An Analysis of Persistent Memory Use with WHISPER [*]

Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos[†], Kimberly Keeton[†]

University of Wisconsin-Madison, USA [†]Hewlett Packard Labs (HP Labs), USA

{sankey,swapnilh,markhill,swift}@cs.wisc.edu, {haris.volos, kimberly.keeton}@hpe.com

Abstract

Emerging non-volatile memory (NVM) technologies promise durability with read and write latencies comparable to volatile memory (DRAM). We define Persistent Memory (PM) as NVM accessed with byte addressability at low latency via normal memory instructions. Persistent-memory applications ensure the consistency of persistent data by inserting ordering points between writes to PM allowing the construction of higher-level transaction mechanisms. An *epoch* is a set of writes to PM between ordering points.

To put systems research in PM on a firmer footing, we developed and analyzed a PM benchmark suite called WHISPER that comprises ten PM applications we gathered to cover all current interfaces to PM. A quantitative analysis reveals several insights: (a) only 4% of writes in PM-aware applications are to PM and the rest are to volatile memory, (b) software transactions are often implemented with 5 to 50 ordering points (c) 75% of epochs update exactly one 64B cache line, (d) 80% of epochs from the same thread depend on previous epochs from the same thread, while few epochs depend on epochs from other threads.

Based on our analysis, we propose the Hands-off Persistence System (HOPS) to track updates to PM in hardware. Current hardware design requires applications to force data to PM as each epoch ends. HOPS provides high-level ISA primitives for applications to express durability and ordering constraints separately and enforces them automatically, while achieving 24.3% better performance over current approaches to persistence.

CCS Concepts **• Information systems → Storage class memory**

Keywords Persistent memory (PM); Non-volatile memory (NVM); Storage-class memory; Caches; Benchmark

1. Introduction

Persistent memory (PM) has received significant attention in software research [9, 18, 21, 39], hardware research [26, 36, 43], and industry [5, 23, 25, 37]. We define PM as non-volatile memory (NVM) accessed with byte addressability (not just blocks) at low latency (not I/O bus) via regular memory instructions (not system calls). Prior PM research studied either existing applications targeting traditional volatile memory systems and disk storage technologies or micro-benchmarks developed in isolation that exercise only a specific mechanism. Although this has been a great first step, it is challenging to compare various PM systems and be confident that proposed systems are optimized for actual use.

The time is ripe to consider a comprehensive benchmark suite that captures important and unique properties expected in PM applications. First, PM applications are expected to store data directly in PM for fast persistence in addition to accessing data through more traditional block-based filesystem and database interfaces that offer other useful functionality such as sharing and global naming. Second, heterogeneous memory systems—which contain volatile DRAM and NVM—will likely be the dominant model for main memory in the near future [22] due to the performance, reliability and cost of candidate NVM technologies. This organization precludes automatically making *all* memory persistent [34, 36], and instead requires applications to selectively allocate data in PM and ensure its consistency and durability. As a result, applications will likely include both volatile and persistent memory data structures, thus exhibiting a mix of memory traffic. Finally, PM applications have to be crash-recoverable. In contrast to volatile memory programs, they have to carefully order and persist writes to memory with respect to failures. At a low level, these properties are ensured by (i) explicitly writing data back from the processor cache to PM, (ii) enforcing ordering between writes to dependent structures [15], and (iii) waiting for data to become durable in PM before continuing execution. The set of writes between ordering points constitute an *epoch* [18, 35] and can reach PM in any order, as long as they are durable before *any* writes from following epochs are durable. Software can implement persistent transactions with ACID semantics using the ordering guarantees of epochs [28].

[*] WHISPER stands for Wisconsin-HP Labs Suite for Persistence

ASPLOS '17 April 8–12, 2017, Xi'an, Shaanxi, China

© 2017 ACM. ISBN 978-1-4503-4465-4/17/04. . . $15.00

DOI: http://dx.doi.org/10.1145/3037697.3037730

This paper seeks to put PM systems research on a firmer footing by developing, analyzing and releasing a PM benchmark suite called WHISPER: Wisconsin-HP Labs Suite for Persistence. It comprises ten PM applications we gathered to cover a wide variety of PM interfaces such as databases, in-memory data stores, and persistent heaps. WHISPER covers applications that access PM directly, those using a PM transactional library such as Mnemosyne [39], and those accessing PM through a filesystem interface. We modified the applications to be recoverable and instrumented them for our own and future analysis. WHISPER is available at `research.cs.wisc.edu/multifacet/whisper`.

PM analysis. A trace-based and quantitative analysis of WHISPER yields several behavioral characteristics and insights into the design of future PM Systems. On average, (a) only 4% writes in PM-aware applications are to PM and the rest are to volatile memory, (b) software transactions are often implemented with 5 to 50 ordering points and require durability only after the last ordering point, (c) 75% of epochs update exactly one 64B cache line, not necessarily the same line (d) 80% of epochs from the same thread depend on previous epochs from the same thread while few epochs depend on epochs from other threads. These observations suggest there is value in handling PM writes in a special structure—that does not disturb the writeback caches used for the 96% of volatile accesses—but one that supports multiple writes to the same lines from different epochs.

HOPS design. Following our analysis, we propose the Hands-Off Persistence System (HOPS) design. HOPS realizes the special structure mentioned above with logically per-thread Persist Buffers (PBs) that (i) support multiversioning (same line stored from different epochs) (ii) separate a more common, light-weight ordering barrier (`ofence`) from a less common, heavy-weight durability barrier (`dfence`), and (iii) distribute state to per-thread/core structures for scalability to larger processors. Our results indicate that HOPS improves application performance by 25% over current approaches to persistence.

The rest of this paper is as follows. Section 2 presents various programming models for PM. Section 3 discusses the applications comprising WHISPER. Section 4 describes our methodology. Section 5 reports our results and insights into PM application behavior. Section 6 concludes with a description of HOPS and its evaluation.

2. Background

Three central challenges exist in programming for PM. First, data is only durable when it reaches PM; data in volatile processor caches are currently lost on a power failure. As a result, applications that require durability must ensure that data leaves the cache and wait for it to reach PM. Second, system crashes including power failure amidst persistent data structure modifications may result in an inconsistent state on recovery. Finally, write-back processor caches can re-order updates to PM, implying that even ordered updates may reach PM out of order.

Although the terms non-volatile memory (NVM) and persistent memory (PM) are often used interchangeably, we differentiate between the two in this paper. NVM refers to a class of memory technologies that exhibit near-DRAM access latency, and preserve data across power failures. We define PM as NVM accessed with byte address-ability (not just blocks) at low latency (not I/O bus) via user-mode CPU memory instructions (not system calls).

Programming persistent memory. We classify existing software programming models for PM into three broad categories—Native Persistence, Library Persistence, and Filesystem Persistence. Figure 1 shows a simple example programmed in different styles, with the objective that the update to *flag* is never made durable before the update to *pt*.

Native persistence. Applications can be written to ensure consistent updates to PM by writing data to PM and waiting for it to become durable. For this purpose, the x86-64 architecture specification now includes the `clflushopt` and `clwb` instructions [19] that flush or writeback a specific line from the cache hierarchy to memory[1]. A subsequent `sfence` instruction stalls the thread until all its outstanding flushes and writebacks are complete. The instruction sequence `clwb A; sfence` guarantees that when the `sfence` completes the data in cache line A will survive a crash. Applications can also use non-temporal instructions (NTIs) to bypass the cache and write directly to PM; the fence is still required to ensure durability and ordering by waiting for write-combining buffers (WCB) to drain. Programmers use these operations directly by moving data to PM for durability and ordering updates for consistency as needed. Figure 1(a) shows an example of native persistence.

There are two drawbacks to this model. First, it conflates ordering with durability. In Figure 1, it may be sufficient to ensure that the update to object *pt* is ordered before the flag update, while durability is not needed until later, at the end of the code sequence [15]. Currently, x86-64 does not provide an instruction to express order of updates to PM, so the *pt* update must be durable before initiating the flag update. These cache flushes are long-latency operations and occur in the foreground of program execution. Using careful programming, it is possible to overlap some of the flushes but this optimization is case-specific.

Second, this approach forces the programmer to reason about the layout of application state in memory at cache line granularity. If an object spans multiple cache lines, as does *pt*, the programmer must flush each individual cache line, and update this code sequence if the object layout changes. This assembly-language style of programming may result in buggy code and decrease developer productivity.

[1] Intel proposed the `pcommit` instruction to flush data from memory-controller buffers, but has deprecated it. Intel now requires platform support to flush memory-controller buffers on a power failure [20].

```
pt->x = 1
pt->y = 1
clwb(&pt->x)
clwb((&pt->y)
sfence

flag = 1
clwb(&flag)
sfence
```

(a)

```
TX_BEGIN {
    pt->x = 1;
    pt->y = 1;
    flag = 1;
} TX_END
```

(b)

```
pt->x = 1
pt->y = 1
pmem_persist(&pt,
    sizeof(pt))

flag = 1
pmem_persist(&flag,
    sizeof(flag))
```

(c)

```
pwrite(fd, &pt,
    sizeof(pt),
    offset_pt)

pwrite(fd, &flag,
    sizeof(flag),
    offset_flag)
```

(d)

```
pt->x = 1
pt->y = 1
ofence

flag = 1
dfence
```

(e)

Figure 1. Persistent memory programming models. The diagram shows five ways of updating a persistent structure *pt* that contains two variables *x* and *y*. Setting a persistent *flag* indicates a successful update to pt. *x* and *y* do not occupy the same 64B cache line. Updates to *x* and *y* can be re-ordered with respect to each other but must happen before the *flag* is set, to preserve crash consistency. (a) Native persistence (b) Library persistence—Durable transactions (c) Library persistence—Atomic updates (d) Filesystem persistence (e) Our proposal HOPS with ordering and durability primitives.

Library persistence. The task of enforcing consistent updates to PM can be delegated to libraries like Mnemosyne, NV-Heaps, and NVML [17, 39]. Libraries provide useful functionality such as memory allocation, memory leak avoidance, type safety, durable transactions and atomic updates. These libraries provide a transaction interface, shown in Figure1(b), that provides atomic and durable updates of multiple objects. Figure 1(c) illustrates how libraries provide an atomic update operation that persists a value atomically. This interface frees programmers from the burden of manual data movement. However, the general-purpose nature of these libraries can preclude low-level software optimizations and results in conservative ordering constraints. For example, atomic transactions may not be needed for some data structures, such as an append-mostly log or copy-on-write trees [18]. Additionally, the persistent metadata maintained by these libraries for consistency can amplify the number of updates to PM. The use of flush/write-back/fence instructions in these systems can also result in the same performance degradation seen with native persistence.

Filesystem persistence. Legacy applications written for file systems can gain the performance benefits of PM by using PM-aware file systems such as PMFS, BPFS, NOVA [18, 21, 41] and DAX-based filesystems on Linux such as {ext2, ext4, XFS}-DAX. These filesystems bypass the operating system block layer and directly update data and metadata in PM. Hence they can provide stronger reliability and consistency guarantees than traditional file systems by persisting data synchronously, rather than asynchronously as done for hard drives and SSDs. Figure 1(d) shows a program issuing system calls to the file system to persist data.

3. WHISPER

We have assembled and made publicly available [`research.cs.wisc.edu/multifacet/whisper`] a new benchmark suite of ten PM applications, WHISPER—Wisconsin-HP Labs Suite for Persistence—that captures various proposed software layers for accessing PM including native, heap libraries and filesystems. Table 1 summarizes the applications, how they access PM, our driving workloads, and their intensity of epoch use. Three characteristics of WHISPER are essential for guiding hardware and software design for PM.

- WHISPER includes a mix of real-world applications and micro-benchmarks exploiting the properties of PM. Using full programs generates realistic memory traffic to both PM and DRAM. We include both legacy and newly written programs, including popular industrial systems such as Memcached, Redis, NFS and the Exim mail server. For simulator-suitable studies, we include two micro-benchmarks whose memory access patterns are representative of larger workloads.
- WHISPER includes crash-recoverable applications, which means that they persist all information in PM that is necessary to recover after a crash. For example, we modified Vacation from the STAMP suite [33] to persist its data structures, to observe the true cost of persistence.
- WHISPER assumes heterogeneous memory as noted in the introduction. We modified Vacation, N-store and Echo, which originally assumed all memory is persistent, to selectively place their data structures in PM.

3.1 Access Layers

We leverage three existing PM systems to act as access layers for the applications: two transactional PM libraries, Mnemosyne [39] and Intel's NVM library NVML [5], and a PM-aware filesystem, PMFS [21]. We also include two applications that directly interact with PM: the N-store SQL database [9] and the Echo NoSQL key-value store [10].

Libraries. *Mnemosyne* and *NVML* are x86-64 libraries that provide access to PM via durable transactions. We used publicly available NVML v1.0. We ported Mnemosyne to GCC, as it was originally written for the now obsolete Intel TM compiler. These expose PM through memory-mapped segments that are broken up into objects by a persistent allocator. Both libraries provide transactions to consistently

Benchmark	Access Layer	Workload/Configuration	Epochs per second
Echo	Native	*echo-test* / 4 clients, 1 million transactions	1.6 million
N-store	Native	*YCSB like* / 4 clients, 8 million transactions, 80% writes.	5 million
		TPC-C like / 4 clients, 400K transactions, 40% writes	7.3 million
Redis	Library/NVML	*redis-cli* / lru-test, 1 million keys	1.3 million
C-tree	Library/NVML	4 clients, 100K INSERT transactions	1 million
Hashmap	Library/NVML	4 clients, 100K INSERT transactions	1.3 million
Vacation	Library/Mnemosyne	4 clients, 2 million transactions, 16 million tuples	700K
Memcached	Library/Mnemosyne	*memslap* / 4 clients, 100K ops, 5% SET	1.5 million
NFS	FS/PMFS	*filebench* / 8 clients, 8 NFS threads, *fileserver* profile	250K
Exim	FS/PMFS	*postal* / 8 clients, 100 KB msgs, 1000 msgs/min, 250 mailboxes	6250
MySQL	FS/PMFS	*OLTP-complex* / *sysbench*, 4 clients, 1 table of 10 million tuples	60K

Table 1. WHISPER Applications. The rightmost column in the table shows the number of epochs per second in each application under the workload in the third column.

and atomically update arbitrary data structures, such as hash tables, stored in the segments. A key difference between the two libraries is that Mnemosyne *automatically* detects and logs all updates to a persistent object within a transaction, while NVML has to be informed of such updates, unless the object was allocated in the same transaction.

Crash consistency: Mnemosyne achieves consistency of data structures via a redo log. It updates the log using non-temporal instructions (NTI) ordered by an `sfence`. It saves modified data to a temporary location, and at transaction commit uses cacheable stores to update data structures followed by flushing modified cache lines to persist updates. It provides APIs to atomically allocate and free persistent objects (*pmalloc()* and *pfree()*). NVML achieves consistency of data structures via an undo log. It uses cacheable stores/flushes to execute all log and data updates to PM and provides APIs to atomically allocate and free objects in PM (*pmemobj_tx_alloc()* and *pmemobj_tx_free()*).

File system. PMFS is a Linux filesystem for x86-64 that provides access to PM via system calls. Although deprecated, it is the most functional filesystem for PM to date and is representative of other research prototypes like BPFS [18] and SCMFS [40]. It exposes PM using files, and persists user data and filesystem metadata synchronously. Most other filesystems persist state asynchronously and thus can lose recent data on a crash.

Crash consistency: PMFS stores user data in 4KB blocks and metadata in persistent B-trees. It employs an *undo log* to ensure metadata consistency and uses cacheable stores for metadata related updates, and flushing and fencing instructions for consistency. It does not guarantee consistency of user data, which is updated by NTIs that bypass the cache followed by an `sfence`. This design assumes that user data in files has low temporal locality.

3.2 WHISPER Applications

3.2.1 Native Applications

WHISPER includes two applications that persist their data structures in PM using custom transactions.

N-store [9] is a RDBMS for PM inspired by the design of H-store [27]. It models the database (DB) as partitions of tables and each DB thread executes transactions on a single partition independent of others. Each tuple in a table is an array pointing to one primary key and a number of attributes of varying sizes. Among the six back-end engines in N-store, we chose the optimized write-ahead log (OPTWAL) engine. OPTWAL directly interacts with PM without relying on a filesystem to persist data, making it the fastest of the engines.

Crash consistency: OPTWAL maintains a doubly linked list of PM segments, allocated per thread by a global PM allocator. OPTWAL places tables and indexes in these segments and uses an undo log to atomically update them. It updates logs, tables and indexes using cacheable stores and flushes them from the cache, using fences to enforce ordering.

Modifications: N-store originally assumed a homogeneous memory system in which all memory is persistent. We modified it to place tables, indexes and logs in PM, keeping thread stacks and heap in DRAM. This required changes to 77 LOC. We ensure that all updates to PM occur atomically in durable transactions. To illustrate the behavior across different workloads, we used simple implementations of TPC-C and YCSB, shipped with N-store.

Echo [10] is a scalable key-value store (KVS) for PM. It employs a master thread to manage the persistent KVS while client threads batch and send updates to KV pairs to the master. Each client thread contains a volatile KVS similar in structure to the master, which it uses to service local reads, and finalize and batch updates, making Echo scalable.

Crash consistency. The master KVS is a persistent hash table. Each hash table entry is a key and a chronologically ordered list of versions of a value. Client submit updates to key-value pairs, which are stored in a persistent log. After a successful submission, the master processes the log and moves the updates to its persistent KVS in PM.

Modifications: We modified Echo to use the persistent memory allocator from N-store. We instrumented and ensured all updates to the heap occur in durable transactions. This required changes to 80 LOC.

3.2.2 Library-based Applications

WHISPER includes three object stores using transaction libraries to store and access key-values pairs in PM. We modified Memcached, and Vacation from the STAMP suite to access PM via Mnemosyne, and used Redis modified to access PM via NVML.

Memcached [24] is an in-memory key-value store used by web applications as an object cache between the application and a remote object store. It stores objects in a hash table and an LRU replacement policy.

Modifications: We modified Memcached to allocate the hash table in PM segments, ensured that all accesses to PM execute atomically in durable transactions, and replaced all locks used for synchronizing concurrent access to the table with transactions. This required changes to 17 LOC.

Vacation [13] is an OLTP system that emulates a travel reservation system. It implements a key-value store using red black trees and linked lists to track customers and their reservations. Several client threads perform a number of transactions to make reservations and cancellations.

Modifications: We modified Vacation to allocate red black trees and linked lists in PM segments using Mnemosyne and ensured that all accesses to PM execute atomically in a durable transaction. During this process, we fixed many stray updates in Vacation that altered PM non-atomically, leading to the possibility of an inconsistency.

Redis [14] is a REmote DIctionary Service used by web applications like Twitter as an in-memory key-value store. It stores frequently accessed key-value pairs in a hash table and resolves collisions through chaining. It uses a single-threaded event programming model to serve clients.

Modifications: We borrowed a partially recoverable version of Redis from a third party source [6] that was modified to store string keys and values in a hash table allocated in PM using NVML. We ensured that all accesses to PM execute atomically in a durable transaction.

C-tree and Hashmap are multi-threaded micro-benchmarks written for NVML that perform inserts and deletes operations into a persistent crit-bit tree or a hashmap [1]. These benchmarks are part of the examples shipped with NVML.

3.2.3 Filesystem Applications

WHISPER includes three common applications to store and access files in PM using PMFS. These applications are unmodified popular open-source programs.

NFS is an in-kernel Linux server and client that provides remote access to a filesystem. We exported a PMFS volume using NFS and executed the *fileserver* profile from filebench [38] to act as a remote application.

Exim [2] is a mail server. For each connection, a master process spawns three child processes that receive the mail, append it to a per-user mail box, and log the delivery.

```
/* Update undo log size in PMFS */
       PM_SET(journal->size,                \
                   cpu_to_le32(size));
---
/* Write log entry in Mnemosyne */
#define asm_movnti(addr, ...)               \
       PM_MOVNTI(addr, sizeof(pcm_word_t), \
                   sizeof(pcm_word_t));     \
       PM_FENCE();
---
/* Set tuple values in N-store */
void set_varchar(vc_str, ...) {
        PM_STRCPY((vc), vc_str.c_str(),    \
                    (vc_str.size()+1));    \
        PM_MEMCPY((&(data[sptr->columns    \
                    [field_id].offset])), \
                    (&vc),(sizeof(char*)));
---
/* Flush CPU cache in NVML */
void flush_clflush(addr, ...) {
       PM_FLUSH((addr), (64), (64));
```

Figure 2. WHISPER instrumentation. PM_* macros emit a trace of PM updates and fences for offline analysis.

MySQL [4] is one of the most widely used RDBMSs, often used for online transaction processing. We ran OLTP-Complex workload from Sysbench [8].

4. Methodology

Our goal is to characterize the write behavior of WHISPER applications. Ensuring consistency of writes is the dominant cost of making data persistent. WHISPER includes a trace framework that records PM updates, hardware barriers, cache flushes, and transaction *begin* and *end* events performed during application execution. Our tracing framework incurs 2-10x overhead, depending upon the rate of PM accesses by the application, mostly for writing out the trace.

Identification: We identified code that performs PM accesses in each application. For user-space code, we used Intel's PIN tool and found over 100 statements in each code base. As PIN cannot be used for kernel filesystem code, we modified *mmiotrace* [3]—a memory-access tracer for device drivers in Linux—to identify all statements that perform PM accesses in the kernel. There are 265 statements in PMFS and ten in the rest of the kernel.

Instrumentation: We annotate all PM operations in our applications to enable a variety of analyses beyond those in this paper. We designed C macros (PM_*) to capture all modes of updating PM and emit a trace of PM addresses accessed during a transaction or system call. The size of the trace is limited only by storage capacity. We identify transaction start and end events by instrumenting routines in access layers or program code that indicate these events. For our analysis, these macros generate trace entries captured by *ftrace* [7], a function tracer in Linux. A key benefit of our framework is that it can trace and record PM accesses from user space and kernel code on all processor architectures.

Execution: We execute WHISPER applications on an Intel i7-6700K Skylake Processor running at 4 GHz. It contains 4 cores, each with 2 hardware threads, and 8GB of DRAM,

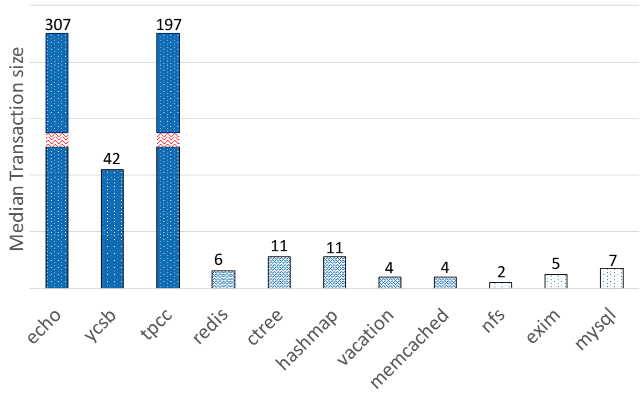

Figure 3. Distribution of transaction sizes. The size of a transaction is the number of epochs or ordering points in the transaction. Echo and N-Store TPC-C have well over a hundred epochs per transaction.

of which we reserved and used 4GB as PM. All applications ran to completion (roughly two minutes) and produced a trace of persistent operations along with a timestamp for each operation using a global clock provided by *ftrace*.

5. Persistent-Memory Application Behavior

In this section we first present an analysis of the epoch-level behavior of all ten applications followed by a discussion of cross-cutting issues. We then analyze the relationship between DRAM and PM accesses based on a simulation study for a subset of WHISPER applications.

5.1 Epoch-level Behavior

We consider an epoch to consist of stores, whether cacheable or non-temporal, to PM between two `sfence` instructions. For this analysis, we ignore cache flush operations. Higher-level consistency mechanisms such as transactions are built using epochs, to order their writes and perform consistent updates to PM. We identify transaction boundaries from the program or library code.

Epochs per transaction. Most durable transactions had between 5 and 50 epochs, although in some cases, like Echo and TPC-C with N-store, there were well over a hundred. Figure 3 gives the median of the number of epochs per transaction. Native PM applications and TM libraries showed the highest rate of epochs while filesystem applications had the lowest (Table 1). Importantly, transactions do not require durability until they commit, which is generally the last epoch. Thus, enforcing durability for every epoch, as done in current hardware, is an overkill. Our results show that current software is far from an ideal high-performance transaction modeled by Kolli *et al.* [28] as containing just 3 epochs.

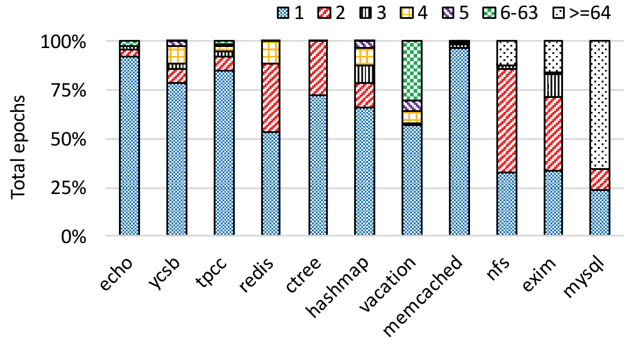

Figure 4. Distribution of epoch sizes. The size of an epoch is the number of unique 64B cache lines it stored to PM.

CONSEQUENCE 1. *Epoch implementations should separate ordering requirements for epochs from durability requirements for transactions.*

CONSEQUENCE 2. *Epoch implementations should be fast, as epochs are much more common than transactions.*

Epoch size. Figure 4 shows the distribution of epoch sizes for WHISPER applications. On average 75% of epochs in native and library-based applications were *singletons*, i.e., they were just one 64B cache line in size. 25% of epochs had sizes varying between 2 and 63 PM cache lines. In contrast, on average 30% of epochs in PMFS update a single cache line, 30% update two cache lines, 30% update 64 cache lines while the remaining 10% update anywhere between 3 and 63 cache lines. We see large epochs in PMFS because it updates 64 cache lines when writing a file-system block of 4KB. We observed that some applications (N-store and those using NVML) sometimes modify data in one epoch and flush it in another. This occurs with undo logging: fencing after writing an undo record may add unflushed data writes to an epoch, with the flushes performed at transaction commit.

While fast PM encourages fine-grained writes, we observed the dominant cause of small epochs was *not* application data but metadata writes from memory allocation and logging. All our PM applications contain log operations for recovery at cache line granularity. Mnemosyne, NVML and PMFS process or clear each log entry in its own epoch, which contributes to the large number of singletons. This could be avoided without compromising crash consistency by processing or clearing log entries in a batch. Undo logs, used in PMFS and NVML, exhibit more small epochs than redo logs: undo entries must be ordered before data writes to ensure the old value is available for recovery, and thus they fragment a transaction into a series of alternating epochs to write log entries and to update data. Redo logging, in contrast, allows batching by writing back all log entries in one epoch and then writing back data in another epoch.

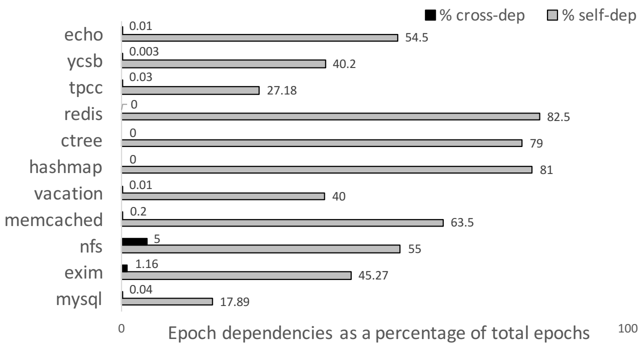

Figure 5. Epoch dependencies. The diagram shows epochs exhibiting self and cross dependencies within 50 μsec as a fraction of total epochs in application execution. Self-dependencies are abundant, while cross-dependencies are rare.

CONSEQUENCE 3. *Epoch implementations can optimize for singleton epochs for better performance.*

Of the singletons, we saw that 60% updated fewer than 10 bytes. As we discuss in more detail in the next section, many of these writes are from the user-space persistent memory allocator, which uses only a few bytes to update the state of the allocator. Log metadata (e.g., log descriptors in Mnemosyne) also contributed many of the small writes.

CONSEQUENCE 4. *Epoch implementations should optimize for byte-level persistence.*

Cross- and Self-dependencies. Epoch dependencies in PM applications arise when application threads either update shared persistent data, or metadata to preserve crash consistency of persistent state. For our analysis, we measure and study write-after-write (WAW) dependencies to PM addresses. To simplify trace processing, we only look for dependencies within a 50 μsec window, which is the upper limit for which a flushed cache line could be buffered before becoming persistent. For clarity, we define:

- E_i^m is a set of cache lines updated by thread *i* during the *m-th* epoch.
- *Cross-dependency*: $E_i^m \otimes_c E_j^n$ denotes *m-th* and *n-th* epoch on threads *i* and *j*, respectively, that write to cache line *c*, where E_j^n follows E_i^m in the order of execution.
- *Self-dependency*: $E_k^m \ominus_c E_k^{m'}$ denotes epoch *m* and *m'* on thread *k* that write to cache line *c*, where $E_k^{m'}$ follows E_k^m in the order of execution.

Applications had a small fraction of epochs with cross-dependencies—5% for NFS, 1.16% for exim, and less for the rest. In contrast, Figure 5 shows that there is wide variation in the occurence of self-dependencies across applications. The fraction of epochs with self-dependencies varies between 25% and 55% for native applications (TPC-C, YCSB, Echo), but increases to 80% for NVML-based Redis, ctree and hashmap applications. Mnemosyne-based applications exhibit a moderate amount of self-dependencies. MySQL shows the lowest amount of self-dependencies as it has few metadata writes, which are the primary cause of self-dependencies with PMFS.

We found three sources of both cross- and self- dependencies: (i) applications writing the same data repeatedly, such as shared persistent variables, (ii) transaction metadata, and (iii) the persistent memory allocator. At the application level, Vacation has global counters of the number of cars/flights/rooms which are updated in transactions, leading to cross-dependencies. Similarly, Echo initializes a descriptor associated with its data structures and alters its status from INPROGRESS to CREATED, using two consecutive epochs in a thread that writes the same cache line.

Transaction metadata self-dependencies occur in PMFS, NVML and Mnemosyne, when they initialize the status and contents of the log to prepare it for use. For example, NVML sets and clears its log entries and PMFS alters the status in the log descriptor from UNCOMMITTED to COMMITTED after a successful commit.

Memory allocator self- and cross-dependencies occur in the single-slab allocators of PMFS, Echo and N-store when they recycle PM blocks across and within threads. Multiple-slab allocators, such as in Mnemosyne, are also susceptible to dependencies, although less frequently. Applications using PM as a scalable DRAM may require additional epochs to label a block as either persistent or volatile. For example, N-store allocates both volatile and persistent data from a persistent heap, and decides later which objects should persist across crashes by storing a state variable with each block—FREE, VOLATILE or PERSISTENT. Transactions that alter the state of a block write to this variable thrice cause self-dependencies in N-store.

Dependent epochs across threads require that writes from one thread must be persisted after those from another thread; if this is not done correctly then data may be inconsistent after a failure. This can occur, for example, in a producer-consumer situation where a consuming epoch becomes durable before the producing epoch.

CONSEQUENCE 5. *Cross-dependencies exist and must be handled correctly, but are uncommon.*

Within a thread, repeated writes to a cache line require either that the thread wait for an earlier write to become durable before updating it, which is slow, or that multiple copies of the cache line exist simultaneously. This contrasts with the standard use of volatile memory, where fast caches encourages re-accessing the same cache line.

CONSEQUENCE 6. *To gracefully handle self-dependencies, processors should allow multiple versions of a cache line from different epochs to be buffered simultaneously to avoid* *stalling while data from an earlier epoch is written back to NVM.*

CONSEQUENCE 7. *Applications should avoid data structure designs that repeatedly write to the same persistent data across epochs, with different allocation policies (i.e., not LIFO) and object layouts.*

5.2 Cross-cutting Behavior

In addition to epoch behavior, we also investigated other cross-cutting behaviors in our benchmark applications.

How does memory allocation affect behavior? Persistent memory allocators have an unexpectedly large impact on behavior. They are often invoked within transactions. The N-store and Echo allocators have a single heap for all allocation sizes, leading to frequent splits and coalescing of blocks, each requiring a persistent metadata write. Allocators with multiple slabs for different allocation sizes, as in Mnemosyne and NVML, store a bitmap of allocated blocks and use volatile structures to speed allocation. Furthermore, NVML's allocator guarantees atomicity, so blocks allocated during an aborted transaction are freed, but at the cost of extra epochs for logging. Mnemosyne's allocator can leak memory if a power failure occurs during a transaction, but does not create more epochs.

One approach to eliminating epochs introduced by the allocator is to do away with persistent maps that explicitly track allocated objects, as done by BPFS [18]. BPFS considers a block as allocated when there is a reference to it. Although this requires a scan of the file system structure to find free blocks, this design optimization dramatically reduces the number of ordering points and commits [16]. Further, writing to a newly allocated data block avoids the need to do either undo or redo logging—the block can simply be reclaimed on failure, if the write to it is interrupted. Language and runtime support, such as garbage collection [11] of unreachable objects after a restart, could similarly help reduce ordering points in PM applications.

CONSEQUENCE 8. *Memory allocator designs should consider relaxing guarantees or rely on other runtime support such as garbage collection to reduce the number of epochs needed to execute a transaction.*

How much write amplification occurs? We define write amplification as the number of additional bytes written to PM for every byte of user data stored in PM during a transaction. The additional bytes are incurred by recovery mechanisms such as undo and redo logs and the memory allocator. For PMFS, the amplification is 10%: for every 4096-bytes of user data appended to a file, roughly 400 additional bytes of filesystem metadata and journal are written out to PM in a transaction. PMFS does not log user data. For Mnemosyne, the amplification is between 300% and 600% for updating the persistent allocator state, which is a bitmap of free blocks. In contrast, for NVML, the amplification is 1000% because it (i) logs the allocator state in a redo log before mutating it, (ii) mutates the state after processing the redo log, (iii) sets/clears transaction undo log entries and (iv) initializes several other auxiliary data structures. Much of this cost is avoided in Mnemosyne by allowing memory to leak during a failure, leading to the need for a garbage collection mechanism. For N-store, amplification varies between 200% and 1400%, depending on the workload and operations, largely due to its PM allocator that uses a buddy system.

CONSEQUENCE 9. *PM libraries add substantial overhead in order to provide atomicity and recovery, so applications should consider whether they need all the properties offered by the libraries.*

How is PM written? Applications write to PM using cacheable store instructions followed by flushes, or uncacheable NTIs, which do not leave data in the cache. NTIs are useful for persistent data that is only needed for recovery, as in recovery logs. For example, Mnemosyne uses NTIs to write redo log entries, which are only read following a restart. PMFS avoids cache pollution when writing user data and for zeroing pages with NTIs. Overall, about 96% of writes in PMFS and 67% in Mnemosyne use NTIs. Despite important uses, some proposals for epochs [18, 26] do not discuss bypassing the cache for non-temporal stores.

CONSEQUENCE 10. *PM hardware systems should allow bypassing the cache when writing low-locality data.*

5.3 Proportion of PM Accesses

Although our trace-based analysis yielded useful results about PM usage, it can only trace PM accesses, and hence does not describe how PM and DRAM accesses relate. We evaluate the relationship between PM and DRAM accesses using the gem5 simulator [12] on the subset of benchmark applications that run well in simulation; Section 6.4 gives the full methodology for these results.

As shown in Figure 6, we observe that the majority (>96%) of accesses are to DRAM. We expect that this trend will continue for three main reasons. As access latencies to PM are much higher than DRAM, applications optimize by placing transient data structures in volatile memory. For example, volatile indexes speed allocation in Mnemosyne without increasing the amount and cost of making data persistent. Similarly, Echo maintains a local volatile store for each worker thread, which is frequently used, while the persistent master store is rarely updated. Second, most NVM technologies are expected to have limited write endurance, which is not an issue for DRAM. As a result, software techniques will be employed along with hardware optimizations to reduce PM writes. Finally, most applications need only a limited amount of data to recover from a crash. For example,

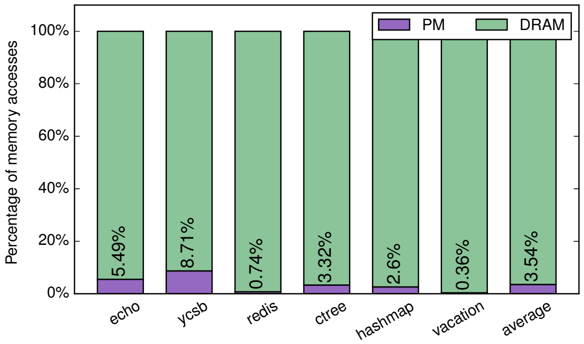

Figure 6. Proportion of PM accesses among all memory accesses. PM accesses constitute only a small fraction of total memory accesses.

checkpoints for long-running scientific workloads and save files for games have typically been kept small.

CONSEQUENCE 11. *Hardware facilitating persistent accesses should not add overheads to volatile accesses.*

6. Hands-Off Persistence System

We use the insights from our workload analysis to guide the design of hardware support for efficient PM access. Here, we assume that a specific range of physical memory is earmarked for PM. By default, loads and stores refer solely to non-volatile accesses and threads refer to hardware threads.

6.1 Goals

Based on our observations in earlier sections, we see that the following design goals facilitate efficient hardware design.

- In PM applications, accesses to 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. Hence, a standalone, lightweight ordering primitive should be supported.
- 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.
- Epochs frequently conflict with prior epochs from the same thread. Such conflicts lead to flushing on the critical path, as dirty cache lines from older epochs cannot be overwritten by newer epochs, to prevent illegal reorderings. This can be avoided by maintaining multiple versions of cache lines with some ordering information.

In addition, we seek a solution that makes data persistent without explicit flushes, removing the need for programs to be conscious of how their data is laid out across cache lines.

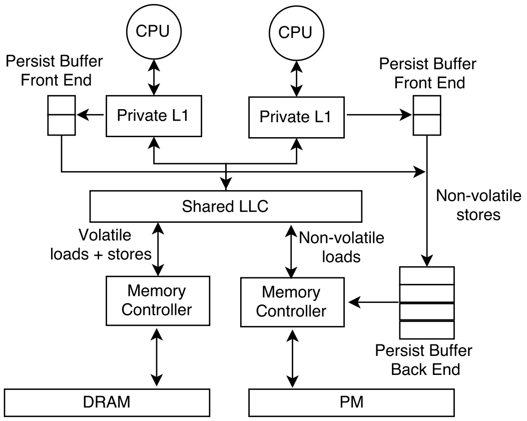

Figure 7. Persist buffer. Figure shows split persist buffer design to track and order persistent writes.

6.2 HOPS Design

We propose the Hands-Off Persistence System (HOPS) to achieve our hardware design goals. HOPS orders and persists PM updates in hardware to facilitate programming of crash-consistent applications, while allowing high-performance implementations. HOPS consists of hardware extensions, *Persist Buffers* (PB), and two ISA primitives, `ofence` and `dfence`. Persist Buffers (Figure 7) track updates to PM redundantly along with the cache hierarchy and enforce write ordering to PM as per the Buffered Epoch Persistency model (BEP) [26]. BEP enables multiple epochs to be buffered in volatile structures. The lightweight `ofence` ensures ordering between different epochs from a thread, while `dfence` provides durability guarantees when needed (e.g. ACID transactions).

ISA Primitives. HOPS independently supports both ordering and durability primitives — Ordering FENCE (`ofence`) and Durability Fence (`dfence`). These primitives are based on the overloaded Persist/Sync Barriers [26, 28] used to demarcate software epochs. `ofence` signals an epoch boundary, thereby ordering stores preceding it before later stores. `dfence` makes the stores preceding it durable. Thus, the former can be used as an asynchronous flush of buffered PM updates, and the later as a synchronous flush. Programmers can use `ofence` at the end of epochs, and `dfence` when committing ACID transactions or before irreversible I/O operations. Figure 8 demonstrates a simple undo-log based persistent transaction implemented using these primitives.

Writes from different threads are made persistent in an order determined by the synchronization of threads. The order can be inferred from RAW and WAW conflicts between epochs from different threads. In the absence of conflicts, epochs from different threads are unordered and can be flushed out simultaneously.

Persist Buffers. Persistent updates are buffered and tracked in per-thread PBs, moving long-latency PM flushes to the background of execution while preserving crash consistency.

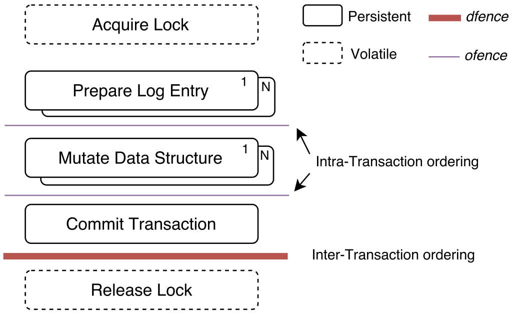

Figure 8. A persistent transaction implemented using epochs with `ofence` and `dfence`.

Each PM store updates the PB and the L1 cache. This redundancy allows caches to service data reuse, but keeps additional complexity and state needed for tracking PM writes out of the caches. The modified data is only written out to PM by the PBs, and is dropped by the LLC on eviction.

PBs rely on `ofence` for intra-thread ordering and monitor coherence activity for gleaning inter-thread ordering. Intra-thread ordering is facilitated by BEP as updates only need to be tracked at epoch granularity. To handle the frequent self-dependencies without flushing, multiple versions of a cache line are allowed to exist in the PBs, with only the latest value present in the volatile caches. For maintaining inter-thread ordering, cross-thread dependencies between buffered epochs are preserved. A cross-thread dependency is (conservatively) identified based on the loss of exclusive permissions to a cache line by an L1 cache. The thread acquiring exclusive permissions is provided information about the source thread and epoch to allow it to enforce this dependency when the dependent epoch is being flushed. Epoch deadlocks are prevented by splitting epochs, as demonstrated previously [26].

Finally, HOPS minimizes the performance degradation of flushing buffered updates, especially with multiple memory controllers (MCs). In the absence of cross-dependencies, epochs from different threads are unordered as are writes from a thread belonging to the same epoch. Such writes are flushed concurrently to the MCs, thereby sustaining high performance with small-sized PBs.

6.3 HOPS Implementation

We implement the persist buffers using a split design to minimize the use of valuable silicon real estate near the cores. The *PB Front End* contains the metadata for each buffered update (address, epoch TS, dependency pointer) and is located near the private L1D cache. The PB front end behaves as a circular buffer, with newer updates appended to the tail, and flushes beginning at the head. Based on the address of the buffered update, the modified cache lines belonging to updates from all cores are co-located in larger *PB Back Ends* situated near each of the PM controllers. The PB Back Ends are statically partitioned among threads, and allow optimizations such as epoch coalescing, which we leave for future work.

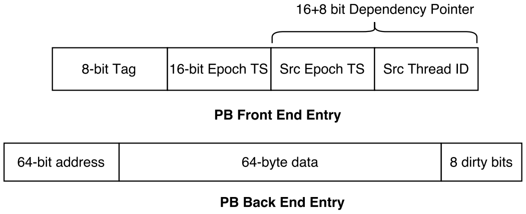

Figure 9. Hardware requirements of PB entries.

HOPS maintains write ordering with 16-bit epoch timestamps (TSs). Each hardware thread maintains a thread TS register near the private L1 cache, which indicates the timestamp of the current, inflight epoch. This thread TS is recorded as the epoch TS field of the PB Front End entry for each PM store from the thread. The `ofence` primitive simply increments the thread TS register to denote the end of an epoch, and thus is a low latency operation. Epoch TSs are local to a thread, and are used to govern the order in which epochs from a thread become durable. In case of inter-thread dependencies, dependency pointers containing both a thread ID and a source epoch TS are used to identify the epoch from another thread after which a PB entry is ordered. To simplify the hardware, we conservatively use the current epoch TS at the source thread instead of the exact source epoch TS.

When making updates from an epoch durable, HOPS issues write requests for all entries belonging to the same epoch simultaneously, without waiting for ACKs from the memory controllers. However, flushing an epoch can only commence after all ACKs are received for the previous epoch. Note that the MC ACK may be sent when PM is updated or earlier, if the MC request queues are automatically flushed on failures. For preserving cross-thread dependencies while writing updates to PM, HOPS maintains a global TS register at the LLC. This register is a vector of per-thread epoch TSs, storing a recently flushed epoch TS from each thread, and is updated regularly. HOPS looks up this register in case of a cross-dependency, to delay the flushing of dependent epochs till the source epoch has been completely flushed to PM. Thus, a `dfence` instruction that cleans a thread's PB can be a heavyweight operation in case of cross-dependencies, but otherwise can be handled locally.

A complete PB entry consists of the data (Back End) and its metadata (Front End). The hardware overheads of a PB entry are shown in Figure 9. Beyond the PBs, some additional hardware changes are needed. HOPS adds a single bit to each cache line state indicating whether the address is part of PM. We re-use existing x86 memory-type range registers (MTRRs) or the Page Attribute Table to indicate which addresses are persistent. A sticky-M state is used in the LLC to point to the L1 cache which most recently held exclusive permissions to a cache line, a technique used previously in LogTM [42]. Thus, after an L1 replacement of

Events	Action
`ofence`	Increment Thread TS to end current epoch.
`dfence`	Increment Thread TS to end current epoch, and stall thread till local PB is flushed clean.
L1 read hit, miss	No change.
L1 write hit, miss	Get exclusive permissions (miss), update cache line and mark clean. Create PB entry with epoch TS = thread TS and dependency pointer (if any).
Forwarded GET	Respond with data and (if line cached exclusively) dependency pointer (thread ID, TS) to requestor.
LLC hit	No change.
LLC miss	Send request to MC, which stalls request if address present in any PB.

Table 2. Handling of major events in HOPS.

a buffered cache line, the LLC can still forward requests to the source cache and PB to populate the dependency pointer on a dependency. We associate counting Bloom filters with the PB Back End to maintain a conservative list of buffered addresses. On a last-level cache (LLC) miss, if the address is present in this list, the miss is stalled until the address is written back to PM. Such stalls are expected to be rare as the modified data is expected to survive longer in the cache hierarchy than in the PBs. Hence, we choose to make the PB simple write-only structures, and stall on affected LLC misses. Finally, to virtualize PBs, the OS must flush PBs on context switches.

The overall PB operation is summarized in Table 2 and can be illustrated using a simple example. Consider the following code sequence:

```
mov A, 10; ofence;
mov A, 20; dfence;
```

Suppose the thread TS is 1 initially. 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 {`ts:1, Address:A, value:10`}. When `ofence` executes, it marks the start of a new epoch by incrementing the thread's TS to 2 (a purely local operation). The second store to A updates the cached value and creates another entry in the PB with {`ts:2, Address:A, value:20`}. Finally, the `dfence` increments the thread's TS to 3 and waits for the PB to drain. The PB writes the value 10 to 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 `dfence` completes.

6.4 Evaluation

Methodology. We use the gem5 micro-architectural simulator [12] to evaluate the benefit of HOPS. We use Linux v3.10 in full-system simulation mode. The simulated system is a four-core (one hardware thread per core) 8-way out-of-order x86 processor with a two-level cache hierarchy and two memory controllers. Table 3 shows the relevant configuration parameters of the simulated system. We evaluate HOPS with 32 entry PBs per thread, and flushing is launched at 16 buffered entries. We use a subset of applications from WHISPER and run them to completion.

CPU Cores	4 cores, 8-way OOO, 2Ghz
CPU L1 Caches	private, 64 KB, Split I/D
CPU L2 Caches	private, 2 MB
Cache Policy	writeback, exclusive
Coherence	MOESI hammer protocol
DRAM	4GB, 40 cycles read/write latency
PM	4GB, 160 cycles read/write latency

Table 3. Simulation configuration.

We compare HOPS to the current x86-64 approach of using `clwb` and `sfence` instructions to persist data, and to an ideal implementation. Our ideal implementation obviates `clwb` and `sfence`, thus ignoring all order between PM writes and is not crash-consistent. For the x86-64 and the HOPS implementations, we evaluate the performance under two conditions. First, with conventional memory controllers, data becomes durable only when it reaches NVM. Thus, the long-latency NVM write is on the critical path for a durability operation such as a `dfence`. Second, a persistent write queue (PWQ) at the PM controller guarantees that the data reaching the MC will become durable before a subsequent crash. Thus, this results in faster durability operations. This does not affect the non crash-consistent ideal implementation that does not guarantee durability.

Figure 10 shows the runtimes normalized to the x86-64 implementation with durability guaranteed at NVM. For the x86-64 implementation, the PWQ reduces runtimes by 15.5% on average. HOPS without PWQ outperforms the x86-64 implementation without PWQ by 24.3% and more importantly, outperforms the x86-64 implementation with PWQ by 10%. HOPS only guarantees durability at the rare `dfence` instructions. Hence, the PWQ only improves runtime by 1.4% for HOPS. This improvement comes from moving most flushes from the foreground to the background of execution. As such, the individual speedups observed are roughly proportional to the frequency of PM accesses and flushes in our workloads. The ideal implementation outperforms the baseline (x86-64 NVM) by 40.7% and HOPS (NVM) by 19.7%.

7. Related work

PM workloads. Prior PM proposals have used micro-benchmarks [17, 18, 26, 31, 32, 36, 43], conventional non-PM workloads [26, 28, 30, 36], or at best, a few persistent workloads [18, 21, 31, 39] for evaluating their implementations. As seen in our analysis, simple micro-benchmarks or non-PM applications do not capture the unique behavior of real-world PM applications. To counter this, WHISPER comprises applications that consider various interfaces to PM, update PM consistently and are recoverable.

Software/Hardware support for PM. There have been many hardware proposals facilitating fast PM accesses.

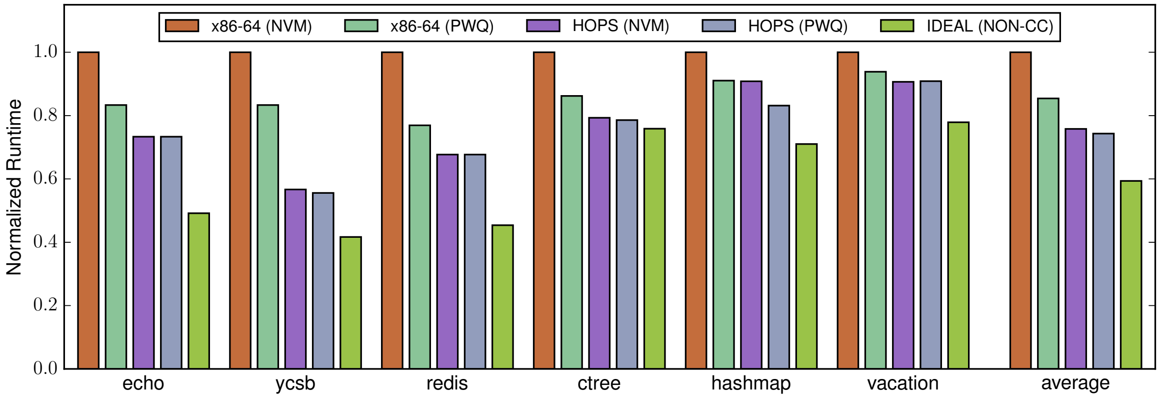

Figure 10. Performance of HOPS relative to x86-64 instructions for writeback and fence, and an ideal but crash-inconsistent implementation. Persistent Write Queue (PWQ) implementation moves the point of durability to the MC, while the NVM implementation considers the point of durability as the NVM device.

BPFS [18] proposed augmenting caches with epoch ordering hardware to allow software control of the order of writebacks. Efficient Persist Barriers [26] builds on this idea to provide lightweight epoch ordering and efficiently support inter-thread dependencies. Both proposals lack the durability needed for ACID transactions. Kiln [43] supports hardware transactions but without isolation guarantees. These proposals add state to volatile caches, which can adversely affect volatile accesses.

Most closely related is Delegated Persist Ordering (DPO), a concurrent proposal that shares with HOPS the development of persist buffers, albeit with different support mechanisms [29]. Like HOPS, DPO optimizes for fast epoch boundaries by ordering epochs without making their updates durable, handles inter-thread and intra-thread conflicts without explicit flushes and provides an express lane for persists. However, DPO does not make clear how applications ensure data is durable, e.g., for implementing ACID transactions. Additionally, DPO enforces Buffered Strict Persistency (BSP), which allows concurrent flushing of updates from the same epoch in systems with a single MC and a relaxed consistency model (ARMv7). BSP may not scale well with multiple MCs and a stronger consistency model (x86-TSO), resulting in serialized flushing of updates within an epoch. DPO's precise cross-dependency tracking mechanism requires that all incoming snoop requests, including common volatile accesses, snoop fully-associative PBs. HOPS's epoch-granular dependency tracking eliminates this overhead at the cost of false positives. DPO also requires a global broadcast on every flush of a buffered update from the PBs.

Techniques like *deferred commit* and *execute in log* have been proposed to optimize persistent transactions [28, 31]. Although these techniques consider an idealistic view of persistent transactions that differs from our observations of real-world workloads, the proposed techniques can be used even for transactions implemented with `ofence` and `dfence`.

ThyNVM [36] proposes hardware checkpointing for crash-consistency. Although transparent checkpointing removes the burden of modifying code to support persistent memory, it precludes the use of heterogeneous memory systems that include both volatile and persistent memory.

Ordering and durability. An analogous problem of conflated ordering and durability in file systems was solved by Optimistic Crash Consistency [15]. OCC introduces two new primitives—osync and dsync—to improve file system performance while satisfying application-level consistency requirements. We follow a similar approach in this work.

Loose-Ordering Consistency (LOC) [32] also proposes to relax the ordering constraints of transactions. LOC introduces *eager commit* and *speculative persistence* to reduce intra-transaction and inter-transaction dependencies. These techniques are complementary to HOPS, which instead handles dependencies efficiently.

8. Conclusion

Persistent memory is a promising interface for exposing the low-latency persistence of forthcoming NVMs to programmers. We assembled a benchmark suite, WHISPER, that comprises realistic PM applications to analyze their access patterns and identify general trends. We propose HOPS that achieves 24.3% gain in application performance by tracking and enforcing ordering and durability constraints of PM applications in hardware.

Acknowledgments

We thank HP Labs, Wisconsin Multifacet group, Prof. Thomas F. Wenisch, Dr. Vasilis Karakostas and Aasheesh Kolli for their feedback. This work was supported by the National Science Foundation under grants CNS-0915363, CNS -1218485, CNS-0834473, CNS-1302260, CCF-1438992, CCF-1533885, CCF-1617824 and John P. Morgridge Chair. Hill and Swift have significant financial interests in AMD and Microsoft respectively.

References

[1] Crit-bit tree. cr.yp.to/critbit.html.

[2] Exim Internet Mailer. exim.org.

[3] In-kernel memory-mapped I/O tracing. kernel.org/doc/Documentation/trace/mmiotrace.txt.

[4] MySQL : The world's most popular open source database. mysql.com.

[5] pmem.io: Persistent memory programming blog. pmem.io.

[6] Redis, enhanced to use NVML's libpmemlog. github.com/pmem/redis.

[7] Secrets of the Ftrace function tracer. lwn.net/Articles/370423/.

[8] SysBench: a system performance benchmark. imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[9] Arulraj, Joy and Pavlo, Andrew and Dulloor, Subramanya R. Let's Talk About Storage and Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 707–722, 2015.

[10] Bailey, Katelin A. and Hornyack, Peter and Ceze, Luis and Gribble, Steven D. and Levy, Henry M. Exploring Storage Class Memory with Key Value Stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[11] Bhandari, Kumud and Chakrabarti, Dhruva R. and Boehm, Hans-J. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 677–694, 2016.

[12] Binkert, Nathan and Beckmann, Bradford and Black, Gabriel and Reinhardt, Steven K. and Saidi, Ali and Basu, Arkaprava and Hestness, Joel and Hower, Derek R. and Krishna, Tushar and Sardashti, Somayeh and Sen, Rathijit and Sewell, Korey and Shoaib, Muhammad and Vaish, Nilay and Hill, Mark D. and Wood, David A. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.

[13] Cao Minh, Chi. *Designing an Effective Hybrid Transactional Memory System*. PhD thesis, Stanford University, Stanford, CA, USA, 2008.

[14] Carlson, Josiah L. *Redis in Action*. Manning Publications Co., Greenwich, CT, 2013.

[15] Chidambaram, Vijay and Pillai, Thanumalayan Sankaranarayana and Arpaci-Dusseau, Andrea C. and Arpaci-Dusseau, Remzi H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 228–243, 2013.

[16] Christopher Frost. *Improving File System Consistency and Durability with Patches and BPFS*. PhD thesis, University of California, Los Angeles, 2010.

[17] Coburn, Joel and Caulfield, Adrian M. and Akel, Ameen and Grupp, Laura M. and Gupta, Rajesh K. and Jhala, Ranjit and Swanson, Steven. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118, 2011.

[18] Condit, Jeremy and Nightingale, Edmund B. and Frost, Christopher and Ipek, Engin and Lee, Benjamin and Burger, Doug and Coetzee, Derrick. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pages 133–146, 2009.

[19] Intel Corporation. Intel architecture instruction set extensions programming reference, August 2015.

[20] D. Williams. Replace pcommit with ADR or directed flushing. lwn.net/Articles/694134/.

[21] Dulloor, Subramanya R. and Kumar, Sanjay and Keshavamurthy, Anil and Lantz, Philip and Reddy, Dheeraj and Sankaran, Rajesh and Jackson, Jeff. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems*, pages 15:1–15:15, 2014.

[22] Dulloor, Subramanya R. and Roy, Amitabha and Zhao, Zheguang and Sundaram, Narayanan and Satish, Nadathur and Sankaran, Rajesh and Jackson, Jeff and Schwan, Karsten. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the 11th European Conference on Computer Systems*, pages 15:1–15:16, 2016.

[23] Hewlett Packard Enterprise. Persistent memory. hpe.com/us/en/servers/persistent-memory.html.

[24] Fitzpatrick, Brad. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5–, August 2004.

[25] Intel Newsroom. Intel and Micron produce breakthrough memory technology. newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology, July 2015.

[26] Joshi, Arpit and Nagarajan, Vijay and Cintra, Marcelo and Viglas, Stratis. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 660–671, 2015.

[27] Kallman, Robert and Kimura, Hideaki and Natkins, Jonathan and Pavlo, Andrew and Rasin, Alexander and Zdonik, Stanley and Jones, Evan P. C. and Madden, Samuel and Stonebraker, Michael and Zhang, Yang and Hugg, John and Abadi, Daniel J. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, August 2008.

[28] Kolli, Aasheesh and Pelley, Steven and Saidi, Ali and Chen, Peter M. and Wenisch, Thomas F. High-Performance Transactions for Persistent Memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–411, 2016.

[29] Kolli, Aasheesh and Rosen, Jeff and Diestelhorst, Stephan and Saidi, Ali and Pelley, Steven and Liu, Sihang and Chen, Peter M. and Wenisch, Thomas F. Delegated Persist Ordering. In *Proceedings of the 49th International Symposium on Microarchitecture*, pages 1–13, 2016.

[30] Liu, Ren-Shuo and Shen, De-Yu and Yang, Chia-Lin and Yu, Shun-Chih and Wang, Cheng-Yuan Michael. NVM Duet:

Unified Working Memory and Persistent Store Architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 455–470, 2014.

[31] Lu, Youyou and Shu, Jiwu and Sun, Long. Blurred Persistence: Efficient Transactions in Persistent Memory. *ACM Transactions on Storage*, 12(1):3:1–3:29, January 2016.

[32] Lu, Youyou and Shu, Jiwu and Sun, Long and Mutlu, Onur. Loose-ordering consistency for persistent memory. In *International Conference on Computer Design*, pages 216–223. IEEE, 2014.

[33] Minh, Chi Cao and Chung, JaeWoong and Kozyrakis, Christos and Olukotun, Kunle. STAMP: Stanford transactional applications for multi-processing. In *International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.

[34] Narayanan, Dushyanth and Hodson, Orion. Whole-system Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[35] Pelley, Steven and Chen, Peter M. and Wenisch, Thomas F. Memory Persistency. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture*, pages 265–276, 2014.

[36] Ren, Jinglei and Zhao, Jishen and Khan, Samira and Choi, Jongmoo and Wu, Yongwei and Mutlu, Onur. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 672–685, 2015.

[37] SNIA. NVM Programming Technical Work Group. `snia.org/forums/sssi/nvmp`.

[38] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[39] Volos, Haris and Tack, Andres Jaan and Swift, Michael M. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–104, 2011.

[40] Wu, Xiaojian and Reddy, A. L. Narasimha. SCMFS: A File System for Storage Class Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 39:1–39:11, 2011.

[41] Xu, Jian and Swanson, Steven. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, pages 323–338, 2016.

[42] Yen, Luke and Bobba, Jayaram and Marty, Michael R. and Moore, Kevin E. and Volos, Haris and Hill, Mark D. and Swift, Michael M. and Wood, David A. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, pages 261–272, 2007.

[43] Zhao, Jishen and Li, Sheng and Yoon, Doe Hyun and Xie, Yuan and Jouppi, Norman P. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432, 2013.