# Hands-Off Persistence System (HOPS)


## Abstract

*Programming persistent memory (PM) applications is challenging as programmers have to reason about crash consistency and use low-level programming models. To remove some of these constraints, we propose the Hands-off Persistence System to efficiently order PM updates in hardware using persist buffers. HOPS also provides high-level ISA primitives for applications to express durability and ordering constraints separately, to enable the use of ACID transactions.*


## 1. Introduction

Non-volatile memory (NVM) technologies, like NVDIMMs, phase-change-memory and others [2, 6] are increasingly attached to processors on the memory bus [9, 8]. We refer to this implementation as Persistent Memory, and it allows direct, low-latency access via regular load/store instructions to NVM.

Unfortunately, the potential of PM is marred by the programming challenges associated with it. For creating recoverable applications, programmers need to reason about the order of updates to persistent data structures. Epochs are commonly used, which are ordered groups of updates to PM which are made durable before later epochs. The low-level epoch programming model combined with manual data movement between different levels of the memory hierarchy drastically reduces programmer productivity. These also increase the likelihood of application bugs affecting recovery after a crash. Finally, the performance of PM applications is degraded from frequent, long-latency cacheline flushes to PM.

Based on an analysis of real-world PM applications, we propose the Hands-Off Persistence System (HOPS) design [7]. HOPS enables ACID transactions by supporting two distinct hardware primitives – a more common, light-weight, ordering fence (OFENCE) and the rarer durability fence (DFENCE). HOPS facilitates these primitives by tracking PM updates in hardware, and flushing in the background of program execution. Our evaluation shows that with HOPS, applications can achieve a 20% speedup over current approaches.

## 2. Hands-Off Persistence System

To drive our hardware design, we leverage insights gleaned from concurrent work [7] on analyzing real-world PM applications, which we briefly summarize here.

### 2.1. Insights

We observed the following trends in our analysis of PMBench.

- In PM applications, accesses to volatile DRAM make up about 96% of all accesses. Any PM-specific additions to caches and other structures shared between PM and DRAM should not adversely impact volatile memory accesses.
- ACID transactions are made up of 5-50 epochs. Ordering guarantees suffice between most epochs, and durability is only needed at transaction commit.
- Epochs from different threads rarely conflict with each other. Thus, in the common case, ordering and durability can be ensured locally, although inter-thread conflicts need to be handled for correctness.
- There are frequent conflicts between epochs from the same thread. Such conflicts lead to flushing on the critical path, as dirty cachelines from older epochs have to be flushed out to avoid reordering epochs.

Lastly, commercial hardware only supports primitives for flushing cachelines (e.g., `clwb` in x86-64). Thus, programmers have to conscious of data layout across cachelines.

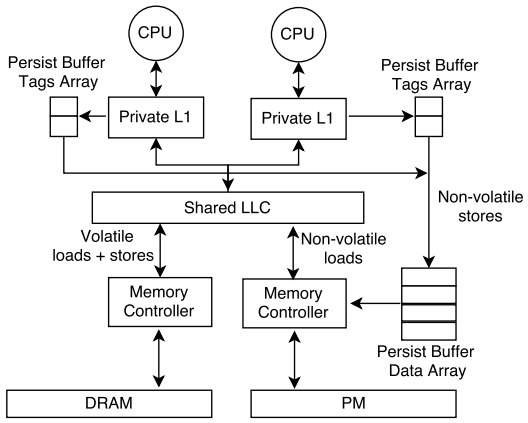


**Figure 1: Persist Buffer Design to track and order persistent writes**

### 2.2. Design Details

HOPS is composed of hardware extensions in the form of Persist Buffers and two ISA primitives—*OFENCE* and *DFENCE*. Persist Buffers (Figure 1) transparently persist PM updates while enforcing write ordering as per programmer-specified constraints. The *OFENCE* primitive enables Buffered Epoch Persistency (BEP) [3], allowing multiple epochs to be buffered in volatile structures, while *DFENCE* provides durability guarantees when needed (e.g., ACID transactions).

**ISA primitives.** HOPS separates but supports both ordering and durability ISA primitives — Ordering FENCE (*OFENCE*) and Durability FENCE (*DFENCE*). These primitives are based on the Persist/Sync Barriers [3, 4] used to demarcate software epochs. *OFENCE* orders stores preceding it before later stores, while *DFENCE* makes the stores preceding it durable. The

former is implemented as an asynchronous flush of buffered PM stores, and the latter as a synchronous flush. Both are handled locally in the absence of the rare cross-thread dependencies. Programmers can use *OFENCE* to end the current epoch and begin a new epoch, and use *DFENCE* at the end of transactions or before performing irreversible I/O operations.

Writes from different threads are made persistent in the background of execution (except on a *DFENCE*) and in a concurrent manner. Two writes are only ordered if they belong to different epochs from the same thread, or RaW and WaW dependencies exist between their epochs on different threads. This ordering is enforced by the Persist Buffers (PBs).

**Persist buffers.** HOPS orders and persists buffered PM updates in hardware to allow programmers to incorporate crash-consistency in their applications easily. Ordering is tracked in per-thread PBs, as it is more commonly enforced between epochs from the same thread. Each PM store updates the PB at the core in addition to the L1 cache. This redundancy allows caches to service any data reuse, while keeping additional state and complexity for tracking writes out of the caches. However, the modified data is only written to PM by the PBs, and is dropped by the LLC on eviction.

The PBs rely on *OFENCEs* for intra-thread ordering, and monitor coherence activity for recording any cross-thread dependencies for the buffered epochs. Self-dependencies are also handled without flushing as the PBs are multi-versioned. Thus, PBs can buffer multiple updates to the same PM address from different epochs from a thread. Finally, buffered updates are flushed concurrently while respecting these orderings.

As an example, consider the following code sequence:

```
mov A, 10 ; orderPB
mov A, 20 ; durable PB
```

The first store to A brings the cache line into the L1 cache, updates the cached value of A to 10 and creates an entry in the thread's PB of `{epoch:1, Address:A, value:10}`. When *orderPB* executes, it marks the start of a new epoch. The second store to A updates the cached value and creates an entry in the PB with `{epoch:2, Address:A, value:20}`. Finally, the *durablePB* waits for the PB to drain. The PB writes the value 10 address A in PM and when it receives an ACK from the memory controller that the update is durable, the PB writes 20 to address A. When the second ACK reaches the PB, the *durablePB* completes.

### 2.3. Evaluation

We evaluate the performance benefits of HOPS using the gem5 micro-architectural simulator [1]. Our simulated system is a four-core, 8-way, out-of-order x86 processor with 32 entry PBs, two-level cache hierarchy and two memory controllers. A subset of applications from WHISPER [7] were simulated.

We compare HOPS to the current x86-64 approach of using `clwb` and `sfence` instructions to persist data, and to an ideal implementation (Figure 2). Our ideal implementation obviates

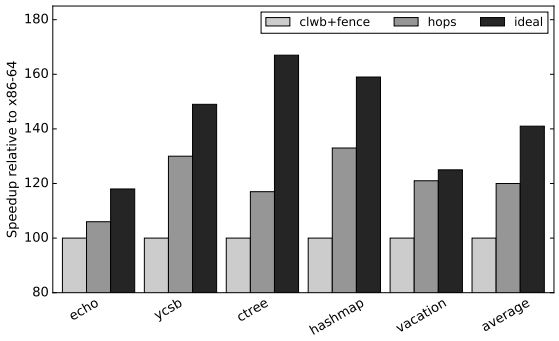


**Figure 2: Performance of HOPS Persist Buffer relative to existing writeback and fence instructions.**

`clwb` and `sfence`, thus ignoring all order between PM writes and is not crash-consistent. This allows the ideal case to improve performance by 40% compared to x86-64. HOPS shows a 20% speedup on average compared to the x86-64 approach. This improvement comes from moving most flushes off the critical path of execution. As such, the individual speedups observed are proportional to the frequency of PM accesses and flushes in our workloads.

## 3. Related work

There have been prior proposals for facilitating PM accesses in hardware. Efficient Persist Barriers (EPB) [3] provide lightweight epoch ordering and handle inter-thread dependencies. However, EPB adds state proportional to the number of cores and inflight epochs to the cache tags. Delegated Persist Ordering (DPO) [5] is concurrent work which also orders PM updates in hardware. Neither proposal provides durability primitives, preventing the implementation of ACID transactions. Also, DPO is optimized for systems with one memory controller (MC), and does not scale well to multiple MCs. HOPS is designed to scale to multiple MCs easily, particularly due to the BEP model and PB design.

## References


[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, Muhammad S., N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, August 2011.

[2] Hewlett Packard Enterprise. Hpe persistent memory. http://www.hpe.com/us/en/servers/persistent-memory.html.

[3] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. MICRO-48, NY, USA, 2015.

[4] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. ASPLOS '16.

[5] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. MICRO-49, 2016.

[6] Micron. Breakthrough nonvolatile memory technology. https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[7] S. Nalli, S. Haria, M. M. Swift, M. D. Hill, H. Volos, and K. Keeton. How applications use persistent memory. ASPLOS '17. "http://research.cs.wisc.edu/multifacet/papers/asplos17_using_pm.pdf".

[8] S. Pelley, P. M. Chen, and T.F. Wenisch. Memory persistency. ISCA'14.

[9] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. ASPLOS '11.