC
2) it sends back invalidation signal to private caches.
3) Private cache writes the corresponding dirty cache line (new value) back to MC.

D. Cache Flush

When a transaction finishes executing, to make all the updates persistent, cache flush is required to flush the modified data back into persistent storage. According to previous researches [18], [27], based on the criteria that when updates in NVM take place, flush can be categorized into synchronous flush [20], [34], [35] and asynchronous flush [18], [27], [28]. Under synchronous flush, modified data should be written back before commit, while for asynchronous flush, updates in NVM can be delayed. The choice of synchronous flush and asynchronous flush is a trade off between performance and complexity. Getting rid of the time of waiting all the modified data to be flushed in the critical path, asynchronous flush provides better utilization of bandwidth and performance. However, it makes compromises with cache modifications, isolation level and complex mechanisms. FlyHTM can adopt either version of synchronous flush and asynchronous flush.

To implement asynchronous flush is not hard. First, it needs to allow the existence of non-persistent modified data after transaction commit and allow such data be accessed by other transactions.

As like above techniques, we can tag each cache line with a transaction ID.

===flushing procedure

FlyHTM adopts synchronous flush.

we can also easily implement asynchronous flush. ??

Our flush targets are modified cache lines and their corresponding logs. Because the use of inclusive cache, we only need to flush logs in LLC. In our design, log and data share the same address, the eviction of log in LLC will directly cause the write back of modified data in L1. This introduces a further problem: How to target at modified data's address in LLC. Flush in L1 is easy. We can identify a modified cache line according to its dirty bit. But in LLC, to flush logs, we don't have such apparent information. The main reason for this problem is that we do not have the address information of the modified data. So, we need to get the transaction's write set. Previous researches use Bloom Filter to solve this problem [3]. still some doubts. We propose to directly leverage the directory cache coherence protocol. The directory that resides in LLC maintains the information of all cache lines' states. In term of cache line's state, we can infer the write set information. When the transaction tends to commit, the LLC controller will start to flush. According to the address information provided by write set, LLC flush corresponding log back to MC.

L1 and L2, write-back from one level, simply invalidate-bit set in another level?

E. Log Area And Log Overflow

In a transaction, log takes the responsibility of failure recovery. Writing logs back to NVM costs a lot of time and bandwidth. However, failures occur occasionally. Most of the time, transaction will commit successfully and log will not be used. It doesn't sound like a good deal to sacrifice performance for the accidental crash. In order to eschew this cost, FlyHTM adds MC to the persistent domain. The moment data reaches MC, it becomes persistent. We introduce a log buffer in MC as a separate write pending queue for logs as Proteus [34]. Avoiding writing log entries to NVM. ===================

We desire to delay the write back process of logs as much as possible. Logs only need to reside in the log buffer and don't need to be forced written back to NVM. The structure is shown in Figure **??**.

add introduction/figure of circular array;

If the log buffer in MC is unlimited, then the problem is solved. But it is size-limited, which is the common fault of hardware methods. When dealing with slightly large transactions, log overflow is inevitable. Thus, we need to turn to NVM for help. Our principle is: if log buffer doesn't overflow, log stays entirely in MC's log buffer; if log buffer has overflow risk, the logs of the transaction that causes overflow must entirely be written to NVM. More than 80% of the buffer occupied is the signal that the buffer has overflow risk should be hardware only. The overflow will trigger an exception to ask the OS for help. According to the transaction ID, OS will allocate a separate space in NVM to store logs. The space is still organized into a circular structure. Then the log buffer is informed to write the logs of the transaction that has overflow risk back to NVM.

F. Efficiency Analysis

analyze efficiency, in terms of transaction time ...;

Here we give our time analysis. We can divide the whole procedure into two major parts: executing and flushing. For convenience, we only take the `store` instruction into consideration in execution. Suppose a transaction stores A, B, C, D four variables ???why this assumption, based on Figure 6? redraw and explain;.

Execution Time. In traditional hardware undo logging designs, log generation is coupled with the `store` instruction. It requires extra time to create a log entry. To complete a store, we need take the time of judging whether log should be generated and accessing the L1 cache to retrieve old value into consideration.

$$Time_{Prev\_Store} = Time_{Store} + Time_{Judge} + Time_{Access}$$

While in FlyHTM design, we decouple log creation and the `store` instruction. The log creation is delayed until the cache line is going to be evicted out of caches. Thus, it doesn't need simplex steps when dealing with the `store` instruction. Obviously, we can arrive at the conclusion:

$$Time_{Prev\_Store} > Time_{FlyHTM\_Store}$$

Flushing Time. Traditional methods only need to flush dirty cache lines back to NVM because of log is coupled with write modification and has been persisted already. But FlyHTM needs to flush both log and data. We think it won't affect performance, because writing logs and writing data can run in parallel. Some may doubt that flushing L1 is faster than flushing LLC, because LLC L1? has shorter access latency. We think it's not the case. Because LLC actually does coalescing and make more use of spatial locality. Though it needs more time to flush once, but it requires much less flushes. For a transaction that has spatial locality, FlyHTM's flushing time will be shorter.

$$Time_{Traditonal\_Flush} \approx Time_{FlyHTM\_Flush}$$

not quite accurate;

Total. In total

$$Num_{store} \times Time_{FlyHTM\_Store} + Time_{FlyHTM\_Flush}$$
$$< Num_{store} \times Time_{Traditional\_Store} + Time_{Traditonal\_Flush}$$

Upon our time analysis, our FlyHTM design runs faster than traditional undo logging methods.

G. *Lazy Commit*

delay commit until several transactions ...; not necessarily flush and commit a transaction ...; as its modified data might be used and modified by subsequent transactions soon after its commit;

V. Implementation

A. *Software Support*

Log Support.

B. *Non-Volatile Area Extension*

Memory Controller Extension. Persistent MC is introduced into FlyHTM. In memory, we add a log buffer to store log entries. The log buffer is organized into a circular queue as Figure **??** reveals. We keep track of the

C. *Volatile Area Extension*

Added Registers. To support transactions, we add some special registers.

D. *Hardware Overhead Analysis*

VI. Evaluation

Implementation and configuration. We implement PhantomCache using ChampSim [1], a trace-based microarchitecture simulator. As with hardware-only randomized mapping solutions (e.g., NewCache [22], [37] and CEASER [31]), our modification over the inclusion-enabled ChampSim touches only the LLC module. We adopt LRU as the replacement policy. Table **??** shows the architectural configuration.

Workloads. We evaluate PhantomCache performance by running workloads from the SPEC CPU 2017 benchmark package [2]. Specifically, we use all the 20 benchmarks from the SPECspeed 2017 Integer and SPECspeed 2017 Floating Point suites. We run at least 2 billion instructions per workload. The first 1 billion instructions are used for warming up the cache while the other 1 billion or more instructions are used for collecting performance statistics.

Metrics. We evaluate PhantomCache using three performance metrics—instructions per cycle (IPC), misses per 1,000 instructions (MPKI) of LLC, and miss rate of LLC. To evaluate how PhantomCache impacts cache performance, we normalize all these metrics using the ratio of PhantomCache's metrics to that of the baseline cache without modification. A higher normalized IPC indicates a better performance, exceeding 100% if PhantomCache outperforms baseline. Moreover, a lower normalized MPKI or normalized miss rate demonstrates a better performance. To measure aggregate performance, we further report the geometric mean of normalized IPC and the average of normalized MPKI and normalized miss rate.

Results. Based on the configuration in Table **??**, the results show that, to secure the LLC against the powerful $\mathcal{O}(|E|)$ attack, PhantomCache introduces only 0.3% slowdown on average if the LLC is manufactured to support its r-sets parallel-search requirement (Section **??**-Section **??**). Furthermore, for securing an 8-bank 16 MB 16-way LLC, PhantomCache leads to only about 0.5% slowdown on average because of sacrificing certain parallelism (Section **??**).

VII. Conclusion

We have studied the idea of exploiting localized randomization against conflict-based cache timing attacks. It proves to have the same strong defense effect as global randomization countermeasures and avoids the inefficient mechanisms in preceding global randomization designs such as random replacement and dynamic remapping. We implement localized randomization through PhantomCache. The analysis of its security shows that the attacker cannot successfully launch a conflict-based cache timing attack within 100 years when the degree of PhantomCache is set to 8. Finally, we implement PhantomCache using ChampSim and the evaluation shows that PhantomCache only brings a 0.5% performance degradation and affordable hardware overhead.

=============

TABLE I
Formatting guidelines for submission.

Field	Value
File format	PDF
Page limit	11 pages, **not including references**
Paper size	US Letter 8.5in $\times$ 11in
Top margin	1in
Bottom margin	1in
Left margin	0.75in
Right margin	0.75in
Body	2-column, single-spaced
Space between columns	0.25in
Line spacing (leading)	11pt
Body font	10pt, Times
Abstract font	10pt, Times
Section heading font	12pt, bold
Subsection heading font	10pt, bold
Caption font	9pt (minimum), bold
References	8pt, no page limit, list all authors' names

Acknowledgements

The authors would like to sincerely thank the Chairs and Reviewers ISCA 2021 for your helpful feedback. We wish you health and safety during the coronavirus pandemic.

References

[1] "The champsim simulator. https://github.com/champsim/champsim."
[2] "Spec cpu2017 home page: www.spec.org/cpu2017."

[3] U. Aydonat and T. S. Abdelrahman, "Hardware support for relaxed concurrency control in transactional memory," in *MICRO*, 2010, pp. 15–26.

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

[5] L. Baugh, N. Neelakantam, and C. Zilles, "Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory," in *ISCA*, 2008, p. 115–126.

[6] 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 *ISCA*, 2008, pp. 127–138.

[7] M. Cai, C. C. Coats, and J. Huang, "Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory," in *ISCA*, 2020.

[8] 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. 46–58, 2008.

[9] I. Coorporation, "Intel 64 and ia-32 architectures optimization reference manual," 2016.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS*, 2009, pp. 157–168.

[11] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *HPCA*, 2016, pp. 77–89.

[12] E. Giles, K. Doshi, and P. Varman, "Hardware transactional persistent memory," in *MEMSYS*, 2018, p. 190–205.

[13] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, "Pisces: a scalable and efficient persistent transactional memory," in *ATC*, 2019, pp. 913–928.

[14] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *MICRO 2019*, 2019, pp. 466–478.

[15] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993, pp. 289–300.

[16] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for ibm system z," in *MICRO*, 2012, pp. 25–36.

[17] S. A. R. Jafri, G. Voskuilen, and T. Vijaykumar, "Wait-n-gotm: improving htm performance by serializing cyclic dependencies," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 521–534, 2013.

[18] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *MICRO*, 2018, pp. 520–532.

[19] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *ISCA*, 2018, pp. 452–465.

[20] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *HPCA*, 2017, pp. 361–372.

[21] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *ASPLOS*, 2020, pp. 335–349.

[22] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[23] 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 *ASPLOS*, 2017.

[24] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, 2014, pp. 216–223.

[25] M. Lupon, G. Magklis, and A. González, "A dynamically adaptable hardware transactional memory," in *MICRO*, 2010, pp. 27–38.

[26] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: log-based transactional memory," in *HPCA*, 2006, pp. 254–265.

[27] T. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *MICRO*, 2018, pp. 507–519.

[28] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems," in *HPCA*, 2018, pp. 336–349.

[29] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA*, 2014.

[30] X. Qian, B. Sahelices, and J. Torrellas, "Omniorder: Directory-based conflict serialization of transactions," in *ISCA*, 2014, pp. 421–432.

[31] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *MICRO*, 2018.

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

[33] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *MICRO*, 2015, pp. 672–685.

[34] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *MICRO*, 2017, pp. 178–190.

[35] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS*, 2011.

[36] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent transactional memory," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 58–61, 2014.

[37] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *MICRO*, 2008, pp. 83–93.

[38] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *SC*, 2013, pp. 1–11.

[39] 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 *MICRO*, 2013, pp. 421–432.