PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency)

Jungi Jeong
jungijeong@purdue.edu
Purdue University, USA

Changhee Jung
chjung@purdue.edu
Purdue University, USA

ABSTRACT

Persistency models define the persist-order that controls the order in which stores update persistent memory (PM). As with memory consistency, the relaxed persistency models provide better performance than the strict ones by relaxing the ordering constraints. To support such relaxed persistency models, previous studies resort to APIs for annotating the persist-order in program and hardware implementations for enforcing the programmer-specified order. However, these approaches to supporting relaxed persistency impose costly burdens on both architects and programmers.

In light of this, the goal of this study is to demonstrate that the strict persistency model can outperform the relaxed models with significantly less hardware complexity and programming difficulty. To achieve that, this paper presents PMEM-Spec that speculatively allows any PM accesses without stalling or buffering, detecting their ordering violation (e.g., misspeculation for PM loads and stores). PMEM-Spec treats misspeculation as power failure and thus leverages failure-atomic transactions to recover from misspeculation by aborting and restarting them purposely. Since the ordering violation rarely occurs, PMEM-Spec can accelerate persistent memory accesses without significant misspeculation penalty. Experimental results show that PMEM-Spec outperforms two epoch-based persistency models with Intel X86 ISA and the state-of-the-art hardware support by 27.2% and 10.6%, respectively.

CCS CONCEPTS

• **Hardware** → **Memory and dense storage**; • **Computer systems organization** → *Parallel architectures.*

KEYWORDS

Persistency Model, Strict Persistency, HW/SW Codesign

ACM Reference Format:
Jungi Jeong and Changhee Jung. 2021. PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446698

1 INTRODUCTION

Although persistent memory (PM) technology has advanced recently, e.g., Intel/Micron 3D XPoint [2], it still imposes high overheads to realize recoverable data structures in the current architecture. They must carefully deal with data being persisted in PM for the recovery to rebuild the consistent states across failure on which all volatile data in caches and DRAM disappear. For this purpose, recoverable data structures necessitate controlling the order in which data is written to PM. However, modern architecture allows reordering of data writebacks to maximize the throughput, and enforcing any specific order—e.g., with cache-flush and store-fence—thus leads to significant performance degradation.

Recent works propose persistency models that define the persist-order which controls the order of writes to PM [8, 11, 16, 17, 24, 28, 30, 36, 39]. Combined with failure-atomicity support in software [3, 8, 10, 15, 19, 20, 29, 31, 32, 34, 45] and hardware [6, 13, 18, 22, 23, 26, 38, 42], these models enable a part of program, which is delineated by a failure-atomic transaction, to be recoverable with all the data persisted therein all or nothing. In general, the relaxed persistency models are more performant than the strict one since they increase PM write concurrency by relaxing the ordering constraints.

After all, the higher performance comes at the costs of program annotation and hardware complexities [17, 30, 36]. To exploit inherent parallelism in program, programmers must reason about the persist-order and insert new instructions such as fence/barrier to where they should be. Then, hardware modifications follow to enforce the persist-order specified in program, e.g., prior research proposals employ a special buffer alongside the L1 cache to govern the persist-order as shown in Figure 1. Such extensions require intrusive modifications on the existing cache hierarchy as well as the cache-coherence mechanism.

To a large extent, the current PM research trend recalls the advent of relaxed consistency models such as TSO for addressing SC's poor performance and scalability. For example, TSO relaxes the ordering constraints of SC and requires programmers to insert synchronization operations such as mutexes correspondingly for program correctness. Likewise, previous studies of relaxed persistency models place additional burdens on programmers and architects to achieve higher system throughput. Unfortunately, the hardware cannot improve throughput (higher PM write concurrency) unless programmers properly insert the ordering primitives into program, which often requires understanding the subtle concept of the underlying relaxed persistent models.

This reminiscence motivates us to rethink how hardware should support the memory persistency models. With that in mind, we propose PMEM-Spec, a hardware-software codesign scheme that can minimize (1) hardware modifications while leaving the CPU caches unmodified and (2) ordering annotation in program as with the

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04...$15.00
https://doi.org/10.1145/3445814.3446698

strict persistency model, while (3) delivering higher performance even compared to the relaxed models. In a sense, PMEM-Spec enables a *lightweight yet performant strict persistency model leaving the program almost as-is*. The key idea of PMEM-Spec is to allow any PM accesses *speculatively* without stalling and later correct if they violated the ordering constraints. Given that the ordering violation (e.g., `misspeculation`) is rare, we dare to challenge the presumption that the relaxed models outperform the strict ones. The empirical results demonstrate that PMEM-Spec achieves significantly higher performance than the epoch-based relaxed models.

To guarantee the persist-order with minimal hardware changes, PMEM-Spec proposes to separate load/store-paths to PM. PM loads go to the regular-path (e.g., through CPU caches) while PM stores through the *persist-path* that bypasses the cache hierarchy, as shown in Figure 1d. The persist-path directly connects the store queue to the PM controller and *leaves CPU caches unmodified*. PMEM-Spec sends PM data being stored to both the CPU caches and the persist-path simultaneously when they leave the store queue after their commit. The store requests sent through the persist-path arrive at the PM controller in their *commit order*—simplifying intra-thread persist-order and rendering strict persistency. In contrast, the LLC data are silently dropped on eviction without being written to PM.

However, PMEM-Spec's separate data-paths can cause the ordering violation—e.g., speculation can be `misspeculation` in case of data-race in different data-paths. In particular, PMEM-Spec identifies two possible PM misspeculations that can arise for loads and stores. First, PM load misspeculation causes incorrectly fetching the *stale value* in PM while the new value is pending in the persist-path. This violation can occur when a load arrives at the PM controller earlier than stores—executed earlier than the load—to the same memory address. Second, PM store misspeculation can happen when stores from different threads access the same address through different persist-paths, resulting in a *missing update* if they persist out of order. The store execution order (e.g., the coherence order) can differ from the actual persist-order made to PM if inter-thread dependency exists. In such a case, PMEM-Spec encounters PM store misspeculation that violates the inter-thread persist-order.

Ideally, misspeculation can be detected when stores overwrite recently-fetched (for load misspeculation) or recently-modified (for store misspeculation) blocks by checking the previous fetch or persist was benign. However, the question is how many following stores PMEM-Spec should monitor for overwriting. Our key observation is that misspeculation happens with a data-race between different data-paths. Therefore, *speculation always will be identified as correct or incorrect within a short interval*, which we call the `speculation window`.

Based on this observation, PMEM-Spec presents HW/SW codesign that detects misspeculation in hardware and delegates misspeculation recovery to software. To detect PM load misspeculation, PMEM-Spec leverages the fact that PM load misspeculation never occurs when the data is in CPU caches. That is, PMEM-Spec monitors recently evicted blocks from LLC whether they are overwritten by stores within the speculation window. Furthermore, PMEM-Spec exploits a happens-before order established by synchronization primitives in data-race free programs to detect PM store misspeculation. To convey the happens-before order to the hardware, the PMEM-Spec compiler annotates a critical section to assign the speculation ID to each thread that enters therein without programmers' effort. That way, PMEM-Spec can identify PM store misspeculation when it receives data with a lower speculation ID than the previous ones for a given cache block aligned address—e.g., this violates the inter-thread persist-order.

Finally, PMEM-Spec takes advantage of failure-atomic software to correct misspeculation by treating it as a *virtual* power failure [3, 8, 10, 15, 19, 20, 29, 31, 32, 34, 45]. Upon detecting misspeculation during program execution, PMEM-Spec immediately interrupts the operating system (OS). The OS then relays misspeculation detected to the failure-atomic runtime that aborts the current failure-atomic sections (FASEs) or transactions and executes the failure-recovery protocol. Once the recovery completes (as defined in failure-atomic runtime), the interrupted FASEs or transactions restart from the beginning instead of re-executing the whole program. It turns out that misspeculation is rare, and thus PMEM-Spec can accelerate PM accesses without a hassle. Experimental results show that PMEM-Spec outperforms the epoch-based persistency models with Intel X86 ISA and HOPS [36], the state-of-the-art implementation of the epoch-based model support by 27.2% and 10.6%, respectively.

In summary, this paper makes the following contributions:

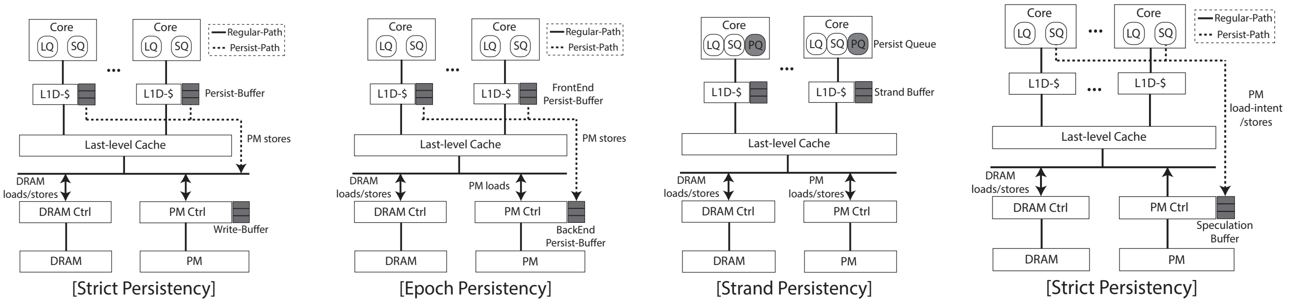
- We show how an efficient architecture/OS/compiler interaction achieves a high-performance strict persistency at a low hardware cost with minimal program change. For the first time, we demonstrate that the strict persistency (PMEM-Spec) can outperform the relaxed persistency (HOPS)
- We devise the decoupled persist-path that bypasses the cache hierarchy and directly connects the CPU store queue to the PM controller. The separate persist-path simplifies the intra-thread persist-order by sending stores in order, leaving the caches and their coherence unmodified.
- We classify how load and store misspeculations happen and devise detection schemes leveraging the speculation window. Misspeculation turns out to be rare, and thus delegating the misspeculation handling to software does not impose significant slowdown.

2 BACKGROUND

2.1 Persistency Models

The failure-atomicity ensures to recover data structures in persistent memory after a power outage, which requires to control the order of stores to persistent memory. However, modern hardware features complicate the recoverability since they buffer stores and reorder flushes to the underlying memory system. For instance, write-back caches lazily flush dirty cache lines to memory without preserving the order specified by the program.

Persistency models define the order of stores to persistent memory and allow programmers to control the persist-order in the program [11, 17, 24, 30, 36, 39, 43]. The persistency models span from the strict and relaxed models, which determine concurrency of the persist operations [39]. For example, in the strict persistency model or synchronous ordering, the volatile memory order equals the persistent memory operations. This model is intuitive and easing the burden on the programmer. However, it costs performance degradation to ensure failure-atomicity by limiting concurrency.



(a) DPO [30] **(b) HOPS [36]** **(c) StrandWeaver [17]** **(d) PMEM-Spec**

Figure 1: Comparing implementations of architecture support for persistency models. Shaded components indicate new hardware structures.

On the other hand, relaxed persistency models break the tie between volatile and persistent memory orders. For example, epoch persistency models divide stores into epochs specified by barriers and allow reordering in it [11, 24, 36, 43]. The persist-barrier strictly orders stores in different epochs. These models deliver higher performance than strict models by allowing out of order persists in an epoch, increasing concurrency. Furthermore, strand persistency creates a strand–each strand clears previous persist dependencies and appears in the persist-order as a new thread. The strand persistency model flushes multiple strands (e.g., epochs) simultaneously if they do not have a dependency on each other, extracting more concurrency within the program [17, 39].

2.2 Hardware Support for Persistency Models

Previous proposals allow programmers to define the desired persist-order so that the hardware can enforce it [17, 30, 36]. Programmers need to annotate program with the custom instructions based on the persistency model they assume. On top of programmers' annotation, the hardware mechanisms enforce the desired ordering defined by the persistency model.

For example, HOPS introduces `ofence` and `dfence` instructions that replace `SFENCE` of Intel X86 [36] while StrandWeaver adds new instructions that manipulate strands, such as `New/JoinStrand` and `persist-barrier` [17]. As shown in Figure 1, although each design presents different persistency models, all of them place a buffer alongside the L1 cache, i.e., persist- [30, 36] or strand-buffer [17], to govern the intra-thread persist-order. They either drop dirty cache lines evicted from LLC [30, 36] or explicitly write-back them before eviction [17]. Also, they monitor the cache-coherence messages to identify inter-thread dependency. The loss of exclusive permission in the L1 cache creates inter-thread dependency between the requester and the responder of coherence messages. Hardware is responsible for enforcing inter-thread dependency when flushing the persist or strand buffers.

3 MOTIVATION

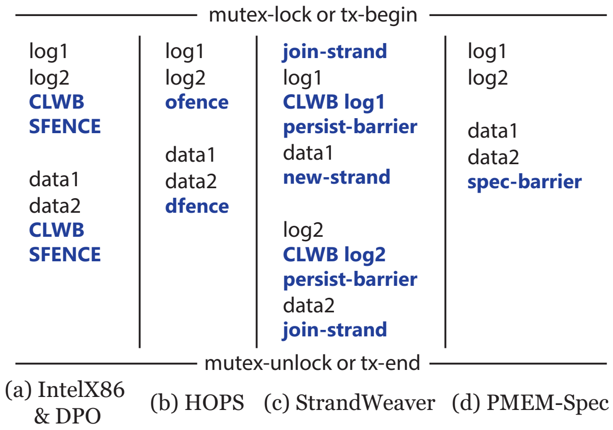
3.1 Intrusive Hardware Extensions

Prior solutions require intrusive modifications on the existing cache hierarchy to enforce the intra- and inter-thread persist-order. First, they place a buffer next to the L1 cache to keep dirty cache blocks before flushing them to PM. This buffer governs the intra-thread persist-order. Moreover, they monitor the coherence messages in the L1 cache to identify inter-thread ordering dependency. However, the challenge arises when L1 caches evict dirty cache blocks to the shared cache before flushing to PM. In this case, they cannot track the dependency only with L1 cache coherence messages, which might violate the inter-thread persist-order.

Although previous work proposed novel solutions, they come at the costs of hardware complexity. For example, DPO extended the cache-coherence protocol to include the persist-buffers [30]. Hence, this extension guarantees that the L1-cache eviction does not cause missing inter-thread dependency tracking since the persist-buffers hold the cache lines. Similarly, HOPS employed the sticky-M state [36], initially introduced in Log-TM [35], which enabled to track the ownership of the cache lines even after evicted to the shared cache. Therefore, DPO and HOPS allow the dirty L1-cache block eviction before being flushed from the persist-buffers but need to incorporate the cache coherence mechanism to enforce the inter-thread dependency.

On the other hand, StrandWeaver postpones the eviction from the L1 cache until the corresponding block in the strand-buffer flushes to PM [17]. For example, if a to-be-evicted block is found in the strand-buffer, the L1 cache deallocates the dirty block and moves it into the writeback buffer. The writeback buffer releases it after the corresponding block in the strand-buffer is flushed. Furthermore, if StrandWeaver receives an exclusive request to a dirty block in the L1 cache, it delays the response until the strand-buffer flushes it to PM. Lastly, StrandWeaver employs the persist queue to handle persist-order-related instructions and reduce store queue overflow, which stalls the CPU.

These extensions presented in previous proposals increase hardware complexity in not only the cache hierarchy but the core. Instead, this paper proposes *non-intrusive hardware changes introducing a minor extension to the store queue and not modifying the cache hierarchy.*



mutex-lock or tx-begin

(a) IntelX86 & DPO
```
log1
log2
CLWB
SFENCE

data1
data2
CLWB
SFENCE
```

(b) HOPS
```
log1
log2
ofence

data1
data2
dfence
```

(c) StrandWeaver
```
join-strand
log1
CLWB log1
persist-barrier
data1
new-strand

log2
CLWB log2
persist-barrier
data2
join-strand
```

(d) PMEM-Spec
```
log1
log2

data1
data2
spec-barrier
```

mutex-unlock or tx-end

Figure 2: Programming models in different persistency models.

3.2 Instrumenting Persist-Order

Figure 2 compares the programming models of different persistency models. In the relaxed persistency models, programmers must instrument program with ordering primitives based on the targeting model so that the hardware can relax relax ordering constraints. For example, HOPS places custom barrier instructions, `ofence` and `dfence`, between log and data operations and at the end of failure-atomic sections [36]. StrandWeaver inserts several instructions for creating or joining strands [17]. Based on the persist-order specified in program, hardware can relax the ordering constraints and exploit higher concurrency, leading to higher throughput than the strict models. However, this approach has a significant drawback. Program annotation increases the programming complexity since it often requires a deep understanding of program semantics to insert ordering primitives into the desired places. Otherwise, naive instruments may not fully exploit concurrency in programs, resulting in low hardware utilization.

On the other hand, hardware support for strict persistency does not add custom instructions to instrument the persist-order but uses the native primitives in unmodified programs [30]. This model ensures the persist-order equal to the volatile memory order by placing cache-flush followed by a store-fence instruction between every NVM store, which is easily dealt with by a compiler or library. Therefore, strict persistency always guarantees that NVM writes happen in-order so that the system can recover from a crash if combined with atomicity solutions. For example, DPO can run the epoch-based persistency model with the Intel X86 ISA without modification. However, this approach may show lower performance than the relaxed model with custom instructions.

Ideally, hardware support must be transparent to software and minimize the programming difficulty, which favors strict persistency for its simple programming model. However, if suffers from low performance due to its strict ordering constraints. To tackle this presumption, this study presents *hardware/software codesign for strict persistency that outperforms the relaxed models.*

4 PMEM-SPEC OVERVIEW

4.1 Design Goals

In this paper, we propose PMEM-Spec that pursues the following design goals:

- *PMEM-Spec minimizes the hardware extension.* PMEM-Spec does not modify the cache hierarchy, including the cache-coherence mechanism, and does not monitor it to track inter-thread dependency. Instead, PMEM-Spec exploits the separate data-path for persists, bypassing the cache hierarchy and directly connecting the store queue to the PM controller.
- *PMEM-Spec minimizes the annotation from programmers.* Since PMEM-Spec follows the strict persistency model, it does not require the persist-order annotation except the spec-barrier instruction at the end of failure-atomic regions.
- Despite strict persistency, *PMEM-Spec maximizes application performance* by allowing any PM accesses as they appear to the PM controller while not delaying or stalling them in caches.

4.2 Separate Data-Path for Persists

Although CPU caches are critical to compensate for PM access latency, they complicate to ensure the persist-order since they buffer and reorder data writebacks to PM. This complication makes it inevitable to bypass the cache hierarchy to minimize hardware modifications. Therefore, PMEM-Spec provides a separate store-path that directly connects the store queue to the PM controller, as shown in Figure 1(d). PMEM-Spec uses this path to update PM data, which means that the dirty cache blocks are silently dropped on their eviction without updating PM. This unique architecture allows the PM controller of PMEM-Spec to receive PM load and store with separate paths, regular- (through caches) and persist-path, respectively.

While the separated data-paths for persists are not new [30, 36], the prior solutions proposed the data-path connecting the persist-buffers beside the L1 cache to the PM controller (shown in Figure 1(a) and (b)). On the other hand, PMEM-Spec pushes data being stored into the persist-path immediately when the store instruction commits from the store queue. The persist-path does not buffer data but directly sends them to the PM controller, which coalesces and buffers the store data.

The persist-path guarantees that the data arrive at the PM controller in the commit-order, rendering the intra-thread persist-order equal to the volatile memory order. Therefore, PMEM-Spec provides a *strict persistency model*, which simplifies the programming model, as illustrated in Figure 2. On the other hand, the persist-paths in each core operate independently and do not provide a global ordering. Stores in different paths can arrive at the PM controller out of order, which can lead to violating the inter-thread dependency (explained in Section 5.2).

PMEM-Spec does not require the ordering annotation by programmers except the durability barrier, `spec-barrier`. The spec-barrier instruction guarantees that previous PM stores arrive at the persistent domain (e.g., the PM controller). The spec-barrier is the same as `dfence` of HOPS [36] and `persist-barrier` of

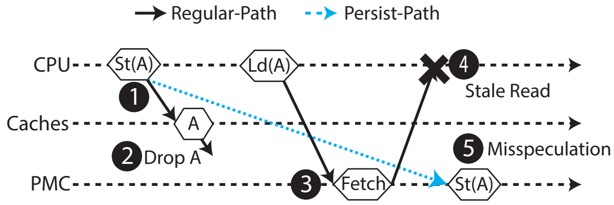


Figure 3: The Stale Read Problem.

StrandWeaver [17] and necessary to build a failure-atomic section (FASE). Therefore, it is required to place the spec-barrier instruction at the end of FASEs or transactions to guarantee durability. Furthermore, PMEM-Spec excludes the barrier-instruction between log and data operations since they persist sequentially via the persist-path.

4.3 Speculative PM Accesses

PMEM-Spec speculates that all PM accesses (both load and store) obey the correct ordering constraints and, thereby, executes them as they appear to the PM controller. PMEM-Spec does not intervene in PM accesses within cores and caches. However, due to the separated persist-path, PMEM-Spec can encounter an ordering violation (e.g., `misspeculation`). For PM load, it must read the latest value generated by the most recent store instruction, i.e., memory consistency. However, PM load misspeculation can happen when a load reads a stale value from PM because of a data-race in the regular- and persist-path (Section 5.1). For PM stores, they must persist in the program order, i.e., strict persistency. Stores from different threads can result in store misspeculation due to a data-race in persist-paths, which violates the inter-thread persist-order (Section 5.2). In both misspeculation, PMEM-Spec must detect and recover it for the program's correctness. Section 5 elaborates on how PMEM-Spec detects misspeculation.

4.4 Misspeculation Recovery

PMEM-Spec detects misspeculation in hardware but delegates recovery to software. To erase the speculated data, i.e., a stale value for load misspeculation and a missing update for store misspeculation, PMEM-Spec treats misspeculation as *virtual* power failure and takes advantage of failure-atomic libraries by aborting and re-executing current FASEs or transactions to recover [3, 8, 10, 15, 20, 31, 32, 34, 45]. If the hardware detects either load or store misspeculation, it traps the operating system (OS) to notify the occurrence of the ordering violation. Then, the OS signals the failure-atomic runtime that handles misspeculation recovery. Note that PMEM-Spec does not restart the whole program on a misspeculation; it only re-executes the transactions or FASEs currently executing at the time of misspeculation encountered. More details about misspeculation recovery are covered in Section 6.

5 MISSPECULATION DETECTION

5.1 PM Load Misspeculation

5.1.1 Stale Read Problem. The latency difference between the regular-path (e.g., PM load) and persist-path (e.g., PM store) can cause the stale read problem. Since there is no ordering guarantee between the two paths, PM loads from the regular-path and PM stores from the persist-path can appear in the PM controller out of order. Figure 3 illustrates how the stale read problem occurs. Suppose that CPU executes a store and load instructions to the same cache block A. If the store completes (❶), and caches evict the block (❷) before the load instruction, the load request goes to the PM controller to fetch the block since it misses in caches (❸). However, *the block brought from PM is stale* (❹) since (1) PMEM-Spec silently drops the dirty block A when evicted (❷) without updating persistent memory and (2) the data being stored (❶) does not arrive at the PM controller yet. Even if it is hard to happen, PMEM-Spec observed PM load misspeculation (Section 8.4).

The prior studies presented solutions to prevent stale reads [30, 36]. DPO includes the persist-buffers in the cache-coherence domain that guarantees the PM controller always serves with new data if it is not found in caches. However, PMEM-Spec cannot leverage the cache-coherence protocol since it bypasses the cache hierarchy to minimize hardware changes. HOPS employs the hardware bloom filter in the PM controller —which contains addresses of cache blocks in the persist-buffers— to check for the stale read problem. However, this approach requires the bloom filter insertion for every PM store and checking for every PM load. HOPS postpones PM read requests if the bloom filter conflicts with the target address or produces a false positive.

5.1.2 Detecting Load Misspeculation. Instead, PMEM-Spec detects PM load misspeculation later once it *speculatively* allows all PM load accesses without delay. Ideally, a prior load request can be identified as misspeculation (e.g., the stale read problem) when stores overwrite the block fetched by the load. For example, in Figure 3, `Ld(A)` (❸) will be identified as a stale read when `St(A)` arrives at the PM controller (❺). However, the question is how far stores PMEM-Spec should monitor for overwriting. The key observation is that misspeculation happens with a data-race in the data-paths so that conflicting accesses must be executed simultaneously in an instruction window. Therefore, *speculation always will be considered as correct or incorrect within a short window.*

This observation drives us to devise a simple detection scheme that monitors recently fetched blocks. PMEM-Spec records addresses of recently fetched blocks and monitors whether stores overwrite them for a `speculation window`. The speculation window determines when the speculative access will be considered safe. This window must be long enough to capture the *worst-case* persist-path latency. Otherwise, the stale read problem goes undetected and can corrupt the programs' correctness. Section 8.1 explains more details on the speculation window. The speculation window begins when a load arrives at the PM controller. If a store is preceding the load in the program order, it must appear to the PM controller within the speculation window to identify the load as misspeculation. Otherwise, the load will be considered correct after the speculation window.

5.1.3 False Misspeculation. However, this monitoring scheme ends up producing frequent false misspeculation—e.g., determining the benign loads as stale reads. Figure 4 illustrates how the write-on-allocation policy causes false misspeculation. Assuming that block A is not present in caches, a store instruction should fetch block A from PM. Although this fetching is safe, PMEM-Spec would classify

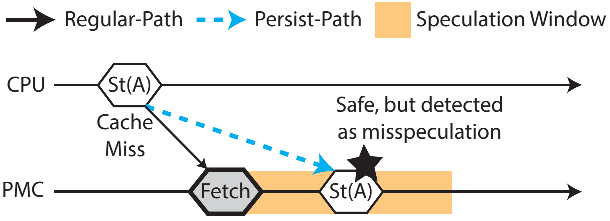


Figure 4: False misspeculation in monitoring fetched block.

Table 1: Description of states in the automata.

States	Description
`Initial`	Initially, all memory blocks are in the Initial state.
`Evict`	PMEM-Spec starts monitoring the block when PMC receives LLC writeback.
`Speculated`	The monitored block is fetched by read requests from the regular path.
`Misspeculation`	The previous read was misspeculation.

Table 2: Description of inputs in the automata.

Inputs	Description
WriteBack	LLC writeback from the regular path.
Read	PM loads from the regular path.
Persist	PM stores from the persist path.
Evict	The speculation window expiration.

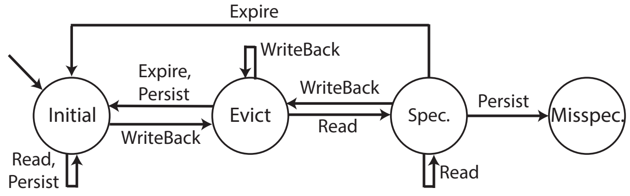
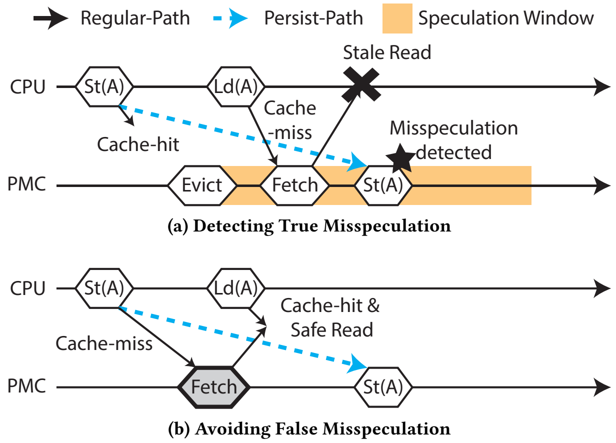


Figure 5: PM load misspeculation detection using automata.



(a) Detecting True Misspeculation

(b) Avoiding False Misspeculation

Figure 6: Eviction-based approach detects misspeculation without false-positives.

it as misspeculation when it receives the store data from the persist-path since the data overwrites the recently fetched block A within the speculation window. This false misspeculation always happens for every store instruction that misses in caches, leading to not acceptable recovery overheads. This limitation necessitates the new detection scheme that produces no (or rare) false misspeculation.

5.1.4 Eviction-Based Detection. To reduce frequent false misspeculation due to the write-on-allocation fetch, this paper presents an eviction-based detection scheme. The key idea is to monitor recently *evicted* blocks from the LLC instead of fetched blocks. The rationale behind this approach is that the block should be evicted before the load. Otherwise, caches will handle the load requests instead of PM. If load requests do not reach PM, PMEM-Spec never misspeculates PM loads, and the stale read problem does not occur.

PMEM-Spec uses automata to detect PM load misspeculation, as shown in Figure 5. PMEM-Spec maintains the automata state in the cache block granularity. Initially, all blocks in PM are in the Initial state. Table 1 describes each automata state and its meaning. Inputs for automata are explained in Table 2. Three inputs—e.g., `WriteBack`, `Read`, `Persist`—are requests received by the PM controller from the regular- and persist-paths. The last one, `Evict`, is a timer for the speculation window expiration. Note that PMEM-Spec does not monitor any blocks until the PM controller receives LLC writeback request (`WriteBack`), which changes the state from Initial to Evict.

The pattern that leads to the Misspeculation state is `WriteBack(s)-Read(s)-Persist`. Figure 6 illustrates how PMEM-Spec uses this pattern to detect misspeculation. In Figure 6a, suppose that the store instruction updates block A in caches, and the block is evicted. PMEM-Spec begins monitoring the evicted block when the PM controller receives the dirty block and triggers the speculation window (the yellow box). The PM controller then observes the fetch and persist requests sequantially from the regular-path and persist-path within the speculation window. When the PM controller accepts the persist request from the persist-path, PMEM-Spec can precisely identify the previous fetch was stale since the automata observe the `WriteBack-Read-Persist` pattern. Note that if store A does not appear in the speculation window, PMEM-Spec stops monitoring the block (`Expire`).

Furthermore, Figure 6b shows that how PMEM-Spec avoids false misspeculation by the write-on-allocation fetch. The block A is fetched from PM by the cache-miss of the store instruction. Meanwhile, the load instruction is issued and pending in the MSHR of caches waiting for the block allocation. Once the cache allocates the block A, the load instruction accesses the block in caches instead of persistent memory. In this case, since PMEM-Spec does not observe the writeback requests, it does not consider thee block A for a potential stale read candidate, e.g., it does not trigger the speculation window. Therefore, false misspeculation does not happen.

5.2 PM Store Misspeculation

5.2.1 Inter-Thread Persist-Order Violation. The persist-paths in different cores run independently and concurrently. Thus, stores from

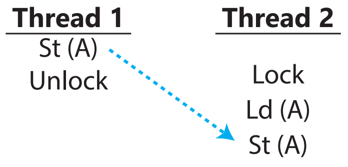


Figure 7: Example of the inter-thread dependency.

different threads can arrive at the PM controller in any arbitrary order. This implies that on the presence of inter-thread dependency, i.e., WAW (write-after-write) and RAW (read-after-write), PM store misspeculation may occur unless the dependence order is preserved during the PM access. For example, stores on the same cache block in different threads can appear out of order in PM, violating the inter-thread persist-order. The main reason for that is because the persist-path bypasses the cache hierarchy, i.e., the coherence-order and the persist-order can be different.

That is why prior works exploit the cache coherence messages to identify inter-thread dependency in the cache block [17, 30, 36]. They extended the cache coherence mechanism to track inter-thread dependency in all cache hierarchy. Unfortunately, this is not a viable option to pursue a low-cost and non-intrusive design, which leads PMEM-Spec's persist-path to bypass the cache hierarchy. Since PMEM-Spec cannot leverage the coherence traffic, it must identify the inter-thread ordering violation only with the stores observed in the PM controller.

5.2.2 Compiler-Assisted Detection. To overcome this challenge, PMEM-Spec leverages the unique characteristic of data-race free program. The main observation is that the memory order of conflicted accesses (e.g., `WAW or RAW inter-thread dependency`) must be explicitly ordered in program by synchronization primitives to be data-race free. For example, when threads have a race on the same memory address, programmers should protect the access with locks or semaphores to ensure correctness under the underlying memory consistency model. We found out that existing persistent programming models already assume a data-race free program as their target [8, 20, 21, 32].

In light of this, *PMEM-Spec exploits the happens-before order, which is dynamically established by synchronization primitives at run time, in order to enforce the inter-thread persist-order.* This approach allows PMEM-Spec to handle the inter-thread dependency without tracking cache coherence traffic. Figure 7 shows an example of a happens-before order made between two threads that write to the same memory address A. Here, the mutex (lock/unlock) establishes a happens-before order between the threads, thus confirming that St (A) in thread 1 precedes St (A) of thread 2.

This implies that the speculation ID is atomically incremented by the threads To convey the happens-before order to the persist-path, our compiler first identifies such a critical section[1]and instruments it with a *speculation* ID, i.e., a volatile global counter variable that monotonically increases at the entrance of the critical section. in the same order they enter critical sections. To realize this, PMEM-Spec adds a new instruction, `spec-assign`, that reads the speculation ID (the current counter value), saves it in a dedicated register, and increments it. Once a lock is acquired for a thread to enter the critical section, the compiler-inserted instrumentation code therein saves the speculation ID in the register using `spec-assign`. When PM stores in the critical sections leave the store buffer, the data being stored is tagged with the speculation ID saved in the register and pushed to the persist-path. That way, PMEM-Spec can accurately deliver the dynamic store execution order—governed by the happens-before order of synchronization operations—to the PM controller.

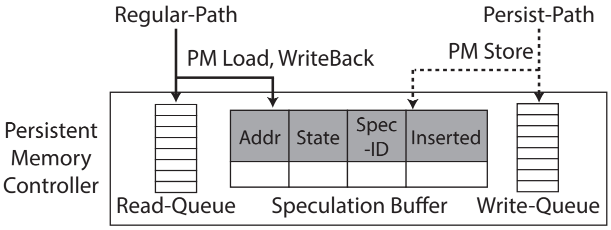


Figure 8: The speculation buffer in the PM controller.

Note that there is no need to identify store misspeculation outside critical sections because the data-race free program guarantees the absence of data-race. Therefore, when a thread exits the critical section, PMEM-Spec stops tagging PM stores. For this purpose, our compiler inserts another new instruction `spec-revoke`, that erases the speculation ID, at the end of critical section. Likewise, the RAW dependency between threads can be handled with the PM load misspeculation detection scheme—e.g., Ld(A) of thread 2 should appear later than St(A) of thread 1—as explained in Section 5.1.4.

PMEM-Spec saves/restores the special register storing the speculation ID across context switches to virtualize it. If not virtualizing the speculation ID, a thread is scheduled out inside a critical section may not tag the speculation ID after the thread is scheduled in.

5.3 Speculation Buffer

PMEM-Spec implements the speculation buffer in the PM controller to manage speculation. Figure 8 illustrates the architecture of the speculation buffer. The `Address` fields store the cache-block aligned addresses of the evicted blocks. The `State` fields maintain the automata state while the `Spec-ID` fields are used to detect PM store misspeculation. Lastly, the `Inserted` fields store the current cycle when PMEM-Spec begins the speculation window. PMEM-Spec creates a buffer entry when the PM controller receives LLC writeback from the regular-path (for PM load misspeculation detection) or persist requests from the persist-path (for PM store misspeculation detection) and starts the speculation window. The entries are deallocated after the speculation window expires. Note that PMEM-Spec does not store data in the speculation buffer

PMEM-Spec can cause the processor to stall when the speculation buffer has no free entries. Then, all cores pause and resume after the speculation window to make free spaces in the speculation buffer. Since the buffer entries are *short-living* and will be removed after the speculation window, PMEM-Spec does not require a large speculation buffer and can maintain high performance with a small number of entries (e.g., four entries in our evaluation).

[1] For user-defined synchronization operations, PMEM-Spec requires programmer annotation since a compiler cannot identify them.

6 MISSPECULATION RECOVERY

6.1 Software Support

This section explains the software support for PMEM-Spec to handle misspeculation.

6.1.1 *OS-Support.*

It requires OS support to deliver and detect misspeculation to the failure-atomic runtime. When the PMEM-Spec hardware detects misspeculation, it stores the physical address, where the misspeculation occurred, into a designated space reserved by the OS and generates a special HW interrupt. Since PMEM-Spec introduces a new HW interrupt, the OS should implement an appropriate interrupt handler for misspeculation handling purpose. For the HW interrupt raised by misspeculation detection, the OS should relay the interrupt to the failure-atomic runtime. In case there are multiple processes running failure-atomic solutions, the OS should be able to identify which process needs misspeculation recovery. To achieve this, the OS should maintain the mapping between the physical address—where PMEM-Spec identified misspeculation—and the process ID—where the failure-atomic program is executing. The OS can read the physical address stored in the designated space and find the process ID with this reverse mapping table. As a result, for a HW interrupt due to misspeculation, the OS can relay the interrupt to the exact process that requires misspeculation recovery.

6.1.2 *Runtime-Support.*

PMEM-Spec leverages the failure-atomic runtime to recover from misspeculation. One possible approach is to restart the whole failure-atomic application; on receiving the OS signal, the failure-atomic runtime could terminate all threads and execute the failure-recovery protocol from scratch as if power failure were encountered. However, recovering the entire process can be expensive.

Instead of restarting the whole program, PMEM-Spec re-executes only the interrupted FASEs or transactions to reduce misspeculation handling overheads. To realize this, PMEM-Spec requires the failure-atomic runtime to support the following:

- It should provide an abort handler that erases all intermediate data (i.e., both volatile and non-volatile) and restarts an interrupted FASE or transaction from the beginning. The transaction-based runtime naturally provides such an abort handler [10, 31, 45], while the locking-based one requires an extension [3, 8, 15, 20, 32, 34].
- It should register its own process ID in the OS's interrupt handler to receives misspeculation detection events.
- It should implement a *misspeculation* handler that receives a misspeculation detection signal from the OS. Section 6.2 further details the implementation of this handler.

6.2 Misspeculation Recovery

Although it is sufficient to abort or restart only the thread encountering misspeculation, PMEM-Spec conservatively rollbacks all the threads that are *currently* executing FASEs or transactions at the moment of misspeculation detection. That is because the hardware cannot figure out which thread causes the misspeculation. However, it is still challenging to interrupt and abort the threads simultaneously; even if a process receives the misspeculation signal, only one thread executes the interrupt handler while others do not.

6.2.1 *Lazy Recovery.*

To overcome this challenge, first, the failure-atomic runtime should maintain a per-thread variable, the *misspeculation flag*. A thread clears its own flag when it begins a new FASE or transaction, while these flags are set to true by a *misspeculation handler* on accepting the signal from OS. This handler sets the misspeculation flag of the threads currently running transactions to true while not modifying that of those outside transactions.

Second, although the hardware detected misspeculation in the middle of transactions, the threads should rollback and restart after completing the current FASEs or transactions. For example, the transaction-based runtime can check the misspeculation flag at the commit stage (e.g., validation phase) [10, 31, 45]. Similarly, the locking-based runtime must check the flag right before the FASE ends, unlocking the outermost lock [3, 8, 15, 20, 32, 34]. If the misspeculation flag is true, a thread executes an abort handler to invalidate all changes in volatile and non-volatile data and re-executes the current FASE or transaction. Since re-executing clears the flag, the thread does not abort again.

However, such a lazy recovery scheme might cause the application to crash before the FASE or transaction completes. Since the lazy scheme allows execution possibly with stale data (e.g., load misspeculation) or missed updates (e.g., store misspeculation), the application could encounter exceptions such as segmentation fault. Therefore, the misspeculation handler must catch all exceptions and selectively ignore them if they have been caused by misspeculation. For example, if the misspeculation flag is set to true, i.e., the exception was due to misspeculation, PMEM-Spec suppresses the exception and rollbacks the interrupted thread instead.

6.2.2 *Eager Recovery.*

Alternatively, the failure-atomic runtime could adopt an eager recovery scheme that stops and restarts all the threads immediately after receiving misspeculation detection. To realize this, the failure-atomic runtime should broadcast the OS signal to all threads within a process. For example, a thread that accepts the OS signal can send a synthetic interrupt to all other threads in a process using `pthread_kill` or similar mechanisms. Then, other threads that receive the synthetic interrupt abort if they are currently executing FASEs or transactions and restart the FASEs or transactions as defined in the abort handler of the failure-atomic runtime. Since the eager recovery scheme does not wait until the FASE or transaction finishes, it could reduce misspeculation overheads. Extending the failure-atomic runtime with the eager recovery scheme is left for our future work.

6.3 Recovery Overheads

Since PMEM-Spec does not restart the whole program on misspeculation, i.e., the abort handler only re-executes the interrupted transactions or FASEs, the recovery overheads are bound to the execution time of FASEs or transactions to be re-executed. For long transactions, they could adopt incremental checkpointing [44] and checkpoint pruning [27, 33] to reduce the recovery overhead. Moreover, the idea of incremental recovery has already been adopted by [32] that partitions program into small idempotent regions—achieving 400x faster recovery for some long FASEs. Thus, the misspeculation overhead is further bound to the re-execution of the regions that encounter misspeculation.

Table 3: Simulator configuration.

Core	2GHz, 8way-OoO 192-entry ROB 32-entry Ld/St Queue
L1 I/D Cache	32/64KB, 4-way, private 2ns hit latency (1ns tag/1ns data latency)
L2 Cache	16MB, 16-way, shared 20ns hit latency (10ns tag/10ns data latency)
PM Controller	32/64-entry read/write queue 4-entry speculation buffer
PM	Read = 175ns/Write = 94ns
Persist-Path	20ns

Table 4: Benchmarks used in our evaluation.

Benchmarks	**Description**
Array Swaps	Random swaps of array elements [30]
Concurrent Queue	Insert/delete nodes in a queue [30]
Hashmap	Read/update values in a hashmap [30]
RB-Tree	Insert/delete entries in a Red-Black tree [30]
TATP	Update location transaction in TATP [30]
TPCC	New order transaction in TPCC [30]
Vacation [7]	OLTP system that emulates a travel reservation system in Mnemosyne [45]
Memcached [14]	In-memory Key-Value store in Mnemosyne [45]

7 LIMITATIONS

PMEM-Spec currently cannot support systems with multiple PM controllers. Since PMEM-Spec detects the ordering violation inside the PM controller, it cannot detect the ordering violation of stores that access different PM controllers in the current design. To guarantee the correctness with multiple PM controllers, PMEM-Spec requires an extension to an on-chip network to make it respect the store order. We leave this extension as our future work.

Besides, compared to prior work that employs persist-buffers in the cache hierarchy [17, 30, 36], PMEM-Spec could generate more traffic to the PM controller since all PM stores bypass the caches. Therefore, in a pathological case such as write-dominant streaming applications, PMEM-Spec could stall CPUs due to the speculation buffer overflow. However, we have not observed such bursty writes in the benchmarks we evaluated.

8 EVALUATION

8.1 Methodology

We implemented and evaluated PMEM-Spec in the full-system simulation mode of the gem5 simulator [5]. We used the Linux kernel version 4.8.13 and Ubuntu 16.04. Table 3 summarizes the simulation configuration. We model the read and write latencies of a PM device by measuring the actual latency of Intel's Optane memory [2]. By default, we configure the persist-path latency to 20ns, which is longer than L1-to-PMC latency (11ns). This latency is reasonable (and conservative) if we implement the decoupled path from a core to the memory controller. If the persist-path latency is shorter than the L1-to-PMC latency, PM load misspeculation never occurs. We evaluate PMEM-Spec with different persist-path latencies in Section 8.3. We assume that the persist-paths are connected in the ring-bus, configuring the speculative period as the number of cores × the idle persist-path latency, which is 160ns in the main experiment.

Also, the speculation buffer has 4-entry where each entry requires 16B to store Address (8B), state (2-bit), the speculation ID (32-bit) and 30-bit for the Inserted field. Therefore, PMEM-Spec requires a total of 64B of storage overheads for the speculation buffer. We empirically selected the speculation buffer size. If not mentioned otherwise, the speculation buffer size is four. Note, the speculation buffer does not require nonvolatility. Thus, it has more room to grow compared to Intel's Write Pending Queue (WPQ), which must be drained to PM on power failure. We studied the impact of the speculation buffer size in Section 8.3.2. We assume that the PM controller supports ADR [40] and is in the persistent domain. All stores to PM from the persist-path will be durable once they appear at the PM controller.

Table 4 lists the benchmarks we used to evaluate PMEM-Spec. We evaluate PMEM-Spec with a set of microbenchmarks. The Array Swaps, Concurrent Queue, Hashmap, and RB-Tree are similar to those in NV-Heaps [10], DPO [30], and StrandWeaver [17]. TATP [37] benchmark executes the `update location` transactions, and TPCC [4] benchmark performs the `new order` transactions. These benchmarks provide failure-atomicity via undo-logging. Microbenchmarks run eight threads, and each thread performs 100K FASEs with a data size of 64B. Additionally, we study the real applications, `Vacation [7]` and `Memcached [14]`. They also run eight threads and use the Mnemosyne framework to support failure-atomicity. The data size of Memcached is 1024B. To measure the throughput of microbenchmarks, we only measured the multi-threaded kernel without considering the single-threaded initialization phase. For Vacation and Memcached, we measured the total execution time.

We compare the following designs in our evaluation:

- **Intel X86:** This design implements the epoch-based persistency model with `CLWB` and `SFENCE` instructions. SFENCE divides a program into epochs and ensures prior CLWBs complete before commit. Also, SFENCE orders log and data operations in this design.
- **DPO:** This design implements DPO [30] that builds the buffered strict persistency model. This design shares the same benchmarks with the Intel X86 design, using CLWB and SFENCE.
- **HOPS:** This design implements HOPS [36] that builds the epoch-based persistency model. HOPS implements `ofence` that divides the program into epochs and `dfence` that ensures durability by draining the persist-buffer. The `ofence` flushes the persist-buffer asynchronously while the `dfence` waits until the persist buffers drain. The `ofence` orders log and data writes while `dfence` ensures previous PM stores durability.

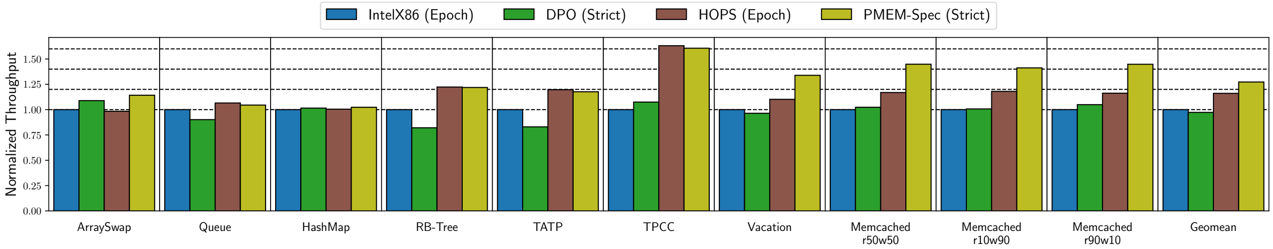


Figure 9: Performance comparison in the 8-core system.

- **PMEM-Spec:** This design implements our proposed architecture, PMEM-Spec. In this design, benchmarks execute no cache-flush and ordering instructions but `spec-barrier` at the end of FASEs or transactions.

8.2 Performance Comparison

Figure 9 compares the performance of each design in the 8-core system. We normalized throughput to the baseline, `IntelX86-Epoch`, and configured the persist-path latency of all DPO, HOPS, and PMEM-Spec to 20ns.

8.2.1 *Compared to the Baseline Intel X86.*

PMEM-Spec outperforms the Intel X86 epoch-based model, although PMEM-Spec implements a strict persistency model that does not relax the ordering constraints as the epoch-based model does. In IntelX86, SFENCE divides the program into epochs by ordering log and data operations and stalls CPUs until prior cache-flushes complete. Furthermore, CLWB and SFENCE consume the store queue entries, blocking CPUs if the store queue overflows. On the other hand, PMEM-Spec does not execute ordering primitives such as CLWB and SFENCE since the persist-path provides the intra-thread persist-order. As a result, PMEM-Spec shows a 1.27x speedup over the IntelX86 baseline in the 8-core system.

PMEM-Spec and other related work do not show performance improvement over the baseline for `Queue` and `Hashmap` benchmarks. These benchmarks have short failure-atomic sections, leading to execute persist-barrier instructions frequently. Since the persist-path latency is about twice longer than the regular-path, the persist-barrier overheads are more dominant in PMEM-Spec and HOPS. However, the persist-barriers in PMEM-Spec (e.g., spec-barrier) and HOPS (e.g., dfence) do not block volatile memory operations as SFENCE does. Therefore, PMEM-Spec and other schemes show comparable throughput even with higher persist-path latency.

On the other hand, PMEM-Spec shows significant performance improvement on the real applications, such as Vacation and Memcached. These benchmarks have relatively long transactions where PMEM-Spec can have enough room for speculation. Therefore, PMEM-Spec outperforms the baseline 33% and more than 40% in Vacation and Memcached, respectively. Note that PMEM-Spec only stalls at the end of failure-atomic regions with the spec-barrier instruction, while IntelX86 frequently executes SFENCE in the middle of failure-atomic regions.

8.2.2 *Compared to Previous Work.*

DPO even shows lower throughput than the baseline since it initially targeted the relaxed consistency model of the ARM architecture. Therefore, DPO enforces the persist-order for not only SFENCE but other barriers inherited in programs. However, this constraint is not necessary for the TSO of the Intel X86 architecture. Furthermore, DPO globally serializes PM stores and allows only a single flush to the persistent memory controller at once. On the other hand, HOPS proposed the custom instructions, lightweight ordering primitives such as `ofence` and `dfence` that replace SFENCE in the IntelX86 design. As a result, HOPS achieves a 15% higher throughput than the baseline.

However, PMEM-Spec also outperforms the prior studies, both DPO and HOPS. HOPS costs additional cycles to lookup the bloom filter in the PM controller for every PM load request and delays them if the bloom filter conflicts. Even if the bloom filter conflicts are rare, every PM load must seek the bloom filter before accessing PM. This limitation hinders HOPS from performing well in Mnemosyne benchmarks, where they have dominant PM loads compared to microbenchmarks. Furthermore, HOPS incurs an additional cycle in the bus between the private and shared caches since it needs to transfer an extra bit (e.g., sticky bit).

On the other hand, PMEM-Spec allows PM accesses without stalling, achieving higher throughput in microbenchmarks and Mnemosyne benchmarks. Therefore, although HOPS implements the epoch-based persistency model, PMEM-Spec delivers a higher throughput, which provides the strict model. This result demonstrates that the strict model can outperform the relaxed one with architecture implementation.

8.3 Sensitivity Study

8.3.1 *Number of CPUs.*

As the number of CPUs in a system increases, the separate data-paths put more pressure on the interconnection network. Figure 10 shows the throughput of all designs, including PMEM-Spec, in the 16-/32-/64-core systems.

In particular, DPO shows a lower throughput than the baseline in all benchmarks. DPO burdens another pressure on the cache-coherence bus since it includes the persist-buffer into the coherence domain. Therefore, as the number of CPUs increases, DPO would suffer more slow-down. Although HOPS shows higher throughput than the baseline in 16-/32-/64-core systems, it suffers performance overheads in caches and the PM controller. Finally, PMEM-Spec outperforms the baseline and HOPS by 18.8%/8.2%, 18.2%/8.0%, and 17.1%/10% in the 16-/32-/64-core systems, respectively.

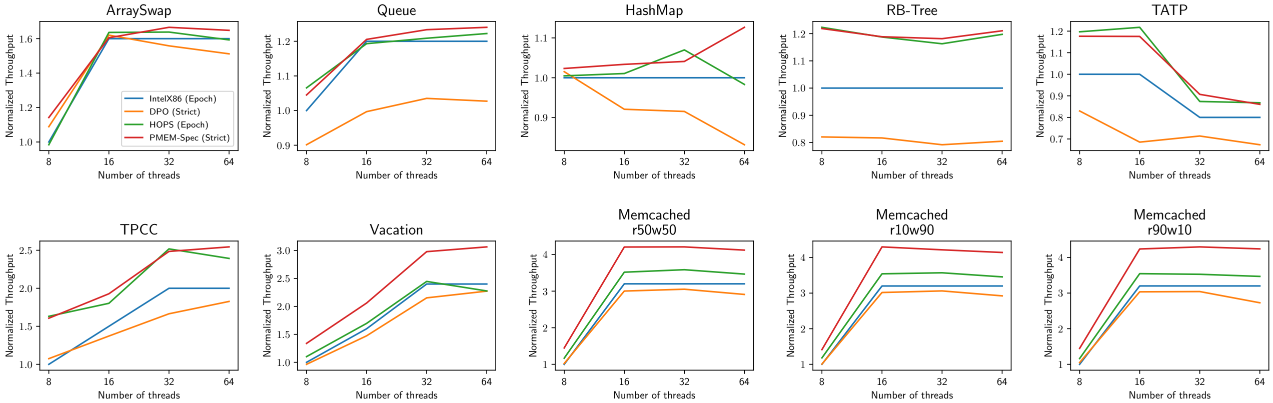


Figure 10: Sensitivity study of different numbers of threads.

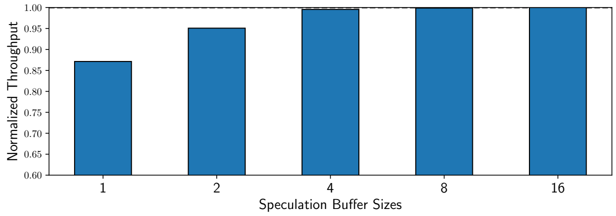


Figure 11: Sensitivity study of different speculation buffer sizes in the 8-core system.

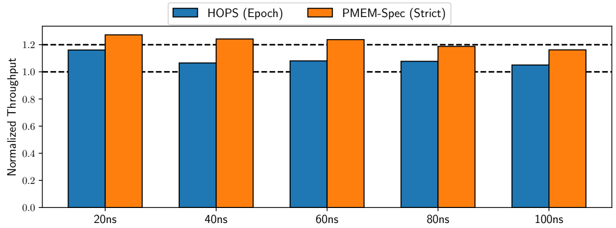


Figure 12: Sensitivity study of different persist-path latencies.

8.3.2 Speculation Buffer Sizes. Figure 11 shows the average throughput of benchmarks in the 8-core system when varying the speculation buffer size. If the speculation buffer becomes full, all cores must pause for the speculation window to wait for the entries to expire and resume. In particular, the application often experience stalls due to the speculation buffer full when the speculation buffer size is one, Such frequent pauses result in 12.8% throughput degradation compared to the overflow-free case, which is the entry size of 16. Note that PMEM-Spec can maintain the speculation buffer since 1) it creates the speculation buffer entry on the dirty block eviction from the last-level cache, and 2) they are short-living only for the speculation window. As the size of the speculation buffer increases, the average throughput also improves. When it comes to the speculation buffer with 16-entry, we have not observed overflows.

8.3.3 Persist-Path Latency. Figure 12 plots the geomean throughput of all benchmarks when changing the persist-path latency from 20ns to 100ns for HOPS and PMEM-Spec. We fixed the regular-path latency during the experiments. Since the durability barrier, such as dfence or spec-barrier, is not frequent, both HOPS and PMEM-Spec show higher performance than the baseline, even if the persist-path latency increases to 100ns.

8.4 Misspeculation Rates

Misspeculation rate is crucial in PMEM-Spec. If misspeculation happens frequently, the recovery overhead will outweigh the performance improvement. In our evaluation, PMEM-Spec never experienced misspeculation.

Although we have written a synthetic program that causes load misspeculation, misspeculation is extremely rare in practice. The following is how the synthetic program generates PM load misspeculation. First, the program updates data in the L1 cache then should make conflicting loads to the same cache set but different cache lines to evict the block all the way to PM. Depending on the cache hierarchy, the program may requires tens of memory accesses. Second, the program should load the data from PM instead of caches. For the PM load to be misspeculation, the program must execute all following loads before the store data arrives at the PM controller—e.g., they must be done within the persist-path latency. Indeed, this code pattern is not realistic. Furthermore, PM load misspeculation is only observed under an unrealistically long persist-path latency (e.g., 10x slower). Note that when the persist-path latency is shorter than the one of regular-path, PM load misspeculation never occurs.

Also, PM store misspeculation that only happens with inter-thread dependencies is utterly rare. According to the prior study [36], typical PM applications have almost zero inter-thread dependencies in a 50 micro-second window. Since PMEM-Spec's speculation window is orders-of-magnitude shorter than 50 micro-second, we can expect PM store misspeculation to be extremely rare as well.

9 RELATED WORK

Persistency models: Prior works have proposed persistency models that define the store orders to persistent memory [1, 11, 39]. Several solutions presented their hardware implementations [12, 17, 24, 30, 36, 41, 43]. DPO [30] implements the relaxed consistency buffered strict persistency (RCBSP). The DPO has a dedicated persist-path similar to PMEM-Spec and enforces the order specified by persistent and volatile barriers. However, it allows a single flush to persistent memory at a time and, thereby, limits concurrency. Moreover, it introduces the persist buffers alongside the L1 cache and snoops the cache-coherence traffic to identify the inter-thread dependency. HOPS [36], Shin et al. [43], and Joshi et al. [24] presented implementations of the buffered epoch persistency model with optimization on persist-barriers. Similar to PMEM-Spec, Shin et al. speculate the execution without stalling on the persist-barrier. However, Shin et al. do not have a dedicated path for persists. PMEM-Spec addressed challenges in detecting and handling misspeculation. Joshi et al. proposed an efficient implementation of persist-barriers. However, PMEM-Spec does not stall on the persist-barrier and guarantees intra-thread ordering with the persist-path. HOPS incurs an extra latency on every NVM reads by consulting the bloom filter to check the persist-path conflict. Furthermore, Dananjaya et al. present the lazy release persistency that designs the hardware for log-free data (LFD) structures [12].

In particular, StrandWeaver [17] presented the hardware implementation of the strand persistency model, initially proposed in Pelly et al. [39]. StrandWeaver allows multiple epochs to flush concurrently if not dependent, requiring programmers to denote creating and joining strands. The authors demonstrated the efficiency of StrandWeaver by showing better performance than HOPS, the state-of-the-art epoch-based persistency model. However, StrandWeaver extended a core with the persist queue along with load/store queues and caches with the strand-buffer and coherence mechanism that delays a response of exclusive requests. On the other hand, PMEM-Spec requires much lower hardware overheads compared to StrandWeaver. PMEM-Spec leaves the CPU cache unmodified and does not need to insert the persist-order information in programs, except for indicating the end of failure-atomic regions. However, this applies to all other prior work as well.

Recently, Themis [41] presented multi-store-path architecture to PM that differentiates temporal and non-temporal stores. Themis uses a non-temporal store path as a fast store-path to PM, while temporal stores use a slow data-path. Due to paths' latency difference, Themis can eliminate almost all persist-barriers, leading to higher performance of persistent applications. Although the multi-store-path architecture of Themis is similar to PMEM-Spec's separated load/store paths, it has the following differences. First, Themis assumes multiple store paths with different latencies. On the other hand, PMEM-Spec proposes separated load/store paths that potentially have no ordering guarantee in between. More importantly, Themis also requires CPU cache modifications similar to previous works [17, 30, 36], while PMEM-Spec does not.

Software failure-atomic solutions: PMEM-Spec relies on the software-based failure-atomic frameworks to handle misspeculation. Numerous previous studies proposed atomic updates in the software [3, 8–10, 15, 20, 21, 31, 32, 34, 45]. NV-Heaps [10], Mnemosyne [45], DudeTM [31], and ArchTM [46] provide library interfaces to build persistent memory data structures with transactions. Furthermore, SoftWrap [15], REWIND [9], Kamino-Tx [34], NVThreads [20], ATLAS [8], JUSTDO logging [21], and iDO [32] support mutex-based failure-atomic regions. PMEM-Spec can take advantage of these studies with an extension described in Section 6.1.

Hardware failure-atomic solutions: Although PMEM-Spec mainly relies on the software-based failure-atomic solutions, it can also use the hardware-based frameworks [6, 13, 18, 22, 23, 25, 26, 38, 42]. Most hardware-based proposals assume the locking-based isolation technique and do not support roll-back [6, 13, 18, 23, 26, 38, 42]. Therefore, PMEM-Spec requires these studies to support aborting intermediate states and re-executing transactions from the beginning, as elaborated in Section 6.1. On the other hand, a few studies have proposed hardware-based durable transactions that naturally provide aborts and restarts on conflicts [22, 25]. Hence, these studies show a better bit for PMEM-Spec since they require minor changes.

10 CONCLUSION

We proposed PMEM-Spec (persistent memory speculation), a novel hardware/software codesign scheme that achieves lightweight yet performant persist ordering. PMEM-Spec allows any persistent memory accesses speculatively without delaying them but detects ordering violation (e.g., misspeculation) in both load and store instructions. Upon detecting misspeculation, PMEM-Spec delegates it to the software failure-atomicity mechanism to recover from it. Also, PMEM-Spec leverages the separate FIFO data-path—connected to CPU's store queue— for persists, leaving CPU and caches unmodified and providing an intra-thread ordering guarantee without inserting barriers into program. Given that misspeculation is very rare, PMEM-Spec solves both performance and programmability problems of prior work without complicating the hardware.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Michael Swift, for his invaluable feedback. Our thanks also to the anonymous ASPLOS reviewers for their insightful comments, to Aasheesh Kolli for providing us with his simulator code, and the members of the CompArch group for early discussions on the project. This work was supported by NSF grants 1750503 (CAREER) and 1814430.

REFERENCES

[1] [n.d.]. Intel 64 and IA-32 Architecture Software Developer's Manual Volume 2A. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-2a-instruction-set-reference-a-l.html.

[2] [n.d.]. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[3] [n.d.]. PMDK: Persistent Memory Development Kit. https://github.com/pmem/pmdk.

[4] [n.d.]. TPC Benchmark C. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).

[6] Miao Cai, Chance C. Coats, and Jian Huang. 2020. *Hoop: Efficient Hardware-Assisted out-of-Place Update for Non-Volatile Memory.*

[7] Chi Cao Minh. 2008. *Designing an Effective Hybrid Transactional Memory System.* Ph.D. Dissertation.

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the*

ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA).

[9] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: ReCovery WRite-Ahead System for In-Memory Non-Volatile DAta-Structures. *Proc. VLDB Endow.* (2015).

[10] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[11] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP).*

[12] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. 2020. Lazy Release Persistency. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[13] Kshitij Doshi, Ellis Giles, and Peter Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[14] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* (2004).

[15] Ellis Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *Symposium on Mass Storage Systems and Technologies (MSST).*

[16] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[17] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[18] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed Logless Atomic Durability with Persistent Memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[19] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[20] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications. In *Proceedings of the European Conference on Computer Systems (EuroSys).*

[21] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[22] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded Hardware Transactional Memory for a Hybrid DRAM/NVM Memory System. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[23] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient Hardware-Assisted Logging with Asynchronous and Direct-Update for Persistent Memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[24] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the International Symposium on Microarchitecture (MICRO).*

[25] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA).*

[26] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[27] Hongjune Kim, Jianping Zeng, Qingrui Liu, Mohammad Abdel-Majeed, Jaejin Lee, and Changhee Jung. 2020. Compiler-directed soft error resilience for lightweight GPU register file protection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[28] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA).*

[29] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[30] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering. In *Proceeding of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[31] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[32] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. IDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[33] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC).*

[34] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the European Conference on Computer Systems (EuroSys).*

[35] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. LogTM: log-based transactional memory. In *The International Symposium on High-Performance Computer Architecture (HPCA).*

[36] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[37] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. 2011. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net

[38] Matheus A. Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *Proceeding of the IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[39] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the Annual International Symposium on Computer Architecture (ISCA).*

[40] Andy M Rudoff. [n.d.]. Deprecating the PCOMMIT Instruction. https://software.intel.com/blogs/2016/09/12/deprecate-pcommit-instruction.

[41] Sara Mahdizadeh Shahrai, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. 2020. (Almost) Fence-less Persist Ordering. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[42] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[43] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA).*

[44] Jaswanth Sreeram and Santosh Pande. 2012. Safe Compiler-Driven Transaction Checkpointing and Recovery. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).*

[45] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[46] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. 2021. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *USENIX Conference on File and Storage Technologies (FAST).*