WHISPER: Wisconsin-HPL Suite for Persistence

Persistent memory (PM) has received significant attention in software research [2, 5, 6, 14], hardware research, [8, 12, 15] and industry [1, 7, 10, 13]. We define PM as non-volatile (NVM) accessed with byte addressability (not just blocks) at low latency (not I/O bus) via user-mode CPU memory instructions (not system calls). Preliminary PM work, however, has used varied benchmarks and micro-benchmarks, making it challenging to compare PM proposals and be that confident proposed systems optimize for actual future PM use.

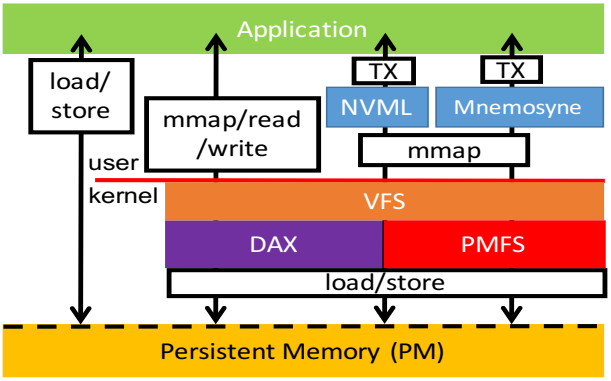

Figure 1: Current interfaces to Persistent Memory. NVML and Mnemosyne provide transactional and the kernel exposes filesystem APIs. DAX (Direct-Access) is a new Linux/Windows feature that allows filesystems (e.g., ext4, XFS) to access PM using load/store and bypass the pagecache and block layers.

This paper seeks to put PM systems research on a firmer footing by developing, analyzing and releasing a PM benchmark suite called WHISPER [9]. It is a suite of ten PM applications we gathered to cover current interfaces to persistent memory (Fig.1). We modified the applications to be recoverable and instrumented them for our own and future analysis. PM programs, such as those in WHISPER, behave differently than programs written for volatile memory only. Importantly, they carefully order and persist their writes to memory for crash-recovery after a power failure. Ordering is conveyed via *epochs*, which is are sets of unordered writes that must eventually be made durable after all previous epochs and before any following epochs [5, 11].

Our quantitative analysis of WHISPER yields several insights relevant to the design of future PM systems: (a) only 5% of application writes are to PM and the rest are to volatile memory. (b) there are 5-50 epochs per durable transaction, (c) 80% of epoch write to only one 64B cache line, (d) 80% of epochs depend on previous epochs on the same thread while only 0.1% of epochs depended on epochs from other threads.

1. WHISPER Suite

WHISPER applications fall in three categories, suitable to study access patterns across different interfaces.

Application	Brief description
N-store	DBMS for PM inspired by H-store
Echo	Scalable client-server key-value store
Memcached	Scalable object cache for web applications
Vacation	Travel booking system from STAMP
Redis	Fast object store for web applications
C-tree	Inserts/deletes to a persistent crit-bit tree
Hashmap	Inserts/deletes to a persistent hashmap
Exim	Popular mail server & mail-transfer agent
NFS	Linux client-server for remote file access
MySQL	Popular relational DBMS for OLTP

Table 1: WHISPER applications cover three interfaces to PM. Native Applications use regular loads/stores to directly access PM; Library-based applications rely on a transaction library; Filesystem applications use a PM-aware filesystem.

Native Applications: N-store [2] and *Echo* [3] access PM using regular loads/stores and implement their own crash consistency mechanisms. N-store implements DB tables atomically updated with an undo log. Echo is a multi-versioning client/server key-value store (KVS). We modified Echo and N-store to place tables, indexes and logs in PM while thread stacks and the volatile heap are in DRAM.

Library-based Applications: Macrobenchmarks using either Mnemosyne or NVML for accessing PM comprising *Memcached*, *Vacation* from the STAMP suite [4] and *Redis*. Memcached and Redis are in-memory object caches used by web applications. Vacation emulates an online transaction processing (OLTP) workload that resembles a travel reservation system. We modified Memcached and Vacation to access PM via Mnemosyne, and use Redis modified to access PM via NVML. Microbenchmarks are *C-tree* and *Hashmap* distributed with NVML for simulator-suitable studies.

Filesystem Applications: NFS, *Exim* and *MySQL* are unmodified open-source programs that access PM using PMFS [6]. NFS is a network file service, Exim is a mail server that stores user emails in per-user files, and MySQL is a popular RDBMS often used for OLTP.

2. Experimental Methodology

We instrumentWHISPER applications to record PM operations performed during application execution such as PM writes,

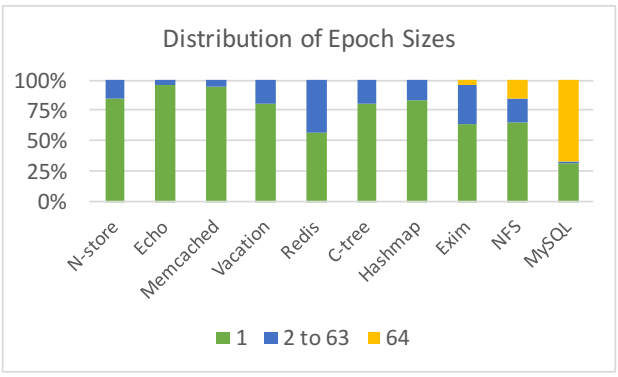

Figure 2: Size of an epoch is the number of unique cache lines written back to PM as part of the epoch.

epochs, and transactions.

Identification. We use PIN for user-space applications and *mmiotrace* for the kernel to identify all statements that update PM. We found 100 statements on average in each application and 275 in the kernel. We identify epochs as PM writes followed by a fence, and transactions boundaries from the program or library code.

Instrumentation. We use C Macros to annotate all PM operations in WHISPER and generate a trace file. This enables a variety of offline analyses in addition to the ones in our study, on various processor architectures.

3. Observations

We make several cross-cutting observations of WHISPER application's PM behavior.

Fraction of PM writes: We observed that only about 5% of all writes by an application were to PM and the rest to volatile memory, suggesting that optimizations for PM should not degrade the performance of volatile memory references.

Epochs per transaction. Most durable transactions had between 5 and 50 epochs, although in some cases, like N-store, there were over one thousand. Importantly, transactions do not require durability until the transaction commits, which is generally the last epoch. As epochs are more common than transactions, PM platforms should separate ordering of epochs from durability requirements for transactions.

Epoch size. Over 80% of epochs in native and library-based applications were *singletons*, i.e., they were one cache line in size (Fig.2). 20% of epochs had sizes varying between 2 and 63 PM cache lines. Filesystem applications exhibited epochs of size 64 cache lines due to writes to 4K data-pages. The dominant cause of small epochs was metadata writes from persistent memory allocators and log subsystems in PM runtimes. These are often invoked within transactions. Updates to the allocator state incur additional epochs as this state must be consistent at all times to avoid persistent memory leaks. PM platforms can optimize for singleton epochs for better performance or can restructure code and data to coalesce small epochs.

Cross- and Self-dependencies. We measure write-after-write (WAW) dependencies to PM addresses across epochs that arise due to shared data between application threads. For clarity, we define:

- E_i^m : Set of cachelines updated by thread i in m-th epoch.
- *Cross-dependency*: $E_i^m \otimes_c E_j^n$ denotes m-th and n-th epoch on two threads i and j respectively that update cacheline c, where E_j^n follows E_i^m during execution.
- *Self-dependency*: $E_k^m \ominus_c E_k^{m'}$ denotes epoch m and m' on thread k that update cacheline c, where E_k^m precedes $E_k^{m'}$ during execution.

Across all applications, 80% exhibited self-dependencies while only 0.1% of epochs exhibited cross-dependencies. Without multiversioning caches, repeated updates to a single PM cache line require that the writer wait for earlier updates to it become durable before proceeding, which is slow. PM platforms 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 PM in the background. Cross-dependencies are rare, and hence may not need optimizing, but must still be handled correctly.

Additionally, we note that applications use non-temporal instructions (NTIs) that bypass the cache for persistent data with low temporal locality. Between 65-99% of writes in PMFS (file data) and 14-45% in Mnemosyne (log records)use NTIs. Most proposals for implementing epochs [5, 8] do not discuss how to support NTIs.

References

[1] Persistent memory programming. pmem.io.

[2] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage and recovery methods for nvm database systems. SIGMOD '15.

[3] K.A. Bailey, P. Hornyack, L. Ceze, S.D. Gribble, and H.M. Levy. Exploring storage class memory with key value stores. INFLOW '13.

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

[5] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. SOSP '09.

[6] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. EuroSys '14.

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

[8] A Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. MICRO'15.

[9] S. Nalli, S. Haria, M.M. Swift, M.D. Hill, H. Volos, and K. Keeton. How applications use persistent memory. To appear in ASPLOS '17. Copy available on request.

[10] Intel Newsroom. newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology.

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

[12] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. MICRO '15.

[13] SNIA. snia.org/forums/sssi/nvmp.

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

[15] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N.P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. MICRO'13.