(Almost) Fence-less Persist Ordering

Sara Mahdizadeh Shahri
Pennsylvania State University
State College, USA
smahdizadeh@psu.edu

Seyed Armin Vakil Ghahani
Pennsylvania State University
State College, USA
arminvakil@psu.edu

Aasheesh Kolli
Pennsylvania State University
State College, USA
akolli@psu.edu

***Abstract*—The semantics and implementation of a memory persistency model can significantly impact the performance achieved on persistent memory systems. The only commercially available and widely used x86 persistency model causes significant performance losses by requiring redundant, expensive fence operations for commonly used undo logging programming patterns. In this work, we propose light-weight extensions to the x86 persistency model to provide some ordering guarantees *without* an intervening fence operation. Our extension, Themis, eliminates over 91.7% of the fence operations in undo-logging PM programs and improves average performance by 45.8% while incurring only 1.2% increase in data cache size.**

***Index Terms*—Persistent Memory, Memory Persistency Model**

I. Introduction

Persistent Memory (PM) technologies blur the difference between main memory technologies (e.g. DRAM) and storage technologies (e.g. SSD) [35, 41, 52, 54, 73, 81]. They offer non-volatility, byte-addressability, and performance close to that of DRAM. Owing to these characteristics, PMs are expected to be placed on the memory bus, accessed using processor load and store instructions, and host recoverable data structures (e.g., databases, file systems) [6, 15, 17, 19, 20, 28, 30, 44, 45, 51, 54, 61, 62, 67, 70, 74, 76, 87].

Memory persistency models [67] are extensions to the memory consistency models that enable programmers to express the order in which different stores update persistent memory. These ordering guarantees are necessary to develop any kind of recoverable storage software (e.g., file systems and databases). The hardware is responsible for enforcing this prescribed order, even in the presence of volatile structures like processor caches. In this work, we specifically focus on applications that use the popular undo logging techniques [10, 15, 21, 28, 36, 45, 50] to ensure correct recovery.

The semantics and implementation of a memory persistency model plays a significant role in the overall performance achieved by the applications [51]. The x86 persistency model is the only commercially available and widely used persistency model. In this work, we show that the x86 persistency model results in unnecessary performance losses due to the redundant use of expensive `sfence` instructions.

The x86 persistency model is unique in that it allows programmers to use two paths to persistence. The commonly used *temporal path* where a memory location is updated first in the cache and then written back to persistence through successive writebacks in the different levels of processor caches. The temporal path can be exercised using instructions like cacheline writeback (`clwb`) [40]. The *non-temporal path* is where a memory location is updated using cache-bypassing store instructions and results in the update going directly from the core to the memory controller. The non-temporal path can be exercised using instructions like `movnt` [39].

For updates sent along either path, the current x86 persistency model requires using an intervening `sfence` instruction to ensure that the updates reach persistence in order [16, 21, 27, 32, 39, 45, 48, 51, 57–59, 61, 62, 67, 68]. Correctly ordering updates to persistence is the bedrock on which crash consistency can be achieved. For example, when using undo logging, the log updates must persist before any data updates persist to ensure proper recovery [10]. However, `sfence` instructions are expensive as they preclude coalescing, buffering, and reordering of memory accesses and hamper out-of-order execution due to their ordering semantics [50, 51]. So, the frequent use of `sfence` to ensure crash consistency results in significant performance losses on modern x86 processors.

In this work, we show that for certain kinds of updates, current x86 architectures already implicitly achieve the desired ordering and the use of an explicit `sfence` instruction to enforce the order already achieved is superfluous. Specifically, when one update is sent along the non-temporal (or cache bypassing) path and a subsequent update is sent along the temporal path (or through the caches), it is very likely that the non-temporal update persists before the temporal update. This behaviour is caused by the fact that the update along the non-temporal path skips past the many levels of caches and heads directly to the memory controller where as the update along the temporal path is retained at various cache levels for reuse and has to undergo successive writebacks at the various levels of caches to reach persistence.

Furthermore, we observe that in the widely used undo logging approach to crash consistency, a frequently occurring programming pattern is to require a non-temporal update reach persistence before a subsequent temporal update. This pattern arises because of the requirement that within an undo logging transaction, a log update reach persistence before the corresponding data updates. And, logs are usually updated using the non-temporal path to avoid polluting the caches with logs that will never be read during failure-free execution[9, 11, 26, 45, 62, 63]. In fact, in different undo logging PM workloads we studied (details in §V), more than 91% of the `sfence` instructions used are specifically to

enforce this exact kind of ordering. Additionally, we observed that this specific ordering is also widely employed in the state-of-the-art iDO logging [57]. Hence, we can significantly improve PM application performance if this kind of ordering can be achieved *without* the use of an `sfence`.

In this work, we propose *Themis* [1], an extension to the x86 memory persistency model and implementation that guarantees a non-temporal update persists before all subsequent temporal updates issued on the same core, without requiring an intervening `sfence`. Themis leverages the fact that non-temporal updates likely already persist before subsequent temporal updates (as the non-temporal path is much faster than the temporal path). So, in the common case, the required ordering guarantee is achieved at virtually no cost. However, in the uncommon case where a non-temporal update is slowed down (due to any reason), Themis builds mechanisms to detect such cases and enforce the correct ordering by delaying the writeback of subsequent temporal updates.

With Themis and its fence-less ordering guarantee, we observe performance improvements as high as 87% and an average improvement of 45.8% over the baseline x86 persistency model implementation for a suite of PM workloads. Furthermore, Themis incurs only a 1.2% increase in the size of the data caches (to store extra state) and requires no modifications to the cache coherence or memory consistency model implementations. These light-weight changes are in contrast to other persistency model implementations [20, 51] and make Themis much more adoptable in commercial multi-core processors. In summary, this work:

- Highlights that x86 processors can provide a fence-less ordering guarantee between a non-temporal update to a PM location and a subsequent temporal update at virtually no cost.
- Shows that such ordering primitives can be especially useful for undo-logging based PM programs and that over 91% of `sfence` instructions in them can be eliminated with this primitive.
- Presents the design and implementation of Themis, a light-weight extension to the x86 persistency model that leverages above insights to improve average performance by 45.8%.

II. Background and Motivation

A. Crash Consistency

Crash consistency refers to the guarantee provided by storage software (e.g., file systems and databases) that data is in a consistent and recoverable state even in the presence of system failures [13] like power interruptions, kernel crashes, etc. For example, consider the bank balance transfer transaction in Figure 1 (a), where the objective is to transfer $50 from Alice's account to Bob's account. This transaction deducts $50 from Alice's balance and adds the same amount to Bob's. To ensure that this transaction is correctly executed even in the presence of failures, first Alice's and Bob's account balances have to

[1]Themis is an ancient Greek Titaness personifying divine order [79]

```
1 accounts.lock();
2  alice.bal -= 50;
3  bob.bal += 50;
4 accounts.unlock();
```

(a)

```
1 accounts.lock();
2  log.prepare();
3  log.insert(alice.bal);
4  alice.bal -= 50;
5  log.insert(bob.bal);
6  bob.bal += 50;
7  log.commit();
8 accounts.unlock();
```

(b)

Fig. 1. Bank balance transfer example: (a) w/o crash consistency, (b) w/ crash consistency using undo logging.

be stored in persistent locations. However, storing the balance information in persistent locations is not enough.

If a crash happens after the $50 were deducted from Alice's account but before the amount gets added to Bob's, on recovery, we would notice a net $50 loss in the overall balance of Alice and Bob, i.e., $50 vanishes from the system. In order to avoid such situations, storage systems (like databases and file systems) provide *all-or-nothing* guarantees, i.e., either both the operations on Alice's and Bob's balances happen or neither will. Developers can use these all-or-nothing guarantees to make their application crash consistent. These all-or-nothing guarantees can be provided at different granularity like transactions [15, 50, 56, 76], failure-atomic sections (FASEs) [7, 10, 33, 42, 57], or synchronization-free regions (SFRs) [28, 49], each bringing its own sets of trade-offs [48]. Finally, there are several techniques like undo/redo logging [3, 10, 14, 15, 21, 27, 28, 33, 36, 42, 45, 50, 51, 60, 76, 77], iDO [57], shadow paging [17, 56, 66], checkpointing [23, 47, 68], and custom data structures [5, 12, 24, 84, 85] that developers can use to achieve these all-or-nothing guarantees and hence crash consistency. In this paper, we are going to limit ourselves to the popular undo logging approaches to crash consistency.

Undo logging [10, 15, 21, 28, 36, 45, 50] is a crash consistency technique that provides all-or-nothing guarantees by undo-ing (or rolling back) changes from an aborted transaction. To be able to roll back changes, undo logging systems create an undo log entry prior to every update performed within the transaction. The undo log entry contains the current value of the memory location/variable that is being updated. Once the log entry has been created and persisted, only then is the actual memory location/variable updated. If a transaction succeeds, all the memory locations modified within the transaction are persisted and then a commit message is atomically persisted to the log to invalidate the log entries belonging to the transaction. If a transaction fails, the recovery process uses all valid log entries to roll back partial changes from that transaction.

For example, Figure 1 (b) shows how to make the balance transfer transaction crash consistent. At the start of the transaction a new log area is prepared (Line 2) for the transaction. Then before Alice's and Bob's balances are updated (Lines 4 and 6 respectively), the old values are inserted into the log (Lines 3 and 5 respectively). Finally, after both the updates are done only then is the log committed, essentially invalidating the log entries. So, if a crash occurs in the middle of the

transaction, the recovery process can use the log entries to roll back the changes from the aborted transaction, guaranteeing all-or-nothing semantics. Note that updating the logs with current balances before updating the actual balance is the most critical requirement of undo logging.

B. Memory Persistency

Persistent Memory (PM) technologies straddle the divide between main memory technologies (e.g. DRAM) and storage technologies (e.g. SSD) [35, 41, 52, 54, 81]. They offer non-volatility, byte-addressability, and performance close to that of DRAM while also providing a load-store interface to storage. PMs are expected to host recoverable data structures (e.g., databases, file systems) [6, 15, 17, 19, 20, 28, 30, 44, 45, 51, 54, 61, 62, 67, 70, 74, 76, 87]. When used to store recoverable data structures, PM systems are expected to provide guarantees on the order in which stores reach the persistent domain, in other words a *memory persistency model* [67]. Persistency models provide primitives that programmers can use to communicate the desired order of persists to the hardware. It is the responsibility of the hardware to ensure that the specified order of persists is enforced. In this work, we focus on the widely used x86 memory persistency model.

C. x86 Persistency Model

Studying and improving the x86 persistency model is particularly important as it is the only persistency model that is commercially available and widely used. The x86 persistency model is also unique in that it provides two paths to persistence: (i) a *temporal path* via the regular cache hierarchy and (ii) a *non-temporal path* or a cache-bypassing path directly from the core to the memory controller. The ISA provides mechanisms to order persists sent along either paths. Next, we provide more details about the persistence domain used and how to achieve the desired order of persists in x86.

Persistence domain: x86 persistency model requires Asynchronous DRAM Refresh (ADR) support [37, 38, 40], essentially making the memory controllers persistent apart from the PM devices. So, an update is considered persistent as soon as it reaches a memory controller.

Temporal → temporal ordering: To order two PM updates (say for location A and then B) while both updates are sent along the temporal path (using regular store (`st`) instructions), the code sequence required is:

```
st A; clwb A; sfence; st B;
```

The CacheLine WriteBack (`clwb`) [40] instruction writes back the cacheline with location A back to memory controller (and hence the persistence domain) and the subsequent `sfence` ensures that the writeback operation is complete before the store to B is performed at the L1 data cache. This sequence ensures that temporal updates to A and B persist in that order.

Non-temporal → non-temporal ordering: To order two PM updates (say for location A and then B) while both updates are sent along the non-temporal path (using non-temporal or cache-bypassing store (`nt-st`) instructions), the code sequence required is:

```
nt-st A; sfence; nt-st B;
```

The `nt-st` instruction causes the updates to bypass the caches and go directly to the memory controller using the *Write-Combining Buffer* (WCB) at the processor core. Since the updates never go to the caches, we do not need a separate `clwb` instruction here. The `sfence` instruction ensures that the updates to A and B reach the memory controller in order.

Non-temporal → temporal ordering: To order two PM updates (say for location A and then B) while the update to A is sent along the non-temporal path while the update to B is sent along the temporal path, the code sequence required is:

```
nt-st A; sfence; st B;
```

The combination of `nt-st` and `sfence` instructions causes the update to A to reach the memory controller before the update to B even reaches the L1 data cache, thus ensuring that updates to A and B reach persistence in order. For example, in Figure 1(b), the log insert for Alice's balance (Line 3) must persist before the actual update to Alice's balance (Line 4). And, usually log updates are done using non-temporal or cache bypassing stores as these logs are only necessary for crash consistency and never read in failure-free execution (the most common case). While the actual data updates are done using temporal stores as there is expected to be some reuse. In fact, this ordering requirement is frequently used as we see in undo logging based systems [4, 9, 11, 26, 45, 62, 63].

Temporal → non-temporal ordering: To order two PM updates (say for location A and then B) while the update to A is sent along the temporal path while the update to B is sent along the non-temporal path, the code sequence required is:

```
st A; clwb A; sfence; nt-st B;
```

The combination of `clwb` and `sfence` instructions causes the update to A to reach the memory controller before the update to B ensuring that they persist in order. This combination of ordering from temporal to non-temporal stores are also quite common in logging systems though not as common as the non-temporal to temporal ordering. For example, in Figure 1 (b), updates to Alice's and Bob's balances (lines 5 and 8) must persist before the log is committed (line 9).

It is typical in crash-consistent software systems running on x86 machines and relying on logging for crash consistency, to use a combination of temporal and non-temporal store instructions to achieve both necessary ordering guarantees while also judiciously using available cache resources for good performance [11, 26]. As far as we are aware, the x86 ISA is the only ISA that provides these two paths to achieve persistence and next we are going to detail the problem with this approach and how we plan to address it.

D. The problem

While ordering persists is necessary for recovery correctness, enforcing the precise order of persists comes at a steep

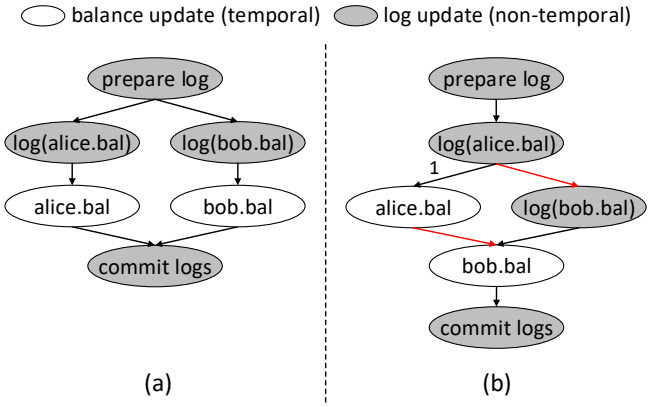

Fig. 2. Relative order of undo logs and the actual balance updates for the bank balance transfer: (a) In the ideal case we only enforce order between undo log and its corresponding write data. (b) Using fence as an ordering semantics leads to a more conservative ordering. Black arrows represent necessary dependencies and red arrows the unnecessary dependencies.

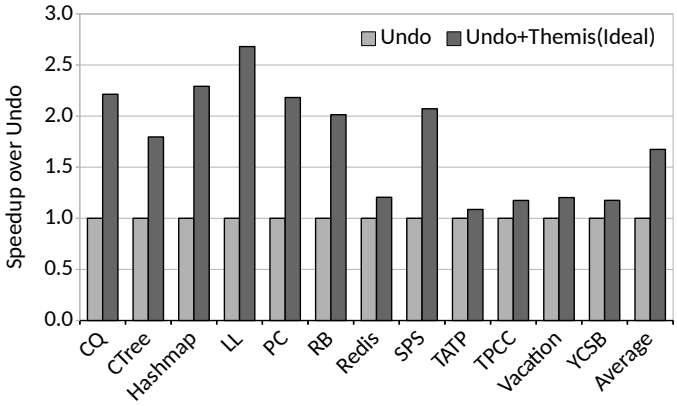

Fig. 3. Ideal Undo logging performance improvement by removing `sfence`

performance cost. This performance loss is due to two reasons: (i) reduced opportunities to buffer, coalesce, and reorder updates to PM and (ii) pipeline stalls caused by the use of fence instructions to express ordering dependencies. In § II-C, we showed that with the x86 persistency model, using an `sfence` instruction is necessary for expressing any kind of persist ordering. However, fence instructions in general are not optimal to express ordering relationships as they enforce ordering at a coarse granularity of *everything before the fence to everything after the fence* [55]. Not all instructions after the `sfence` are dependent on all instructions before the `sfence`. Independent instructions after the `sfence` are unnecessarily delayed. Furthermore, instructions after the `sfence` are not allowed to retire until all operations before the `sfence` are retired resulting in performance loss due to pipeline delays.

For example, consider the the persist ordering constraints for the balance transfer transaction shown in Figure 2. The figure shows the ordering constraints between updates to the log and to the actual balances as part of the transactions. The log operations are shown in different color (gray) from the balances. In the ideal transaction, Figure 2(a), there is only a pair-wise dependency between a balance update and its corresponding log update. However, the x86 implementation of the transaction contains some unnecessary dependencies (red arrows in Figure 2(b)). These unnecessary dependencies are caused because the `sfence` instruction used to express ordering between the log and balance updates for Alice (arrow 1 in Figure 2(b)) unnecessarily causes a dependency between the log and data updates of Alice and Bob (red arrows in Figure 2(b)). Furthermore, the unnecessary dependencies introduced increase with the number of log-data update pairs we have in the transaction.

The central problem with the current x86 persistency model is that the `sfence` used to express the ordering within a single log-data update pair causes unnecessary persist dependencies and these unnecessary dependencies only grow with larger transactions. If we had an "ideal" primitive that allows us to express the dependency within a log-data pair without having to use an `sfence` instruction, significant performance gains can be achieved. To estimate these performance gains of this "ideal", we simply remove the `sfence` instructions used for crash-consistency in a baseline, state-of-the-art undo logging implementation [28] and execute a range of PM workloads on a real x86 processor. While removing the `sfence` may violate crash consistency requirements, it allows us to estimate the performance of our ideal primitive. Figure 3 illustrates the potential improvement in performance with our "ideal" ordering primitive. The improvement in performance ranges from 1.09x upto 2.68x, with an average improvement of 1.67x. These performance improvements are a result of eliminating 91.7% of `sfence` instructions on average.

E. Key insight

Our goal is to enforce the ordering within a log-data pair without using an `sfence` instruction. We observed that the log update is usually done with a non-temporal store instruction while the data update is done with a temporal store instruction. A non-temporal store heads directly to the memory controller and hence persistence. Where as a temporal store is usually retained at multiple cache levels for reuse and needs to be written back or evicted from multiple caches in the cache hierarchy to reach the PM. So, even if the non-temporal log update and the temporal data update were to be started simultaneously, on a modern x86 processor, the log update will likely persist before the data update, *even without any intervening* `sfence` *instruction*. This observation implies that, most of the `sfence` instructions used for enforcing ordering within a log-data pair are unnecessary. However, if the `sfence` is not used, it is possible that there might be instances where the non-temporal update gets stuck in a buffer along its path and the temporal update reaches the memory controller first, violating recovery ordering constraints. In this work, we develop Themis, a light-weight x86 hardware extension that ensures that a non-temporal update persists before all subsequent temporal updates without using an `sfence` instruction.

F. Beyond Undo Logging

iDO [57] is a state-of-the-art logging mechanism that takes a different approach to achieving crash consistency. Instead of logging every individual write to PM, iDO performs logging

at a much coarser granularity of *idempotent program regions*. Before the start of a new idempotent region (identified by a compiler pass), all the information necessary to re-execute the idempotent region is logged. So, if an idempotent region's execution is interrupted by a failure, the iDO logs are used to re-execute the region. While iDO incurs fewer logging operations than undo logging (due to a coarser granularity of logging), the amount of data logged in each logging operation is much larger than the data logged per write operation in undo logging. Due to this trade-off, iDO performs better with programs with larger critical sections/transactions, while undo logging performs better in applications with smaller transactions, especially if the transactions are read-heavy.

Themis improvements are not restricted to undo logging implementations. The iDO [57] logging mechanism also uses `sfence` instructions to enforce non-temporal to temporal store orderings. iDO persists the address of the first instruction in idempotent program region (and other logging metadata) through non-temporal store before executing the region. An `sfence` instruction is used to enforce the ordering between this non-temporal write and subsequent writes in the idempotent region. This fence instruction can be eliminated with our ideal ordering primitive. To estimate the performance improvements possible with a fence-less ordering primitive, we again remove the `sfence` instructions required in iDO. We observe a 13.7% increase in average performance for the same set of applications studied earlier in the real system setup.

III. Themis

In this section, we delve into the details of the design, specification, and implementation of Themis' persistency model, an extension to existing x86 persistency models to reduce the use of fence instructions.

A. Design Goals

Themis is based on three key design goals:

Create a non-fence ordering primitive: As discussed in §II-D, fence instructions, while necessary in the current x86 architectures, are a significant source of performance degradation. We need to develop a mechanism that allows programmers to express an ordering dependency without a fence instruction, especially for the common-case ordering scenarios like non-temporal to temporal store ordering.

Exploit multiple paths to persistence in x86: x86 is a unique architecture in that its persistency models allows multiple paths to persistence, the fast non-temporal path and a slow temporal path. We can exploit the differences in speeds of requests sent along the fast vs slow paths to provide implicit ordering guarantees to programmers without relying on a fence instruction. To the best of our knowledge, no prior work has sought to exploit the presence of multiple paths to persistence.

Light-weight hardware changes: Since this work targets widely used x86 architectures, it is important that the hardware changes proposed are light-weight to allow easy adoption. Prior proposals on improved persistency models require invasive changes to the cache hierarchy, caches, cache coherence protocols, implementations, etc.

Such invasive changes imply that these proposals face a long road to adoption we'd like to avoid.

Next, we describe Themis' persistency model and then describe an architectural implementation of the model.

B. Persistency Model

Formally, we express the Themis persistency model as an ordering relation over memory events *stores*, *non-temporal stores*, *clwbs*, and *fences*. The term *persist* refers to the act of durably writing a store to the persistent domain. We assume persists are performed atomically (with respect to failures) at naturally aligned 8-byte granularity. By "thread", we refer to execution contexts—cores or hardware threads. We use the following notation:

- S_a^i: A regular temporal store from thread i to address a
- NS_a^i: A non-temporal store from thread i to address a
- C_a^i: A `clwb`/`clflush(opt)` from thread i to address a
- F^i: A fence (`sfence` or `mfence`) from thread i.

We reason about two ordering relations over memory events, *execution order* and *persist memory order*. Execution order (EO) is a partial order over memory events that governs the order in which different instructions get executed in an x86 processor. For instructions within the same thread, the program order (the order in which instructions are retired) of the instructions decides the execution order. For instructions across threads, the x86 architecture's memory consistency model (TSO [69]) and its cache coherence implementation determines the order in which the instructions get executed. Persist memory order (PMO) is the order of memory events as governed by the Themis' memory persistency model [67].

We denote these ordering relations as:

- $A \leq_e B$: A occurs no later than B in EO
- $A \leq_p B$: A occurs no later than B in PMO

An ordering relation between stores in PMO implies the corresponding persist actions are ordered; that is,

$$A \leq_p B \rightarrow B \text{ may not persist before } A.$$

We next describe the semantics of Themis' persistency model by describing how to ensure that two different store instructions (of any kind) in the same or different threads will persist in the desired order.

Temporal $\rightarrow$ temporal: Persisting two temporal stores in order requires an intervening cacheline writeback/flush and sfence/mfence combination in the execution order. Formally:

$$S_a^i \leq_e C_a^j \leq_e F^j \leq_e S_b^k \rightarrow S_a^i \leq_p S_b^k \quad (1)$$

Non-temporal $\rightarrow$ non-temporal: Persisting two non-temporal stores in order requires an intervening sfence/mfence in the execution order. Formally:

$$NS_a^i \leq_e F^i \leq_e NS_b^j \rightarrow S_a^i \leq_p NS_b^j \quad (2)$$

Non-temporal $\rightarrow$ temporal: Persisting a non-temporal store and a temporal store in order requires the two stores

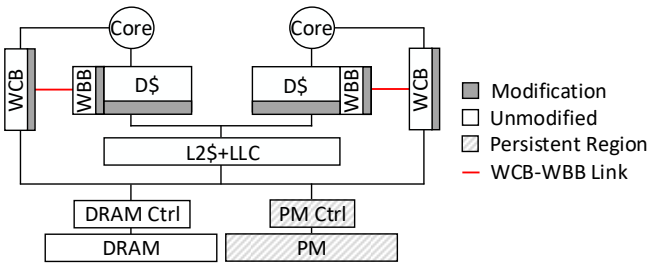

Fig. 4. High-level system architecture with Themis .

to be ordered in the execution order. No intervening fence instruction is necessary, unlike the baseline x86 persistency model. Formally:

$$NS_a^i \leq_e S_b^i \rightarrow NS_a^i \leq_p S_b^i \tag{3}$$

Note that here we provide this fence-less ordering guarantee for stores executed on the same thread (an overwhelmingly common case). If this ordering is required when the two stores are on different threads, an intervening fence is necessary, just in the baseline x86 model.

Temporal → non-temporal: Persisting a temporal store and a non-temporal store in order requires an intervening cacheline writeback/flush and sfence/mfence combination in the execution order. Formally:

$$S_a^i \leq_e C_a^j \leq_e F^j \leq_e NS_b^k \rightarrow S_a^i \leq_p NS_b^k \tag{4}$$

While the above equations formally define Themis' persistency model, these ordering relationships are very similar to the ones described for the baseline x86's persistency model in §II-C with one exception. The only exception is that for non-temporal to temporal ordering, no intervening fence instruction is required, we just have to ensure that the two stores are correctly ordered in the execution order. If the two stores are in the same thread, then we have to ensure that the two stores are ordered within the program order. If the two stores are on different threads, then we need to ensure that there are intervening synchronization accesses (like unlock/lock or release/acquire operations) to order the two stores in the execution order. Next, we describe a high-level design for this persistency model.

C. High-level Design

Figure 4 shows a simplistic multi-core system. Temporal stores go from the core, to the data cache to its writeback buffer (WBB), to the various levels of caches (L2$+LLC) and finally to the corresponding memory controller (based on its physical address). Non-temporal stores go from the core to the write-combining buffer (WCB) where they get reordered, buffered, and coalesced for performance and then directed to the corresponding memory controller.

At a high-level, with Themis, the goal is to ensure that when a non-temporal store is executed followed by a temporal store (i.e., `nt-st` A $\leq_e$ `st` B), A persists before B, even when there are no intervening fences (i.e., Eq 3 in §III-B). We achieve this by making light-weight modifications to the data cache, its WBB, and the core's WCB. The steps involved in this process are as follows:

(1) When `nt-st` A gets executed, it is first placed in the WCB and then eventually drained to the memory controller.

(2) When `st` B comes along, we first check the WCB to see what outstanding non-temporal stores are yet to be drained by noting down the WCB's tail pointer.

(3) When the cacheline with B is eventually written back either due to a regular replacement or due to a `clwb`-like instruction, we place the cacheline and its associated WCB tail pointer in the data cache's WBB.

(4) Before we let the cacheline B leave the WBB, we check if all outstanding non-temporal stores at the time `st` B reached the data cache have been drained from the WCB (by using the tail pointer stored along with cacheline B). If all prior non-temporal stores (including one to A) have been drained, we let the writeback of cacheline B continue. If not, then we wait for WCB to drain its outstanding non-temporal stores and only then we let cacheline B to be written back to L2.

By not allowing the update to B to even leave the data cache's WBB before all prior executed non-temporal stores (include the one to A), we ensure that A persists before B (i.e., `nt-st` A $\leq_p$ `st` B). Our approach is conservative. Ideally, we only have to ensure that A persists only before B is written back from the LLC (and hence reaching the persistence domain). However, we enforce that A persists before B is written back from the data cache, i.e., much earlier than necessary. As we will show in the following sections, this design choice considerably simplifies Themis' architecture with almost no performance penalties.

IV. Implementation

In this section, we detail how we extend the data cache, WBB, and WCB to implement Themis' persistency model.

A. Write Combining Buffer

The WCB is usually implemented as circular buffer of about 16 entries. Each entry typically has the space to buffer a cacheline of data (64B) and the associated physical address. As shown in Figure 5, apart from the entries with data and addresses, the WCB consists of three important pointers: (i) **tail (t):** points to the next available entry for an incoming non-temporal store request (ii) **head (h):** points to the oldest outstanding entry that needs to be drained to the memory controller (iii) **headACK (hAck):** points to the oldest entry that has not been drained to the memory controller, or has been drained but has not yet received an acknowledgement from the memory controller saying that the entry has been persisted. A single bit per entry tracks if a drained entry has been ACKed by the memory controller or not and this bit is used to increment headACK.

The aspects of WCB described above already exist in a modern x86 processor and Figure 5 shows how they help the WCB operate:

(i) **Incoming non-temporal store** ① gets allocated at the tail or gets coalesced into an active entry. If it gets allocated at the tail, the tail is incremented ②, always pointing to the next

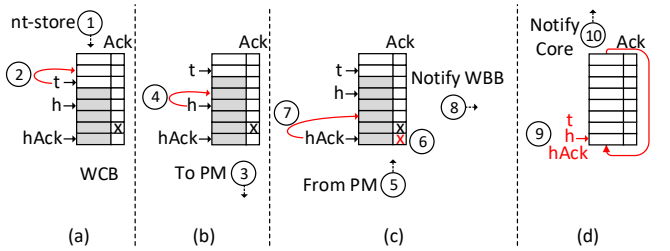

Fig. 5. Major events in WCB: (a) incoming non-temporal store, (b) WCB entry is drained, (c) ACK received from PM controller, (d) an overflow of tail pointer occurs.

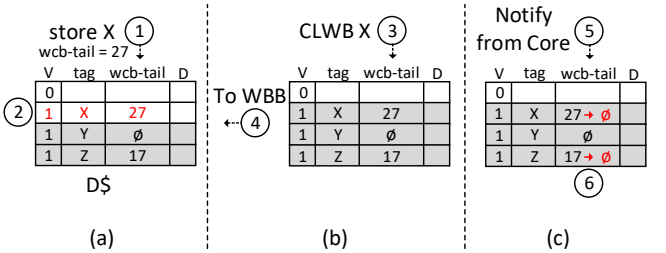

Fig. 6. Important events in L1 data cache with Themis: (a) incoming temporal store, (b) evicting a dirty cacheline, (c) receiving an overflow notification from core.

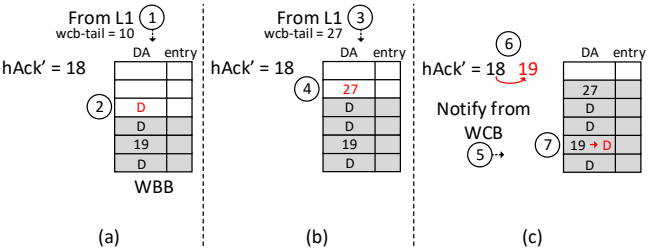

Fig. 7. Important events in WBB with Themis: (a) Incoming writeback request with tail pointer greater than hAck', (b) Incoming writeback request with tail pointer lower or equal to hAck', (c) Receiving headACK update from WCB.

available entry for incoming requests. If no space is available in the WCB, the stall is back propagated to the core.

(ii) **On draining an active entry** ③ to the PM controller, the head gets incremented ④, always pointing to the next entry to be drained. If the memory controller is full, a stall message is back propagated to the WCB.

(iii) **On receiving a PM controller ACK** ⑤ signifying that a particular drained entry has reached the PM controller (and hence persisted), the ACK bit in the associated entry is set ⑥. If the entry pointed to by headACK has been ACKed, the headACK is incremented ⑦ to point to the next oldest entry. This headACK increment is also communicated to the WBB ⑧, so that the WBB has up-to-date information on which entries in the WCB have been persisted.

(iv) **Overflow** is the corner-case situation when the tail pointer overflows. We use 6-bit tail, head, and headACK pointers even while using a 16-entry WCB. The least significant 4 bits are used to index into the WCB. The most significant 2 bits are used to track circular buffer "wrap arounds", starting from 00 and incrementing by 1 every time a wrap around happens. Once the WCB has been through four full uses, we force drain all the outstanding requests and wait for ACKs for all of them from the memory controllers. Once the WCB has been completely drained and ACK, i.e., it is empty, we reset the head, tail, and headACK pointers ⑨ and notify the core ⑩. As we will see next, the core will notify the data cache about the overflow and this information is used to correctly enforce persist orders. In § V-E, we show how we arrived at using 6-bit pointers.

B. L1 Data Cache

As shown in Figure 6, apart from the the usual valid (V), tag, and data (D) fields, we extend each cacheline to store a 6-bit tail pointer value corresponding to the state of the WCB when a particular cacheline was written. Figure 6 shows how the data cache functions with Themis:

(i) **Incoming temporal store** ① request is tagged with the latest WCB tail pointer by the core. And this WCB tail is stored along with tag and data in the cacheline ②. The goal is to make sure that this cacheline persists only after associated WCB entries have been persisted (as indicated by the tail pointer). We achieve this goal by making sure that this cacheline does not even leave the writeback buffer (WBB) of the data cache before the corresponding WCB entry persists.

(ii) **While evicting a dirty cacheline** either due to a `clwb` request ③ or due to regular cacheline replacement, along with the tag and the data, the associated WCB tail pointer is also sent to the the WBB ④.

(iii) **On an overflow notification** ⑤ from the core, we know that all outstanding WCB entries have been drained and persisted, so none of the cachelines have any unresolved dependencies with prior non-temporal stores. Hence, all the WCB tail pointers in the cache are nullified ⑥.

C. Writeback Buffer

As shown in Figure 7, we extend each entry to store a 6-bit "Drainable After" (DA) pointer, pointing to the entry in the WCB after which this entry can be drained from the WBB to the L2 cache. The WBB also maintains a copy of the WCB's headACK pointer (hAck'), allowing it to maintain visibility into which non-temporal stores have been persisted and which ones haven't. Figure 7 shows how writeback buffer functions with Themis:

(i) **On an incoming writeback request** from the data cache ①, we compare the tail pointer associated with the incoming request with hAck'. If the tail pointer is lower or equal to hAck', we can infer that the non-temporal requests that this writeback request is dependent upon have already been persisted and we can mark this writeback request as "drainable" (D) ② and evict it to L2 at any time. If the tail pointer is higher than hAck' ③, then the tail pointer is stored along with the request in the WBB ④.

(ii) **On a notification from WCB about headACK increments** ⑤, WBB first increments its own copy of headACK (hAck') ⑥. Then it traverses active WBB entries and marks all those entries drainable whose tail pointers are smaller than or equal to the new hAck' ⑦.

This sequence of operations in the WCB, data cache, and the WBB ensure that all temporal stores will only reach the L2 when all prior executed non-temporal store instructions have been persisted.

D. Discussion

Next, we discuss some interesting corner cases and design decisions that we made with Themis.

Coherence and consistency. Prior persistency model implementations require invasive changes to the cache coherence and consistency model implementations [51]. However, Themis does not require *any* changes to the existing coherence and consistency model implementations and in fact is completely orthogonal to them. All the information necessary to maintain the correct ordering is maintained only locally at one core's data cache, WBB, and WCB. Cacheline invalidations and evictions that are caused by cache coherence transactions are treated just like any regular replacement policy related invalidations and evictions. By only maintaining local order information, Themis proposes a design that can be easily adopted into existing multi-core architectures.

Volatile vs persistent memory updates. Even in PM systems, there is expected to be some significant amount of DRAM used and there are going to be some program variables that do not need to be persistent (e.g., stack variables) and are simply maintained in DRAM in volatile memory locations. Since volatile memory updates do not update persistent state directly, we do not always need to enforce the non-temporal to temporal store ordering constraints for them. So, we allow volatile cachelines (those backed by DRAM) to be evicted out of the data cache without any restrictions for improved performance. However, persist dependencies for stores executed across cores may still arise due to intervening volatile memory accesses [51] and for these cross-core persist dependencies, programmers have to rely on fence-based ordering primitives that they would normally use in existing x86 processors.

Enforcing non-temporal to temporal store ordering at L1. Technically, we only have to ensure that a non-temporal store persists before a subsequent temporal store is written back from the LLC (and hence reaches the persistence domain). However, we enforce that the non-temporal store persists even before the temporal store is written back from the data cache, i.e., much earlier than necessary. This conservative enforcement provides three important benefits:
(i) **reduced storage overheads:** If ordering is enforced at the LLC, then all the cachelines in the LLC (and other intermediate caches) will need to maintain a WCB tail pointer. Since, LLCs are typically about 32x [34, 78] larger than L1 data caches, this approach would increase storage overheads by about 32x.
(ii) **reduced design complexity:** With the current Themis architecture, all ordering requirements for a thread are enforced at the private L1 cache of the thread where we can be sure that memory accesses are generated only by the thread currently executing at that core. If ordering enforcement were to be moved to shared caches (like the LLC) then the same cacheline could be modified by different threads on different cores simultaneously and we need to maintain ordering information on a per-core basis for each cacheline, significantly complicating the design.
(iii) **reduced implementation complexity:** Even if the ordering information were to be correctly maintained, then while evicting a shared cacheline from the LLC, we would have to make sure all of its dependencies have been resolved at all the cores, requiring the LLC controller to constantly communicate with all the per-core WCBs in a multi-core system. This communication will be complex to implement, especially in multi-socket machines.
These three important benefits make Themis significantly easier to adopt, however, they come with a performance trade-off. As we will show in § V-D, this performance loss is negligible as most non-temporal stores persist before subsequent temporal stores get evicted from the data cache.

TABLE I
SIMULATION CONFIGURATION.

Core	4-cores, 3GHz OoO 8-wide Dispatch, 8-wide Commit, 192-entry ROB, 32/32-entry Load/Store Queue
I-Cache	32KB, 4-way, 64B 1ns hit latency, 8 MSHRs
D-Cache	64KB, 4-way, 64B 2ns hit latency, 8 MSHRs 16-entry WBB
Write-combining Buffer (WCB)	16 entries 20ns delay to PM controller
LLC	2MB per core, 16-way, 64B 20ns hit latency, 32 MSHRs
Memory controller (DRAM, PM)	128/64-entry write/read queue
DRAM	DDR4, 1200MHz
PCM	1200MHz, 346ns read latency 500ns write latency [43]

Comparison to other relaxed persistency models. In this work, we propose Themis as an extension to the x86 persistency model, the only model that provides two paths to persistence, a temporal and a non-temporal one. ARM has recently introduced non-temporal store instructions [1], so, we expect that in the future Themis will be applicable to ARM processors as well. Prior persistency models such as [17, 44, 51, 62, 67] provide only one path to persistence. All of those persistency models cannot facilitate Themis-like reduction in the number of `sfence` instructions used. While strand persistency models can potentially provide similar reduction in unnecessary ordering enforced as Themis, it comes with a radically different memory persistency model that will necessarily require invasive changes to the cache coherence protocol, memory consistency model, and data cache implementations [29].

V. EVALUATION

A. Methodology

Simulation infrastructure. We evaluate Themis using the gem5 architectural simulator [8] in full-system mode. We model a four-core system with the ARMv8 ISA [1] and the key architectural parameters of the evaluated system are summarized in Table I. Note that we used ARM cores as a starting point for our evaluation due to the stability and maturity of the ARM core implementations in gem5 [25]. Even though we use ARM cores, we implemented the entire x86 memory

TABLE II
WORKLOAD DESCRIPTIONS. WORKLOADS WITH AN * WERE ONLY RUN ON REAL HARDWARE IN §II.

Workload	Description
CQ	En/Dequeue nodes in concurrent queue
CTree	Insert/delete nodes in crit-bit tree
Hashmap	Insert/delete nodes in hashmap
LL	Insert/delete nodes in linked-list
PC	Update in hash-table
Redis*	Object-based KV store workload, lru-test, 1M keys [62]
RB	Insert/delete nodes in a red-black tree
SPS	Swap random entries of an array
TATP	`update_location` transaction [64]
TPCC	`add_new_order` transaction [72]
Vacation*	RB + LL KV store workload, 4 clients, 2M transactions [62]
YCSB*	BTree KV store database, 4 clients 8M transactions, 80% writes [62]

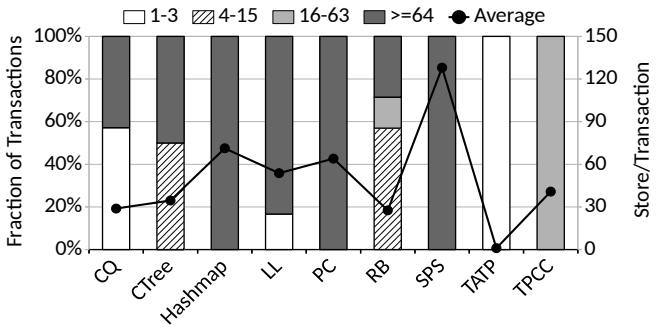

Fig. 8. Number of stores per transaction distribution

persistency model (with `clwb`, `sfence`, and non-temporal store instructions). This approach is similar to the one used by Kolli et al. [51] and we firmly believe that our implementations are representative of the behaviors of modern x86 processors. All gem5 and real system experiments are executed on c6420 Xeon servers on CloudLab [22].

Workloads. We assess Themis with several PM-centric multi-threaded workloads [48, 51, 62, 64, 72] listed in Table II. These workloads vary from those that modify a single data structure, like CTree [62] to database workloads like TPCC [72]. Each workload executes transactions of various different sizes, i.e., different number of log-data update pairs, we will simply refer to the size of a transactions as the number of stores per transaction. For example, the bank balance transfer transaction from Figure 1 has two stores in the transaction for the balance updates, one temporal store each. Figure 8 shows the distribution of size of transactions seen in each of the workloads. The smallest transactions have only one store within them (e.g., TATP's update location transaction [64]) while the larger transactions can have over hundred stores within them (e.g., SPS). These workloads provide a range of average transaction sizes and also a varied distribution in the sizes of individual transactions. Note that "stores within a transaction" refers to only the temporal stores, the non-temporal stores required for logging are not considered as part of the original transaction.

The logging technique we use to make these transactions crash consistent is built upon the state-of-the-art "coupled undo logging" approach introduced by Gogte et al. [28]. We use

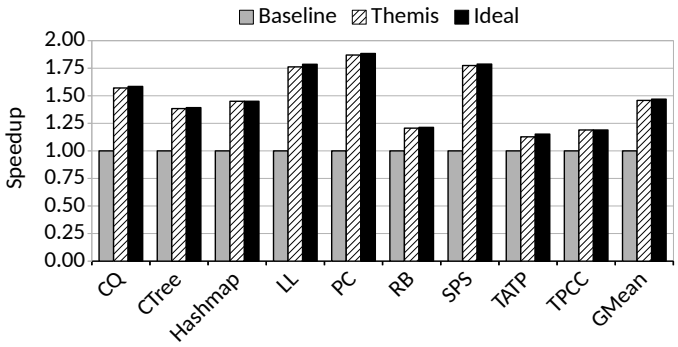

Fig. 9. Speedup for Baseline, Themis, and Ideal.

their code as the starting point and improve the performance of the transactions by moving from using a linked-list-based log to an array-based log. This optimization not only reduces the time required to index into a specific portion of the log, it also reduces the number of stores necessary to add new entries into the log. With list we had to perform two operations one to create a new log entry and the second to splice the new entry onto the list. With an array based structure, we simply have to create a new log entry. We also observed that the code made a number of redundant function calls some of which we were able to completely elide and others we accelerated through function inlining. Furthermore, we also make use of non-temporal store instructions to reduce cache pollution from logging updates. These simple software engineering optimization techniques allowed us to improve the performance of transactions by 2.63x on average. We use these optimized transactions as our baseline and improve their performance with Themis.

Designs evaluated. In this work, we evaluate three different designs. (i) **Baseline** refers to the scenario where the workloads are run with our optimized logging techniques and the baseline x86 persistency model, so, this design requires the use of `sfence` instructions to ensure non-temporal to temporal store ordering. (ii) **Themis** refers to the scenario where the workloads are run on a processor with our Themis persistency model implementation. This design does not require a `sfence` instruction to ensure non-temporal to temporal store ordering. (iii) **Ideal** is the baseline system with no `sfence` instructions for non-temporal to temporal store ordering. This design is not crash consistent, but represents the upper bound on performance that can be achieved by eliminating `sfence` instructions.

Finally, in this evaluation section, we'd like to answer the following questions:

- How does Themis perform? (§V-B)
- What factors impact workload performance? (§V-C)
- What are the implications of Themis' restrictions on writebacks on overall performance? (§V-D)
- What are the hardware overheads of Themis? (§V-E)
- How are differently sized transactions impacted? (§V-F)

B. Performance Comparison

Figure 9 contrasts the speedup of Themis and Ideal over Baseline. The key takeaways from this analysis are:

Themis consistently outperforms Baseline: Themis outperforms Baseline in all benchmarks as it eliminates the use

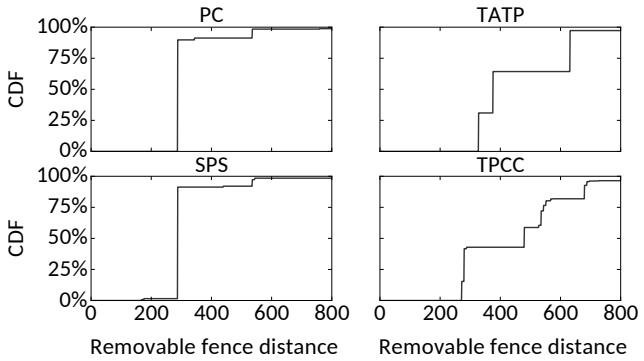

Fig. 10. Number of instructions between two consecutive fences that are removed by Themis.

of expensive `sfence` instructions to enforce non-temporal to temporal store ordering. Themis improves performance by as much as 87% (for PC) and by 45.8% on average across all workloads.

Themis achieves near-ideal performance: Themis almost completely bridges the gap between baseline and ideal performance. This level of performance shows that the restrictions imposed by Themis on the writeback operations at the L1 data cache results in barely any performance degradation.

Best performing workloads: SPS, PC, and LL are the three best performing workloads with performance improvements ranging from 77.4% to 87%.

Worst performing workloads: RB, TATP, and TPCC are the three worst performing workloads with performance improvements ranging from 12.8% to 20.6%.

C. Performance analysis

To better understand the reason behind why some workloads perform better than others, we analyzed the frequency with which `sfence` instructions appear in the baseline that can be removed by using Themis. The higher the frequency with which removable fence instructions appear in a program (i.e., lower inter fence distance), the larger these fences contribute to overall execution time and removing them yields larger performance gains. It is important to note that the absolute number of fence instructions removed or the fraction of fence instructions removed matters less than the frequency with which these fence instructions appear in the program. Larger the frequency or lower the distance between two consecutive removable fence instructions, the higher is the potential for performance improvement.

Figure 10 plots the cumulative distribution function (CDF) of distance between two consecutive removable fence instructions (measured as the number of intervening instructions) for the two best performing (SPS and PC) and worst performing (TATP and TPCC) workloads. As expected, the best performing workloads see that an overwhelming majority of consecutive removable fence instructions have only about 200 instructions between them. Where as for the lower performing TATP and TPCC workloads the distance between removable fence instructions skews much higher.

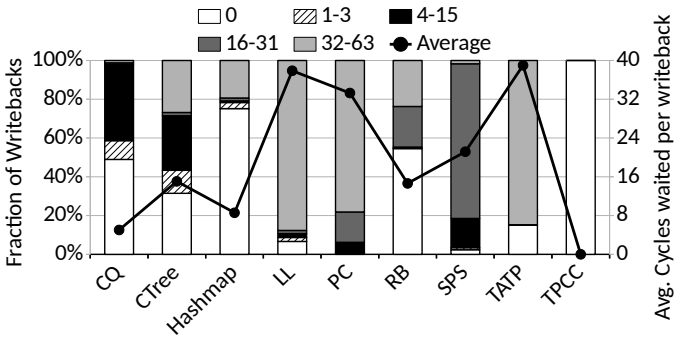

Fig. 11. Distribution of number of cycles a writeback waits in WBB because of Themis' restrictions.

D. Implications of Themis restrictions

Themis achieves the desired persist ordering guarantees by placing restrictions on when a writeback may be drained from WBB in L1 data cache. The larger the wait times, the WBBs get full and the delays are back propagated to the core and show up as pipeline stalls. Note that not all delays result in performance degradation due the latency hiding powers of modern out-of-order processors. However, excessive delays do end up hurting performance. As shown in Figure 9, Themis' near-ideal performance indicates that the restrictions are minimal.

Figure 11 shows the effects of these restrictions in more detail. This figure shows the distribution of a writeback requests categorized by the number of cycles they are delayed. A majority of writebacks see no delays, where as some writebacks see few tens of cycles of delays. Writebacks in LL and TATP see the longest delays (about 38 cycles), while CQ and TPCC see the shortest delays. These delay numbers highlight that Themis places minimal restrictions on cache writebacks.

E. Hardware overheads

The main hardware overhead of Themis arises because of the WCB tail pointers that are stored along with each cacheline in the data caches. So, the size of the tail pointer ends up determining the Themis' storage overhead. As the tail pointer width increases, there will be fewer overflows (as discussed in § IV-A) and lesser performance degradation. However, longer tail pointers result in larger storage overheads. We performed a sensitivity analysis with increasing hardware overheads: 0.5KB (4-bit tail pointer), 0.75KB (6-bit), 1KB (8-bit), and 1.25KB (10-bit). And found very little performance variation among all configurations. We use 6-bit tail pointer, incurring 0.75KB (or 1.2%) of hardware overhead per data cache.

F. Sensitivity to transaction sizes

Finally, we see how the performance of Themis changes with different transaction sizes. Figure 12 shows the speedup achieves with Themis over baseline for different transaction sizes. We change the workloads to artificially set the number of stores per transaction in each of the workloads and see how Themis performs with increasing transaction sizes.

As expected, with increasing transaction sizes, the frequency of removable fence instructions (those required to enforce non-temporal to temporal store orderings) increase and we observe

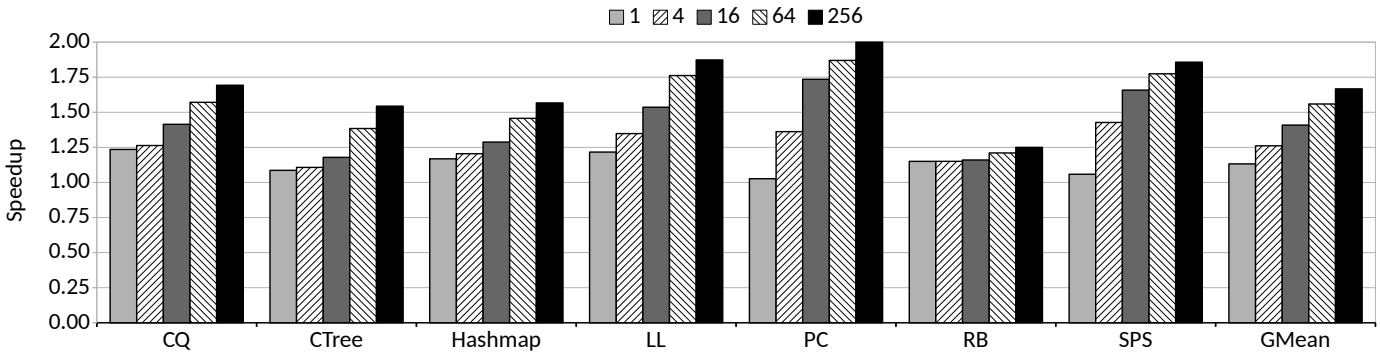

Fig. 12. Speedup for different number of stores per transaction (Geo-Mean)

an increase in speedup with Themis. We observe this pattern in all workloads, however, different workloads exhibit different rates of performance improvements. Note that we omit TATP and TPCC from this experiment as these changes violate the definitions of those workloads.

VI. Related work

In this section we discuss the prior work on facilitating crash consistency solutions to PM systems.

A. Software-based Solutions

Runtimes. A wide range of works have been proposed optimizing file systems for PM in different levels of system stack [17, 21, 46, 53, 75, 80, 82, 83, 86]. Atlas [10] is a compiler and runtime solution for providing undo logging that ensures crash consistency at the granularity of outer-most lock. Gogte et al. [28] proposes an undo logging approach at the granularity of Synchronization Free Regions (SFRs) and addresses drawbacks of providing failure atomicity at the granularity outer-most lock, as Atlas [10] offers. Kolli et al. [50] provides an efficient transaction implementation to reduce the constraints on the order of persistent updates. JUSTDO logging [42] makes sure that the program's execution resumes immediately where the application is interrupted in the critical section by persisting the architectural state prior to the critical section's execution in a system. iDO [57] extends JUSTDO logging to finer grained idempotent program regions.

Libraries. NV-Heaps [15] and Mnemosyne [76] expose interface for applications to build and modify persistent objects that guarantee failure atomicity with granularity of transaction using write-ahead logging. SoftWrAP [27], Rewind [11], and DUDETM [56] employ different logging techniques to provide applications efficient crash-consistent transactions.

All software approaches rely on the underlying memory persistency model and can not provide the same ordering guarantees without the use of `sfence` instruction.

B. Hardware-based Solutions

Multi-versioning. Kiln [87] employs persistent LLC alongside PM to enable in-place updates with no logging. ATOM [45] relies on hardware structures to provide atomic durability using undo logging. Proteus [70] proposes a logging mechanism to enable persisting the transactions by applying drastic modification to the core to manage the logs and order them with respect to the write data. Gupta et al. [31] leverages persistent memory controllers to accumulate the updates in a transaction until they are commited to PM. ThyNVM [68] and PiCL [65] propose hardware-based checkpointing mechanisms to achieve crash consistency.

Ordering. Prior work [2, 40, 67] propose memory persistency model to determine the necessary orderings in which persistent updates need to reach to PM. BPFS [17] provides atomicity and ordering via epoch barriers, allowing reordering of persistent updates only inside of an epoch. Doshi et al. [20], Kolli et al. [51], Nalli et al. [62] and Shin et al. [71] explore different approaches for implementing the epoch persistency model in hardware. Lu et al. [61] proposes loose-ordering consistency to relax the constraints among persistent writes and employ hardware support to resolve the conflicts of transitions that should be persisted. Dananjaya et al. [18] strengthens the one-sided barrier semantics in ARP [48] and proposes release persistency model(RP) to guarantee that any pair of stores are persisted in the same order as the consistency model enforces the ordering. Gogte et al. [29] propose an implementation for strand persistency model to reduce the ordering overheads.

Unlike these set of approaches, Themis requires no invasive changes to the cache hierarchy.

VII. Conclusion

The x86 memory persistency model offers two non-overlapping paths to persistence. And it only provides ordering guarantees using expensive fence instructions across or along these paths. In this work, we propose Themis, a light-weight x86 persistency model extension to provide some implcicit ordering guarantees without using an intervening fence instruction. We improve the average performance of undo-logging based PM workloads by 45.8% while incurring as hardware overhead only 1.2% increase in data cache size.

Acknowledgments

We thank Vaibhav Gogte, Hongjune Kim and Changhee Jung for giving us access to their code and troubleshooting us. We would also like to thank the anonymous reviewers for their valuable feedback.

References

[1] ARM. (2015) Programmer's guide for armv8-a. [Online]. Available: https://static.docs.arm.com/den0024/a/DEN0024A_v8_architecture_PG.pdf

[2] ——. (2016) Armv8-a architecture evolution. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-evolution

[3] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 707–722. [Online]. Available: https://doi.org/10.1145/2723372.2749441

[4] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 263–276. [Online]. Available: https://doi.org/10.1145/2872362.2872377

[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*, ser. INFLOW '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2527792.2527799

[6] M. Bakhshalipour, A. Faraji, S. A. V. Ghahani, F. Samandi, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Reducing writebacks through in-cache displacement," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 2, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3289187

[7] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," *SIGPLAN Not.*, vol. 51, no. 10, p. 677–694, Oct. 2016. [Online]. Available: https://doi.org/10.1145/3022671.2984019

[8] 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*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[9] H.-J. Boehm and D. R. Chakrabarti, "Persistence programming models for non-volatile memory," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 55–67. [Online]. Available: https://doi.org/10.1145/2926697.2926704

[10] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *SIGPLAN Not.*, vol. 49, no. 10, p. 433–452, Oct. 2014. [Online]. Available: https://doi.org/10.1145/2714064.2660224

[11] 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, p. 497–508, Jan. 2015. [Online]. Available: https://doi.org/10.14778/2735479.2735483

[12] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[13] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 228–243. [Online]. Available: https://doi.org/10.1145/2517349.2522726

[14] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From aries to mars: Transaction support for next-generation, solid-state drives," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 197–212. [Online]. Available: https://doi.org/10.1145/2517349.2522724

[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*, ser. ASPLOS XVI, 2011.

[16] N. Cohen, M. Friedman, and J. R. Larus, "Efficient logging in non-volatile memory by exploiting coherency protocols," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133891

[17] 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*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 133–146. [Online]. Available: https://doi.org/10.1145/1629575.1629589

[18] M. Dananjaya, V. Gavrielatos, A. Joshi, and V. Nagarajan, "Lazy release persistency," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1173–1186. [Online]. Available: https://doi.org/10.1145/3373376.3378481

[19] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: A hybrid pram and dram main memory system," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 664–469. [Online]. Available: https://doi.org/10.1145/1629911.1630086

[20] 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)*. IEEE, 2016, pp. 77–89.

[21] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592814

[22] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of cloudlab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1–14.

[23] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan, "Phoenix: Memory speed hpc i/o with nvm," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016, pp. 121–131.

[24] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," *SIGPLAN Not.*, vol. 53, no. 1, p. 28–40, Feb. 2018. [Online]. Available: https://doi.org/10.1145/3200691.3178490

[25] Gem5. (2020) Gem5 status matrix. [Online]. Available: http://www.m5sim.org/Status_Matrix

[26] E. Giles, K. Doshi, and P. Varman, "Bridging the programming gap between persistent and volatile memory using wrap," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2482767.2482806

[27] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[28] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, 2018.

[29] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 652–665.

[30] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Software wear management for persistent memories," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, ser. FAST'19. USA: USENIX Association, 2019, p. 45–63.

[31] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 466–478. [Online]. Available: https://doi.org/10.1145/3352460.3358321

[32] S. Haria, M. D. Hill, and M. M. Swift, "Mod: Minimally ordered durable datastructures for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 775–788. [Online]. Available: https://doi.org/10.1145/3373376.3378472

[33] 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*, 2017, pp. 468–482.

[34] Intel. Intel® core™ i7-10875h processor. [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/core/i7-processors/i7-10875h.html

[35] ——. Intel® optane™ dc persistent memory. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[36] ——. Persistent memory programming. [Online]. Available: https://pmem.io/

[37] ——. Platform brief intel xeon processor c5500/c3500 series and intel 3420 chipset. [Online]. Available: https://www.intel.com/content/www/us/en/intelligent-systems/picket-post/embedded-intel-xeon-c5500-processor-serieswith-intel-3420-chipset.html

[38] ——. (2015) Intel® xeon® processor d product family technical overview. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview

[39] ——. (2016) Intel® 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

[40] ——. (2020) Intel® architecture instruction set extensions and future features programming reference. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf

[41] Intel and Micron. (2015) Intel and micron produce breakthrough memory technology. [Online]. Available: https://investors.micron.com/news-releases/news-release-details/intel-and-micron-produce-breakthrough-memory-technology

[42] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 427–442, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2980024.2872410

[43] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[44] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 660–671. [Online]. Available: https://doi.org/10.1145/2830772.2830805

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

[46] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 494–508. [Online]. Available: https://doi.org/10.1145/3341301.3359631

[47] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. USA: IEEE Computer Society, 2013, p. 29–40. [Online]. Available: https://doi.org/10.1109/IPDPS.2013.69

[48] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 481–493, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080229

[49] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, W. Wang, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language support for memory persistency," *IEEE Micro*, vol. 39, no. 3, p. 94–102, May 2019. [Online]. Available: https://doi.org/10.1109/MM.2019.2910821

[50] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," *SIGPLAN Not.*, vol. 51, no. 4, p. 399–411, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2954679.2872381

[51] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[52] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 256–267.

[53] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 460–477. [Online]. Available: https://doi.org/10.1145/3132747.3132770

[54] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 2–13, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1555815.1555758

[55] C. Lin, V. Nagarajan, and R. Gupta, "Fence scoping," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press, 2014, p. 105–116. [Online]. Available: https://doi.org/10.1109/SC.2014.14

[56] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," *SIGPLAN Not.*, vol. 52, no. 4, p. 329–343, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3093336.3037714

[57] Q. Liu, J. Izraelevitz, 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*, ser. MICRO-51. IEEE Press, 2018, p. 258–270. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00029

[58] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 310–323.

[59] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 411–425. [Online]. Available: https://doi.org/10.1145/3297858.3304015

[60] D. E. Lowell and P. M. Chen, "Free transactions with rio vista," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, p. 92–101, Oct. 1997. [Online]. Available: https://doi.org/10.1145/269005.266665

[61] 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.

[62] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *SIGPLAN Not.*, vol. 52, no. 4, p. 135–148, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3093336.3037730

[63] D. Narayanan and O. Hodson, "Whole-system persistence," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, p. 401–410, Mar. 2012. [Online]. Available: https://doi.org/10.1145/2189750.2151018

[64] M. S. Neuvonen, A. Wolski, and V. Raatikka. (2011) Telecommunication application transaction processing (tatp). [Online]. Available: http://tatpbenchmark.sourceforge.net/

[65] T. M. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 507–519. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00048

[66] Y. Ni, J. Zhao, D. Bittman, and E. L. Miller, "Reducing nvm writes with optimized shadow paging," in *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'18. USA: USENIX Association, 2018, p. 22.

[67] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, p. 265–276, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2678373.2665712

[68] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 672–685. [Online]. Available: https://doi.org/10.1145/2830772.2830802

[69] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-tso: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, p. 89–97, Jul. 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[70] 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*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 178–190. [Online]. Available: https://doi.org/10.1145/3123939.3124539

[71] 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*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 175–186. [Online]. Available: https://doi.org/10.1145/3079856.3080240

[72] T. P. P. C. (TPC). (2010) Tpc benchmark b. [Online]. Available: http://www.tpc.org/tpcc/default.asp

[73] S. A. Vakil Ghahani, M. T. Kandemir, and J. B. Kotra, "Dsm: A case for hardware-assisted merging of dram rows with same content," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 2, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3392151

[74] 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 Stroage Technologies*, ser. FAST'11. USA: USENIX Association, 2011, p. 5.

[75] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592810

[76] 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*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 91–104. [Online]. Available: https://doi.org/10.1145/1950365.1950379

[77] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, no. 10, p. 865–876, Jun. 2014. [Online]. Available: https://doi.org/10.14778/2732951.2732960

[78] wikichip. Core i7 - intel. [Online]. Available: https://en.wikichip.org/wiki/intel/core_i7

[79] Wikipedia. (2020) Themis. [Online]. Available: https://en.wikipedia.org/wiki/Themis

[80] X. Wu and A. N. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[81] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 476–488.

[82] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. USA: USENIX Association, 2016, p. 323–338.

[83] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium*

on Operating Systems Principles, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 478–496. [Online]. Available: https://doi.org/10.1145/3132747.3132761

[84] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15. USA: USENIX Association, 2015, p. 167–181.

[85] M. Zhang, K. T. Lam, X. Yao, and C.-L. Wang, "Simpo: A scalable in-memory persistent object framework using nvram for reliable big data computing," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3167972

[86] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 3–18. [Online]. Available: https://doi.org/10.1145/2694344.2694370

[87] 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*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 421–432. [Online]. Available: https://doi.org/10.1145/2540708.2540744