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

Jungi Jeong and Changhee Jung
Purdue University

I. Introduction

Recent works propose persistency models that define the persist-order, which controls the order of writes to persistent memory [1]–[3]. Combined with failure-atomicity support in software [4], [5] and hardware [6], [7], these models enable a part of the program, which is delineated by a failure-atomic transaction, to be recoverable with all its data persisted all or nothing. In general, the relaxed persistency models are more performant than the strict ones since they increase PM write concurrency by relaxing the ordering constraints.

After all, the higher performance comes at the cost of program annotation and hardware complexities [1]–[3]. 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 to ensure the order. Then, hardware modifications follow to enforce the persist-order specified in the program, requiring intrusive extensions on the existing cache hierarchy and the cache-coherence mechanism.

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 the program as with the strict persistency model, while (3) delivering higher performance even compared to the relaxed models. 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, the empirical results demonstrate that PMEM-Spec achieves significantly higher performance than the epoch-based relaxed models.

In particular, PMEM-Spec proposes a decoupled *persist-path* that directly connects the store queue to the persistent memory controller. The persist-path leaves caches unmodified since it *bypasses* the cache hierarchy. PMEM-Spec sends PM data being stored to both the CPU caches and the persist-path immediately when they leave the store queue after their commit. The store requests sent through the persist-path arrive at the persistent memory controller in the *commit order*—simplifying intra-thread persist-order—whereas the data reached LLC are silently dropped on the eviction of their dirty cache block without being written to PM.

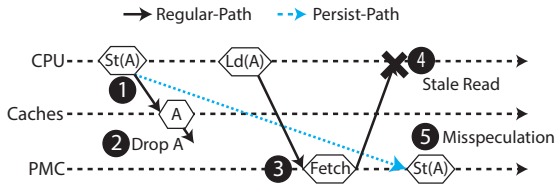
More importantly, PMEM-Spec must detect the ordering violation (e.g., `misspeculation`). In particular, PMEM-Spec identifies two possible PM misspeculations that can arise for loads and stores on the separate data paths: (1) PM load misspeculation occurs when a load from the regular (e.g., CPU caches) path arrives earlier than stores to the same memory address from the persist-path while stores appear before the load in the program order. This violation ends up fetching the stale value from PM while the new value is pending on the persist-path. (2) PM store misspeculation can occur due to the data race between persist-paths if inter-thread dependency exists. Since the persist-path bypasses the cache hierarchy, the coherence-order and the persist-order can be different. For example, conflicting stores in different threads can appear out of order in PM, violating the inter-thread persist-order. For both ordering violations, PMEM-Spec can detect them with a minimal hardware change.

Finally, PMEM-Spec takes advantage of failure-atomic software to correct misspeculation by treating it as a *power failure* [4], [5]. Upon detecting misspeculation during program execution, PMEM-Spec immediately interrupts the operating system. Then, it signals the failure-atomic runtime to abort all threads and recover from a *virtual* power failure. Once the recovery runtime completes (i.e., restoring necessary logs), the program restarts from the last consistent state. It turns out that misspeculation is rare, and thus PMEM-Spec can accelerate persistent memory accesses without a hassle.

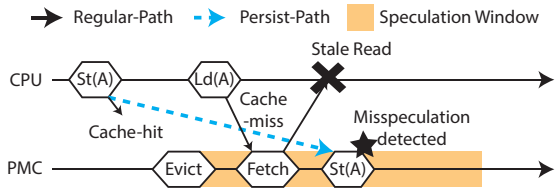
II. Design

1) Separate Data-Path for Persists: PMEM-Spec provides a separate store-path to persistent memory—that directly connects the store queue to the persistent memory controller. PMEM-Spec uses this persist-path to update persistent memory data, which means that the dirty cache blocks in LLC do not update persistent memory and disappear if evicted. PMEM-Spec pushes data being stored into the persist-path immediately when the store instruction commits from the store queue. The persist-path guarantees that the data arrive at the persistent memory 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*.

2) Speculative PM Accesses: PMEM-Spec speculates that all persistent memory accesses follow the correct ordering constraints and, thereby, allows all PM accesses in any order as they appear to PM. However, the data race in the regular (e.g., caches) and persist-paths causes the ordering violation (e.g., `misspeculation`). PM load misspeculation happens when load requests read stale data from persistent memory due to the latency disparity of the regular-path and persist-path. Also, unordered stores from different threads can result in store misspeculation, which violates the persist-order. In both



(a) Load misspeculation (the stale load problem).



(b) Detect load misspeculation

Fig. 1: Load misspeculation handling in PMEM-Spec.

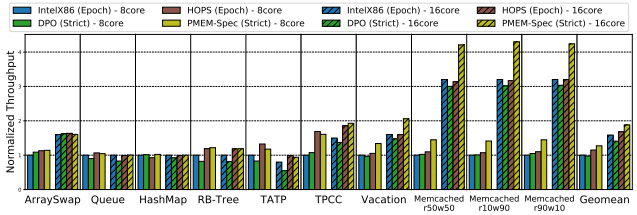


Fig. 2: Performance in the 8-core and 16-core systems.

misspeculation, PMEM-Spec must detect and recover it for the program's correctness.

Detect Load Misspeculation: Figure 1 shows (a) how load misspeculation happens and (b) how PMEM-Spec detects it. The key idea is to monitor recently *evicted* blocks from the caches. The rationale behind this approach is that the block should be evicted after stores and before the load. Otherwise, caches will handle the load requests instead of PM. If load requests do not reach PM, the stale read problem does not occur.

Detect Store Misspeculation: 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 WAR inter-thread dependency`) must be explicitly ordered in the program by synchronization primitives to be data-race free. In light of this, PMEM-Spec exploits the happens-before order, which is dynamically established by synchronization primitives at run time to enforce the inter-thread persist-order. This approach allows PMEM-Spec to handle the inter-thread dependency without tracking cache coherence traffics, simplifying the hardware mechanism.

III. Evaluation

We implemented and evaluated PMEM-Spec in the full-system simulation mode of the gem5 simulator. Figure 2 compares the performance of each design in the 8-core and 16-core systems. We normalized throughput to the 8-core baseline, `IntelX86-Epoch`.

PMEM-Spec outperforms the Intel X86 epoch-based model, although PMEM-Spec implements a strict persistency model that does not relax the ordering constraints. In IntelX86, SFENCE divides the program into epochs by ordering log and data operations. The IntelX86 design stalls CPUs until SFENCE completes. On the other hand, PMEM-Spec never stalls CPUs unless on persist-barriers at the end of failure-atomic regions. As a result, PMEM-Spec shows 1.27x and 1.18x speedup over the IntelX86 baseline int the 8-core and 16-core systems, respectively.

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 performing well in Mnemosyne benchmarks, where they have dominant PM loads compared to microbenchmarks. On the other hand, PMEM-Spec allows PM accesses without stalling, achieving higher throughput in both micro 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.

IV. 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. Upon detecting the ordering violation, 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, providing an intra-thread order 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.

References

[1] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *MICRO*, 2016.

[2] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *ASPLOS*, 2017.

[3] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *ISCA*, 2020.

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

[5] 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 *MICRO*, 2018.

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

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