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

Jungi Jeong and Changhee Jung

Purdue University

1. Motivation

Persistency models govern the order in which stores update persistent memory (PM). As with memory consistency, the relaxed persistency models provide better performance than the strict models by relaxing the ordering constraints, such as allowing reordering within an epoch [2,8,13,16] or a strand [3, 15]. However, the higher performance comes at the cost of program annotation and hardware complexities [3, 11, 13]. 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 programs, e.g., prior research proposals employ a special buffer alongside the L1 cache to govern the persist-order, as shown in Figure 1 in the full paper.

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, such as write-after-write dependency, of SC and delegates programmers to ensure the correctness and place ordering primitives. Likewise, previous studies of relaxed persistency models place additional burdens on programmers and hardware to achieve higher system throughput.

This reminiscence drives us to rethink how hardware support should shape persistency models. Our goal is to devise a hardware-software codesign that *simultaneosly achieves high throughput and less hardware/programming complexity*.

2. Limitations of the State of the Art

2.1. Intrusive Hardware Extensions

Prior solutions require intrusive modifications on the core and cache hierarchy to enforce the intra-thread and inter-thread persist-order [3, 11, 13]. 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 dependency. However, it becomes challenging 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, violating the inter-thread persist-order.

Although previous work proposed novel solutions, they come at the cost of hardware complexity. For example, DPO incorporates the persist buffers into the cache coherence domain [11]. HOPS employs the sticky-M state [13], keeping the cache lines ownership to the private cache where it resided before eviction. StrandWeaver postpones removing the dirty block from the L1 cache until the corresponding block in the strand-buffer flushes to PM [3]. Moreover, StrandWeaver introduces a persist queue in a core to handle PM-related instructions and the overflow of a store queue, which stalls the pipeline.

These extensions increase hardware complexity in not only the cache hierarchy but the core. On the other hand, we propose a *non-intrusive implementation that modifies neither the core nor the cache hierarchy*.

2.2. Instrumenting Persist-Orders

Programmers must instrument ordering primitives in the program based on the target relaxed persistency model [3, 13]. Based on the persist-order specified in program, hardware can relax the ordering constraints and exploit higher concurrency. However, this approach has a significant drawback. Instrumenting the persist-order in program increases the programming complexity. This 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 program, resulting in suboptimal performance due to the hardware underutilization.

On the other hand, hardware support for strict persistency models does not add custom ISAs to instrument the persist-order [11]. For example, DPO can run the epoch-based persistency model with the Intel X86 ISA without modification. However, this approach may show a relatively poor performance compared to the relaxed model with custom ISAs.

We aim to design hardware support that is transparent to software and minimizes the burden on programmers. Meanwhile, we *tackle the presumption that a strict persistency model would show lower performance than the relaxed models by demonstrating the opposite*.

3. Key Insights of PMEM-Spec

3.1. Bypass the Caches

The first insight is to bypass the caches. CPU caches are critical to compensate for PM access latency. However, they also deteriorate persistence and increase hardware complexity by reordering data writebacks to PM. Therefore, this paper proposes a separate data-path—bypassing the cache hierarchy for PM stores— that directly connects the store-queue to the PM controller. PM stores update both caches and PM through the persist-path simultaneously while the PM controller drops dirty cache block without updating PM.

This design decision brings two advantages over the previous solutions.

- CPUs and caches remain unmodified since the persist-path bypasses the cache hierarchy, significantly reducing the hardware complexity.
- The persist-path pushes data being stored—immediately when the store instruction commits—from the store-queue to the persist-path; it guarantees that PM stores happen in the *commit-order*.
- This guarantee removes ordering primitives in programs, such as CLWB & SFENCE between log and data operations, simplifying the programming model.

The persist-path makes the persist-order equal to volatile memory order, rendering PMEM-Spec a *strict persistency model*.

3.2. Speculate PM Accesses

The second insight is to *speculate* PM accesses as they (almost) always obey the ordering constraints. Previous studies control cache-flushes in the L1 cache to enforce the persist-order. On the other hand, PMEM-Spec removes buffers in the cache hierarchy and the core but allows PM accesses in any order as they appear to PM. If the ordering violation (e.g., misspeculation) happens, it should be properly handled for correctness, which leads to the third insight.

3.3. Recover from Misspeculation

We consider misspeculation as a power failure. Since persistent applications support the failure-atomicity via software [1, 5, 6, 10, 12, 17] or hardware [4, 7, 9, 14], we take advantage of them to recover from misspeculation. Upon detecting misspeculation, the proposed design immediately traps the operating system to notify it of a *virtual* power failure. Then, it signals the failure-atomic runtime to abort all threads and recover from the virtual power failure. Once the recovery runtime completes, the program restarts from the last consistent state.

4. PMEM-Spec Details

4.1. Why Misspeculation Occurs

The data race between the regular (e.g., caches) and persist-paths can cause the ordering violation. First, PM load misspeculation occurs when a load arrives at the PM controller earlier than stores to the same memory address while stores appear before the load in the program order. This violation ends up fetching the stale value from PM, violating the memory consistency model. Second, the latency disparity between persist-paths may result in PM store misspeculation if inter-thread dependency exists. Stores of different threads to the same block can arrive out of order at the PM-side. The out-of-order arrival violates the inter-thread dependency.

4.2. Misspeculation Detection

We observe that misspeculation can happen with a data race between data-paths. Therefore, speculative access becomes considered *safe* after the worst-case data-path latency, which we call the *speculation window*.

To detect PM load misspeculation, PMEM-Spec proposes eviction-based monitoring for recently evicted blocks from LLC since PM load misspeculation never occurs when the block is present in caches. Suppose PMEM-Spec detects that stores overwrite the block recently evicted from caches and fetched by the load within the speculation window. Here, it turns out that the load had a stale value (e.g., misspeculation). Furthermore, PMEM-Spec exploits the happens-before order, dynamically established at run time, to detect PM store misspeculation. To achieve this, our compiler annotates a critical section to assign the speculation ID to each thread that enters therein. That way, PMEM-Spec monitors incoming data from the persist-path and identifies the inter-thread ordering violation when it receives a lower ID than the previous one.

5. Key Results and Contributions

In our experiments, PMEM-Spec outperforms the epoch-based persistency model with Intel X86 ISAs and the state-of-the-art, HOPS [13], by 27.2% and 10.6%, respectively.

PMEM-Spec 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 a 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 by directly connecting the CPU store queue to the PM controller. The separate persist-path simplifies the intra-thread persist-order by sending stores in order.
- We propose persistent memory speculation to avoid CPU stalls, obviating the need to wait for data writebacks through the persist-path. Also, we classify how misspeculation happens and devise a lightweight yet effective scheme to detect them. Misspeculation turns out to be rare, and thus delegating the misspeculation handling to software does not impose a significant slowdown.

6. Why ASPLOS

This paper proposes a hardware-software codesign solution that uniquely addresses the ordering violation observed in hardware support for persistency models. Our solution, PMEM-Spec, identifies the ordering violation in hardware but corrects it with software support. In particular, our study showcases the highly synergistic effect of architecture, OS, and compilers for addressing the problems of prior work on persistent memory, which perfectly fits ASPLOS emphasizing multidisciplinary research.

References

[1] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA, 2014.

[2] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, SOSP, 2009.

[3] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Relaxed persist ordering using strand persistency. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture*, ISCA, 2020.

[4] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. Distributed logless atomic durability with persistent memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2019.

[5] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2020.

[6] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the European Conference on Computer Systems*, EuroSys, 2017.

[7] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[8] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the International Symposium on Microarchitecture*, MICRO, 2015.

[9] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2017.

[10] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

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

[12] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. Ido: Compiler-directed failure atomicity for nonvolatile memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2018.

[13] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2017.

[14] Matheus A. Ogleari, Ethan L. Miller, and Jishen Zhao. 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*, 2018.

[15] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the Annual International Symposium on Computer Architecuture*, ISCA, 2014.

[16] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the Annual International Symposium on Computer Architecture*, ISCA, 2017.

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