Understanding the Idiosyncrasies of Real Persistent Memory

Shashank Gugnani
The Ohio State University
gugnani.2@osu.edu

Arjun Kashyap
The Ohio State University
kashyap.49@osu.edu

Xiaoyi Lu
The Ohio State University
lu.932@osu.edu

ABSTRACT

High capacity persistent memory (PMEM) is finally commercially available in the form of Intel's Optane DC Persistent Memory Module (DCPMM). Researchers have raced to evaluate and understand the performance of DCPMM itself as well as systems and applications designed to leverage PMEM resulting from over a decade of research. Early evaluations of DCPMM show that its behavior is more nuanced and idiosyncratic than previously thought. Several assumptions made about its performance that guided the design of PMEM-enabled systems have been shown to be incorrect. Unfortunately, several peculiar performance characteristics of DCPMM are related to the memory technology (3D-XPoint) used and its internal architecture. It is expected that other technologies (such as STT-RAM, memristor, ReRAM, NVDIMM), with highly variable characteristics, will be commercially shipped as PMEM in the near future. Current evaluation studies fail to understand and categorize the idiosyncratic behavior of PMEM; i.e., how do the peculiarities of DCPMM related to other classes of PMEM. Clearly, there is a need for a study which can guide the design of systems and is agnostic to PMEM technology and internal architecture.

In this paper, we first list and categorize the idiosyncratic behavior of PMEM by performing targeted experiments with our proposed PMIdioBench benchmark suite on a real DCPMM platform. Next, we conduct detailed studies to guide the design of storage systems, considering generic PMEM characteristics. The first study guides data placement on NUMA systems with PMEM while the second study guides the design of lock-free data structures, for both eADR- and ADR-enabled PMEM systems. Our results are often counter-intuitive and highlight the challenges of system design with PMEM.

PVLDB Reference Format:
Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. PVLDB, 14(4): 626-639, 2021.
doi:10.14778/3436905.3436921

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/padsys/PMIdioBench.

1 INTRODUCTION

Database and storage system design with persistent memory (PMEM) has been an active research topic for several years. However, high capacity PMEM has been commercially available in the form of Intel Optane DC Persistent Memory Module (DCPMM) only recently. Researchers have not been waiting idly for its arrival. Over the last decade, a significant body of work has attempted to design transactional abstractions [3, 7, 17, 40, 55], persistent data structures [4, 10, 16, 22, 41, 43, 44, 49, 54, 56, 65], and file, key-value, and database systems [2, 6, 8, 11–14, 19, 27, 28, 36, 42, 47, 59–61] for PMEM. Most of these works use emulation to replicate PMEM performance. Only a small subset [6, 11, 22, 36, 41, 43, 44] evaluate with real PMEM (DCPMM). Now that DCPMM is commercially available, researchers have raced to evaluate its performance characteristics. Several recent evaluation studies [35, 39, 62, 63] have shown that PMEM behavior is more idiosyncratic and nuanced than previously thought. In fact, none of the emulation schemes used in prior work have been able to capture the nuanced behavior of real PMEM. Several of the assumed performance characteristics have been shown to be incorrect.

Several in-depth studies of DCPMM [18, 25, 26, 35, 39, 48, 53, 58, 62, 63] are now available to guide the design of DCPMM-enabled systems. However, it is expected that other memory technologies, such as STT-RAM [52], memristor [51], ReRAM [1], and NVDIMM [46], will be commercially shipped as PMEM in the near future. Moreover, Intel is planning to release the next generation of DCPMM with improvements to the architecture and memory technology [34]. Each memory technology has unique performance tendencies, different from those of 3D-XPoint [21] which DCPMM is based on. This paper is motivated by the following fundamental question: *Are DCPMM characteristics and guidelines applicable to other PMEM technologies?*

In this paper, we find that several DCPMM idiosyncrasies are only related to its internal architecture and memory technology. Therefore, the guidelines emerging from recent DCPMM studies are not broadly applicable to other classes of PMEM. Following such guidelines may result in a highly specialized system which works well with DCPMM but not with other PMEM. To prevent such over-specialization, we believe that system designers should have a proper understanding of the root causes of these idiosyncrasies and how they are applicable to different classes of future PMEM. Current evaluation studies fail to understand and categorize the idiosyncratic behavior of PMEM; i.e., how do the peculiarities of DCPMM relate to other classes of PMEM. Clearly, there is a need for a study which can guide the design of PMEM-based storage systems, and is agnostic to the type of memory technology used. This paper intends to provide such a study.

Specifically, we identify the root causes of peculiar PMEM characteristics as well as the degree of their impact. The key idea to isolate their root cause is to conduct targeted experiments on DCPMM and examine performance differences with DRAM or analyze internal hardware counters. Further, we conduct two in-depth case studies to guide the design of storage systems, with a focus on the database area, considering generic PMEM characteristics. The

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436921

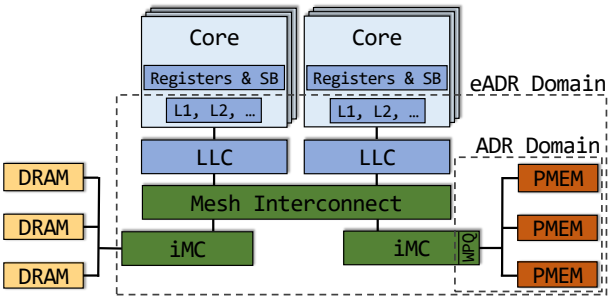


Figure 1: Architecture of PMEM-enabled Systems: *SB* is store buffer, *ADR* is asynchronous DRAM refresh, *eADR* is enhanced ADR, *WPQ* is write pending queue, and *iMC* is integrated memory controller.

first study guides data placement on non-uniform memory access (NUMA) systems with PMEM. Using MongoDB [45] as an example, we identify ways to maximize PMEM capacity utilization with minimal performance impact. The second study guides the design of lock-free data structures, for both asynchronous DRAM refresh (ADR) and enhanced ADR (eADR) PMEM systems. We present lock-free designs for a ring buffer and a linkedlist, two commonly used data structures in database systems. Our analysis shows the common pitfalls of prior work on lock-free persistent data structures. A key insight we make is that some PMEM idiosyncrasies arise because of the way PMEM is organized in the system and not the PMEM technology itself. Overall, we find that our results are often counter-intuitive and highlight the challenges of system design with PMEM.

To summarize, this paper makes the following contributions:

- PMIdioBench, a micro-benchmark suite for measuring the quantitative impact of PMEM idiosyncrasies (§3)
- A methodical categorization of PMEM idiosyncrasies (§4)
- A case study with MongoDB's PMEM storage engine to guide data placement on NUMA-enabled systems (§5)
- A case study with ring buffer and linkedlist to guide the design of lock-free persistent data structures (§6)
- A set of empirically verified technology agnostic recommendations to assist storage system developers (§7)

Throughout the paper, we have highlighted observations that we feel will be relevant to a broader audience. The most relevant PMEM idiosyncrasies and system design recommendations have been consolidated in tabular form (Tables 1 and 4).

2 PMEM BACKGROUND

PMEM System Architecture. We assume a generic PMEM system architecture which conforms to current systems with DCPMM and future systems with other PMEM. Figure 1 shows the generic architecture. We expect the system to consist of one or more identical multi-core NUMA-enabled CPUs. Each CPU has local registers, store buffers, and caches. The last level cache (LLC) is shared across all cores in a CPU. Each CPU has its own memory (DRAM and PMEM) connected to other CPUs through a mesh interconnect. The PMEM system can be designed to support one of two persistence modes – ADR or eADR. In ADR persistence mode, the PMEM DIMMs and the write pending queues (WPQ) in the integrated

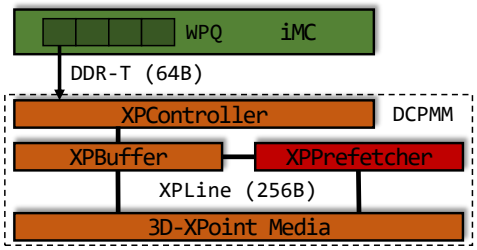


Figure 2: DCPMM Internal Architecture

memory controller (iMC) are part of the persistence domain. On power failure, all stores that have reached the ADR domain will be flushed to the PMEM DIMM. However, the CPU caches are not part of the persistence domain. So, any data left in the CPU cache will be lost in the event of power failure. In contrast, in eADR mode, the CPU caches are also part of the persistence domain (they will be flushed to PMEM in case of power failure). So, data in cache can be considered to be persistent, but data in CPU registers and store buffers will still be lost.

Optane DCPMM. Optane DCPMM is Intel's PMEM solution which conforms to our definition of PMEM and system architecture. Figure 2 shows its internal architecture. Optane DIMMs operate in ADR mode but systems designed for eADR can be tested on them for accurate performance measurements, even though the system may not be crash-consistent. The CPU memory controller uses the DDR-T protocol to communicate with DCPMM. DDR-T operates at cache-line (usually 64B) granularity and has the same interface as DDR4 but uses a different communication protocol to support asynchronous command and data timing. Access to the media (3D-XPoint) is at a coarser granularity of a 256B XPLine, which results in a read-modify-write operation for stores, causing write amplification. The XPLine also represents the error-correcting code (ECC) block unit of DCPMM; access to a single XPLine is protected using hardware ECC. Like SSDs, DCPMM uses logical addressing for wear-leveling purposes and performs address translation internally using an address indirection table (AIT). Optane DIMMs also use an internal cache (XPBuffer) with an attached prefetcher (XPPrefetcher) to buffer reads and writes. The cache is used as a write-combining buffer for adjacent stores and lies within the ADR domain, so updates that reach the XPBuffer are persistent.

3 PROPOSED MODUS OPERANDI

To categorize PMEM idiosyncrasies, we first benchmark the low-level performance of Intel DCPMM. Based on our results, we identify a list of DCPMM idiosyncrasies (see Table 1). Understanding how these peculiarities relate to other classes of PMEM is non-trivial and requires a proper understanding of the root cause of each idiosyncrasy. If the root cause (hardware component/design) is not present in other classes of PMEM, it is not applicable to those devices. Table 2 lists the classes of PMEM we consider in this paper. To identify the root cause, we propose two techniques. The first is to see whether the idiosyncrasy also exists when the same benchmark is used on DRAM. If it also exists with DRAM, we can pinpoint some hardware component in the system other than DRAM and PMEM as the root cause. In such cases, the idiosyncrasy would be applicable to other PMEM classes as well since the root

Table 1: PMEM Idiosyncrasy Categorization. *Impact is worse than other classes +Also applicable to PMEM using PCM/flash

ID	Idiosyncrasy	Root Cause(s)	Applicability	Figure
I1	Asymmetric load/p-store latency	CPU cache	C1–C4	Fig. 3(a)
I2	Asymmetric load/p-store bandwidth	iMC contention & 3D-XPoint latency	C4+	Fig. 3(b),(c)
I3	Poor bandwidth for small-sized random IO	Access granularity mismatch	C3,C4	Fig. 4
I4	Poor p-store bandwidth on remote NUMA PMEM	Mesh interconnect & XPPrefetcher	C1,C2*,C3,C4*	Fig. 5
I5	Store bandwidth lower than p-store bandwidth	XPPrefetcher	C2,C4	Fig. 6
I6	Sequential IO faster than random IO	CPU & XP Prefetchers	C1,C2*,C3,C4*	Fig. 7(a),(b)
I7	P-stores are read-modify-write transactions	XPBuffer design	C4	Fig. 7(d)

Table 2: Classes of PMEM considered in this paper. Note that the classes are not mutually exclusive. For instance, DCPMM is also in C2 and C3.

Class	Description
C1	NVDIMM/Battery-backed DRAM
C2	PMEM with internal cache and prefetcher
C3	PMEM with access granularity mismatch
C4	Intel DC Persistent Memory Module (DCPMM)

cause is not related to the PMEM technology. The exact component can be verified by considering the impact of each component in Figure 2 and micro-architectural numbers. The second technique is to look at DCPMM hardware counters to identify the exact internal architectural component responsible for the peculiarity. In such cases, the peculiarity would be applicable to other classes of PMEM which have the same component too. We use these two techniques to understand the applicability of the most relevant DCPMM performance characteristics. We only focus on characteristics which defy conventional wisdom or have the most impact on performance. To the extent possible, we keep the discussion agnostic to the CPU architecture.

Metrics. We use three metrics to understand the performance characteristics of PMEM systems – effective write ratio (EWR), effective bandwidth ratio (EBR), and read-write ratio (RWR). EWR is a metric specific to PMEM with a mismatch in access granularity, first proposed in [63]. It is the ratio of bytes written to the iMC and the bytes written to the 3D-XPoint media, essentially the inverse of write amplification. Along similar lines, we propose EBR as a PMEM technology independent metric. It is the ratio of average achieved PMEM bandwidth and the peak bandwidth. Unlike EWR, EBR has no direct relation to write amplification but only to achieved bandwidth and is also meaningful when there is no write amplification. RWR is the ratio of bytes read and bytes written to either the iMC or 3D-XPoint media in case of DCPMM. Our analysis shows that these metrics are useful in gaining an in-depth understanding of performance trends. EWR and EBR are useful in identifying sub-optimal access size, pattern, and concurrency. RWR is useful in determining workload type as read or update heavy.

Terminology. We use nt-store to indicate an optimized non-temporal store instruction that bypasses the cache hierarchy. We define persistent store (or p-store) as a store instruction which is persistent. A p-store comprises of a regular store followed by a cache-line flush and store fence or a nt-store followed by a store fence. We define persistence barrier as an instruction which guarantees durability of data (cache-line flush + store fence). In our evaluation, we use cache-flush instructions which invalidate the cache-line instead of write-back instructions which do not invalidate cache-lines because write-back instructions are not fully implemented on Intel CPUs yet [53].

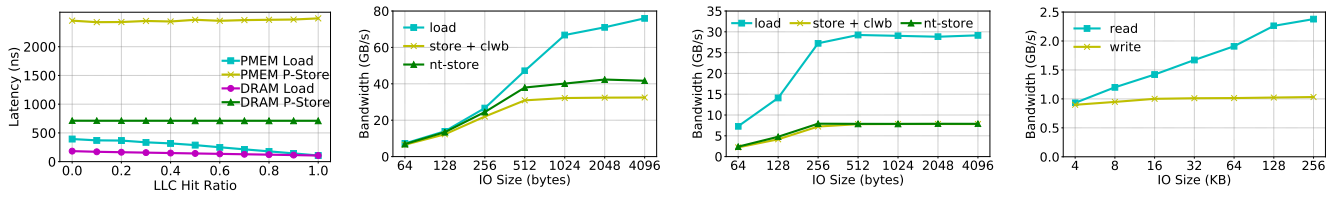
3.1 PMIdioBench

Understanding and categorizing PMEM characteristics is non-trivial. This is because existing benchmarks only provide the ability to evaluate performance characteristics but not isolate their root cause. To fully understand the nuanced behavior of PMEM, we designed a micro-benchmark suite called **PMIdioBench**. The suite consists of a set of targeted experiments to evaluate PMEM performance and identify its idiosyncrasies. There is one benchmark for every idiosyncrasy shown in Table 1. The benchmarks are designed to measure the latency or bandwidth of PMEM access under different scenarios, thread counts, and access sizes. PMIdioBench pins threads to cores, disables cache prefetching (when required), and primes the CPU cache to produce accurate and precise performance measurements. It also includes two tools to help identify the main reason behind performance anomalies and bottlenecks – one to examine relevant hardware counters and display useful metrics (such as EWR, EBR, and RWR) and the other to generate benchmark trace and distill gathered information in the form of a flame graph [20]. PMIdioBench can be used not only to verify and reproduce the results presented in this paper, but also to calculate the quantitative impact of the idiosyncrasies on future PMEM.

Limitations. PMIdioBench only contains targetted experiments to identify idiosyncrasies that have been observed with DCPMM. If the PMEM being evaluated has some unique characteristic not observed earlier, then PMIdioBench can not identify it. In addition, PMIdioBench requires PMEM hardware counters to be available at the granularity of cache-lines. Without access to these counters, it is not possible to identify the root cause of some idiosyncrasies.

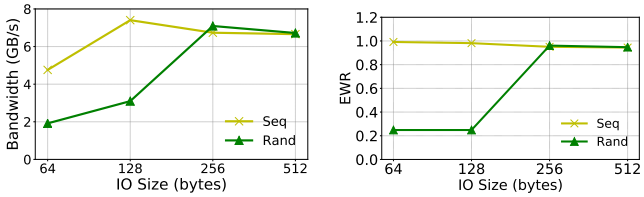
3.2 Experimental Testbed

Our experimental testbed consists of a Linux (3.10) server equipped with two Cascade Lake CPUs (8280L@2.70GHz), 384GB DRAM (2 x 6 x 32GB DDR4 DIMMs), 6TB PMEM (2 x 6 x 512GB DCPMMs) configured in App Direct mode, and a 1TB Intel P4510 Flash SSD. Each CPU has 28 cores and 38.5MB of L3 cache (LLC). All available PMEM is formatted as two `xfs` DAX filesystems, each utilizing the memory of a single CPU socket as a single interleaved namespace.



(a) PMEM and DRAM Load/P-Store Latency (b) DRAM Load/P-Store Bandwidth (c) PMEM Load/P-Store Bandwidth (d) Flash Read/Write Bandwidth

Figure 3: (a) Asymmetric load/p-store latency - I1, (b), (c), and (d) Asymmetric load/p-store bandwidth - I2



(a) PMEM Write Bandwidth (b) PMEM EWR

Figure 4: Poor bandwidth for small-sized random IO - I3

All code was compiled using gcc 9.2.0. PMDK [30] 1.8 was used across all evaluations to keep comparisons fair. Hardware counters were obtained using a combination of Intel VTune Profiler [29], Intel PCM [32], and Linux perf utility. DCPMM hardware counters were collected using the `ipmwatch` utility, a part of the VTune Profiler.

Reproducibility. The experimental analysis in this paper was done while keeping reproducibility in mind. All of the benchmarks, performance analysis tools, and systems used in this paper are open source. Further, code for PMIdioBench as well as our lock-free ring buffer and linkedlist implementations have been open-sourced. A document with detailed steps to reproduce all figures in this paper is included with the code.

4 IDIOSYNCRASY CATEGORIZATION

In this section, we list and categorize PMEM idiosyncrasies through targeted experiments. A summary of the idiosyncrasies with their root cause and applicability can be found in Table 1.

4.1 Asymmetric load/p-store latency

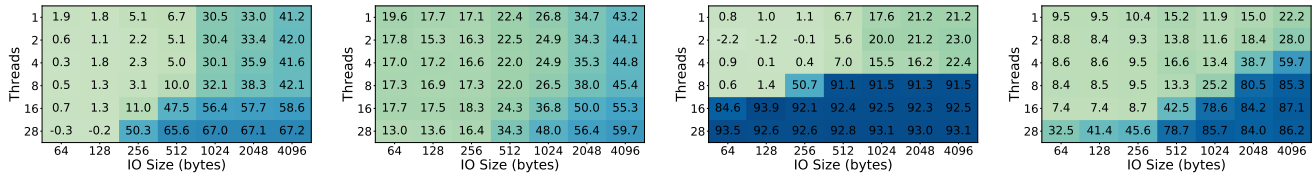
When considering PMEM latency, it is important to consider the effects of caching. Prior studies [35, 62, 63] only consider direct PMEM latency which does not reflect the latency perceived by applications. In this study, we measure cached latency, which reveals the asymmetric nature of PMEM latency. Our experiment uses a region of PMEM equal to the LLC size. We prime the LLC by accessing a fraction (f) of the cache-lines using CPU loads. Finally, we do load or p-store operations on the mapped region with 8 threads and measure operation latency. By changing f, we can effectively control the LLC hit ratio. Figure 3(a) shows the results of this experiment with both DRAM and PMEM. First, we find the p-store latency for PMEM is 3.4x worse than DRAM but load latency is only up to 2.2x worse for PMEM. The reason for this is the poor write bandwidth of PMEM. Second, we observe that when all accesses directly go to memory, the latency difference between PMEM and DRAM loads is significant. However, this difference decreases drastically as the LLC hit ratio is increased. As more loads are serviced from the cache, their latency decreases and converges to the cache latency. However, store latency is dominated by cache flush latency and is unaffected by the increased cache hit rate. At a hit ratio of 1, DRAM has a 6.7x difference between load and p-store latency, while PMEM has a 23.3x difference. Since this trend is observable for both PMEM and DRAM, it is generalizable. This result is significant because it shows that under real-world use cases, where a majority of PMEM accesses are cache hits, p-stores are much slower than loads. Therefore, **there is benefit in avoiding unnecessary p-stores and keeping relevant data in cache**.

4.2 Asymmetric load/p-store bandwidth

We measure the bandwidth of DRAM, PMEM, and Flash for different IO sizes. Results are shown in Figures 3(b), 3(c), and 3(d). PMEM load bandwidth (max 30GB/s) is only 2.5x worse than DRAM (max 76GB/s) but PMEM store bandwidth (max 8GB/s) is 5.2x worse than DRAM (max 42GB/s). In general, store bandwidth is lower than load bandwidth because of the slow write latency of non-volatile storage, an effect previously observed in flash memory [5, 57]. However, the asymmetry is more pronounced for PMEM. There is only a 1.7x and 2.3x difference in peak DRAM and Flash read/write bandwidth but the difference for PMEM is 3.7x. The reason for this additional difference is contention at the iMC (also discussed in [63]). Therefore, we expect that PMEM designed with other memory technologies with characteristics similar to DCPMM (such as PCM [38] and flash) will manifest this asymmetry, but not to the same extent as DCPMM.

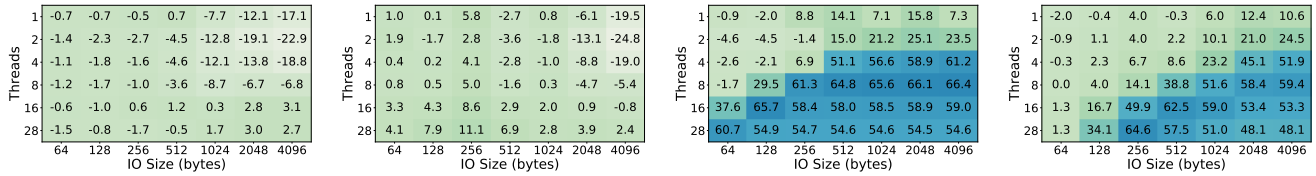
4.3 Poor bandwidth for small-sized random IO

In this experiment, we measure the bandwidth of small-sized (64–512B) IO on PMEM. Figure 4(a) shows the results for both sequential and random IO. We find that sequential bandwidth is mostly the same across all IO sizes but random bandwidth is significantly lower at 64B and 128B IO sizes. The reason for this behavior is the 256B size of the internal ECC block (XPLine) in DCPMM. Any IO smaller than the block size results in write amplification, reducing bandwidth. This effect is less apparent with sequential IO because DCPMM uses an internal write-combining buffer to merge adjacent writes to the same ECC block. To verify this behavior, we measure EWR while running this experiment (shown in Figure 4(b)). Results clearly show the write amplification issue at 64B and 128B for random IO. For access sizes of 256B or larger, there is no write amplification because there are no partial XPLine writes. The fact that small-sized IO results in poor bandwidth is well known and



(a) DRAM Seq (b) DRAM Rand (c) PMEM Seq (d) PMEM Rand

Figure 5: Poor p-store bandwidth on remote NUMA - I4. Heatmap is annotated with the percentage difference between local and remote NUMA bandwidth. A positive value indicates that remote access is worse while a negative value indicates the opposite.



(a) DRAM Seq (b) DRAM Rand (c) PMEM Seq (d) PMEM Rand

Figure 6: Lower bandwidth for store than p-store - I5. Heatmap is annotated with the percentage difference between store and p-store bandwidth. A positive value indicates that store is worse than p-store while a negative value indicates the opposite.

is applicable to any storage device. Therefore, this idiosyncrasy is generic, but the mismatch between the cache-line and XPLine sizes implies that it is more severe with PMEM that has a similar mismatch in access granularity.

4.4 Poor p-store bandwidth on remote NUMA

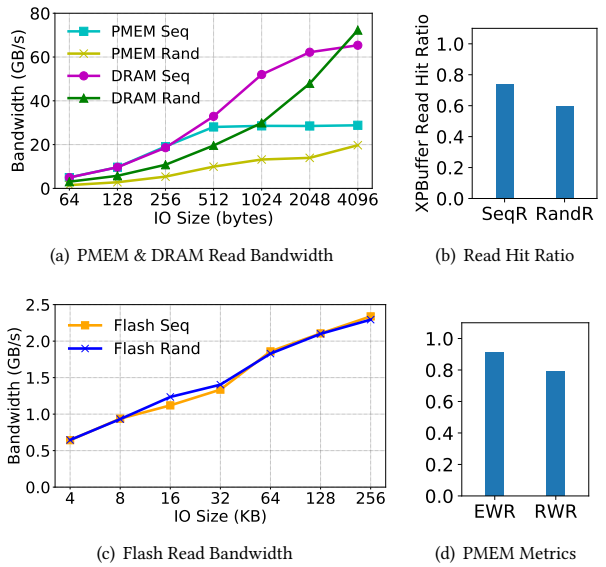
With the recent popularity and ubiquity of NUMA systems, it is important to study the performance implications of remote NUMA access. To this end, we measure the difference in local and remote NUMA bandwidth for both DRAM and PMEM. Figure 5 shows the heatmap for this evaluation annotated with the percentage difference in bandwidth between the two modes. Having to go over the mesh interconnect for each transaction adds overhead which reduces overall bandwidth. A common observable theme across all cases is that high concurrency and large IO sizes show more performance degradation for remote access (visible as the staircase like pattern in the heatmaps). This is because the inter-socket bandwidth and Intel Ultra Path Interconnect (UPI) lanes are both limited. Interestingly, PMEM suffers from higher degradation for remote access, particularly for sequential IO. The reason for this is that UPI lane sharing across threads changes the access pattern from sequential to random. The degradation is higher for PMEM because it is more sensitive to the IO access pattern than DRAM because the XPPrefetcher in DCPMM benefits sequential IO more than random IO (see Figures 7(a) and (b)). We observe reduction in bandwidth with both DRAM and PMEM because of the overheads of remote NUMA access, therefore this idiosyncrasy is applicable to all classes of PMEM. However, it will be more severe with PMEM that use an internal cache with a prefetcher.

4.5 Lower bandwidth for store than p-store

We measure the bandwidth of persistent stores (followed by a cache-line flush) and regular stores (not followed by a flush) for both DRAM and PMEM. Results, presented in Figure 6, show the annotated heatmaps with the percentage difference in bandwidth. We observe that there is significant difference in bandwidth for PMEM but not for DRAM. We believe that this trend is a result of the cache hierarchy changing the cache write-back pattern to a non-deterministic random pattern for regular stores. The cache eviction algorithm works asynchronously and is independent of the application IO pattern. Hence, it effectively converts sequential CPU stores into random writes to PMEM. Random IO on PMEM is worse than sequential IO (see Figure 7(a)) because of the effects of the XPPrefetcher on the internal cache in DCPMM (see Figure 7(b)). However, DRAM random IO does not suffer from this issue. Therefore, we conclude that this characteristic is specific to PMEM with an internal cache using a prefetcher. In eADR mode, applications typically do not flush the cache to minimize latency because the cache is in the persistence domain. However, this experiment shows that **not flushing the cache can in fact be detrimental to bandwidth when using PMEM with a built-in prefetcher**.

4.6 Sequential IO faster than random IO

In this experiment, we compare the sequential and random read bandwidth of DRAM, PMEM, and Flash. Figures 7(a) and 7(c) show the read bandwidth comparison with varying IO size. There is difference between sequential and random bandwidth for both DRAM and PMEM but not for Flash. Flash does not show this idiosyncrasy because its low bandwidth is easily saturated, regardless of IO pattern. Overall, the difference is much higher with PMEM than DRAM. The maximum degradation is 3.5x with PMEM but only 1.7x with DRAM. To understand the additional degradation with PMEM we analyze the read hit ratio of the internal XPBuffer in DCPMM (see Figure 7(b)). We find that the ratio is 19% lower for random IO, likely because the **XPBuffer has a prefetcher which is beneficial for sequential IO**. This idiosyncrasy is also present in DRAM because of the CPU prefetcher but it is worse for DCPMM



(a) PMEM & DRAM Read Bandwidth (b) Read Hit Ratio

(c) Flash Read Bandwidth (d) PMEM Metrics

Figure 7: (a), (b), and (c) Sequential IO faster than random IO - I6, (d) P-stores are read-modify-write transactions - I7

because of the presence of two prefetchers – CPU and XP Prefetcher. These results indicate that this idiosyncrasy is applicable to any class of PMEM, but is considerably worse for classes that have an internal prefetcher.

4.7 P-stores are read-modify-write transactions

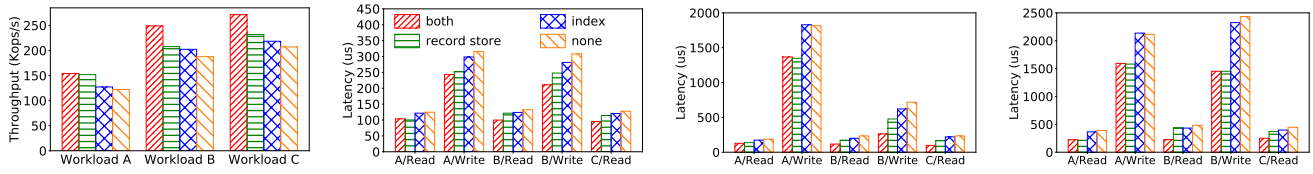
This experiment is particularly targeted to verify certain idiosyncratic behavior of the DCPMM XPBuffer. In the experiment, we do repeated non-temporal writes to the same XPLine and measure the EWR and RWR. Results are shown in Figure 7(d). To our surprise, both EWR and RWR are close to 1. This implies that DCPMM writes an XPLine to media as soon as it is completely written, even if the XPBuffer is not full. Further, even though we are writing a complete (256B) XPLine, writes are broken up into (64B) cache-line size requests because DDR-T operates at cache-line granularity. This results in a read-modify-write transaction for every internal store. These observations lead us to conclude that the XPBuffer is not being efficiently utilized. There are two ways the DCPMM internal architecture can be improved. First, retaining fully written lines in the XPBuffer will increase its hit ratio for workloads with high temporal locality. Second, if the controller finds consecutive writes to same XPLine, then it should not read data from media to avoid unnecessary read-modify-writes. We believe that future generations of DCPMM should adopt these optimizations to improve performance.

5 CASE STUDY – NUMA-AWARE DATA PLACEMENT WITH MONGODB

In this section, we present a case study to guide data placement on NUMA systems with MongoDB's persistent memory storage engine (PMSE) [31]. We currently only consider the case where data structures are placed entirely on local or remote NUMA PMEM. Cases where a single data structure may span across NUMA nodes is left as future work. Our goal is to guide the placement of persistent data structures in MongoDB to minimize performance impact and maximize capacity utilization. In general, placing all data structures on local NUMA is the most performance optimal configuration, whereas distributing them evenly among available NUMA nodes maximizes capacity utilization and performance isolation. To find the optimal configuration, we conduct thorough experiments with MongoDB considering all possible placement scenarios. Once the optimal configuration is found, we analyze the reason for its optimality by examining low-level PMEM counters. Finally, we design a set of empirically verified guidelines to follow which can be applied to other storage systems and is agnostic to PMEM technology.

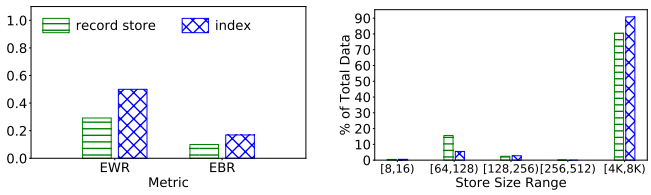
MongoDB's PMSE is an alternate storage engine for optimal usage of persistent memory. Its design primarily consists of two persistent data structures – a b+tree (index) and a record store (RS). Since both structures offer immediate persistence and consistency, journaling and snapshots are not required. PMSE relies on PMDK's libpmemobj-cpp transactional bindings to make the data structures crash consistent. We test the performance of PMSE using YCSB [9] workloads A, B, and C. For our experiments, MongoDB server daemon is pinned to NUMA node 0 and YCSB client threads are pinned to NUMA node 1. We also tuned the b+tree node size and the record store buffer size to be multiples of PMEM block size (256B). A total of four cases are considered – *both*: RS and index both placed on node 0's PMEM, *none*: both placed on node 1's PMEM, *record store*: RS and index placed on node 0's PMEM and node 1's PMEM, respectively, *index*: opposite of *record store*. Figure 8 shows the throughput and latency for different workloads and configurations at full subscription (28 threads). As expected, *both* has the best performance, while *none* has the worst. Local NUMA access is much faster than remote NUMA access, as we verified in Figure 5. We also find that *record* shows performance close to *both* for all workloads. Surprisingly, the write P99 latencies of *record store* and *both* are the same which indicates that record store accesses contribute the most to tail latency. Therefore, *record store* configuration is optimal one since it shows good performance and can also fully utilize the available PMEM capacity. To understand why this configuration is optimal, we analyze the low-level PMEM counters for both data structures separately. We use two metrics to understand the performance characteristics of each structure – effective write ratio (EWR) and effective bandwidth ratio (EBR). For both metrics, a high value is desirable and indicates optimal usage of PMEM.

Figure 9(a) shows the values of these two metrics for the two data structures. Our results show that the record store data structure has far lower values for both metrics which implies that it likely has a sub-optimal access pattern. To verify this, we analyze the PMEM store access pattern for workload A. We measure the percentage of data accessed across different granularities, shown in Figure 9(b). We find that both data structures have a majority of accesses at 4–8KB granularity. However, the record store has more accesses in the 64–128B granularity than the b+tree index because updating its memory allocator atomically requires small-sized stores. This justifies the EWR and EBR trends we observe. Therefore, we conclude that a poor EBR or EWR metric means that the data structure should be placed on the local NUMA node to minimize performance degradation. Since EWR is not applicable to



(a) Throughput (b) Average Latency (c) P95 Latency (d) P99 Latency

Figure 8: Evaluation of MongoDB PMEM Storage Engine with YCSB



(a) PMEM Metrics (b) Store Histogram

Figure 9: MongoDB YCSB Drilldown

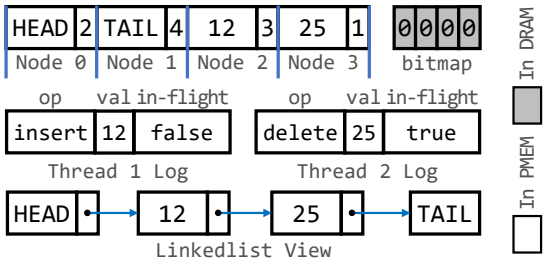


Figure 10: TLog Design Overview with 2 Threads. Thread 1 has completed its operation whereas Thread 2 is in-flight. 0 in the bitmap indicates that the memory region is allocated.

Table 3: Summary of Lock-free Designs Evaluated. Persistence mode 'both' is equivalent to both ADR and eADR.

Name	Data Structure	Persistence Mode	Design Technique
Link-free [65]	linkedlist	both	dirty-bit
SOFT [65]	linkedlist	both	dirty-bit
TLog*	linkedlist	both	per-thread logging
Log-free*	linkedlist	eADR	atomic operations
Volatile [37]	ring buffer	none	atomic operations
TX*	ring buffer	both	transactions
TX-free*	ring buffer	eADR	per-thread status buffer

*Proposed in this paper

all PMEM, we instead advocate for the use of EBR. Our observation can be summarized as the following recommendation: **Place data structures with higher EBR on remote NUMA.**

6 CASE STUDY – DESIGNING PERSISTENT LOCK-FREE DATA STRUCTURES

In this section, we first discuss challenges associated with persistent lock-free data structure (PLFS) design. Then we present the design and evaluation of two PLFSs – ring buffer and linkedlist. The intention of this case study is two fold. First, we want to highlight the difference in PLFS design between eADR and ADR persistence modes. Our findings show that designs which assume the presence of ADR may not provide the best performance in eADR mode, particularly for update heavy workloads. So, there is merit in tailoring designs for eADR mode. Second, we want to compare and contrast PLFS design techniques. Our results show that selecting an appropriate technique can significantly impact performance and should be done considering the workload IO pattern and persistence mode.

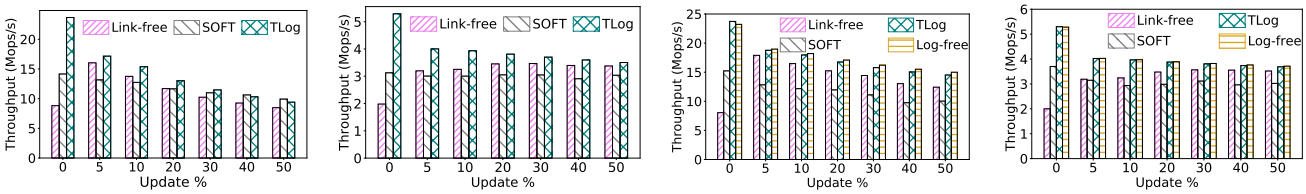
6.1 Design Techniques

There are several existing techniques in literature which can be used to implement PLFS. Providing atomicity and isolation guarantees requires the use of complex transactional systems, which often use locks for mutual exclusion. The use of locks implies that the data structure is no longer lock-free. Transactional abstractions which use redo/undo logging are one of the most common techniques but as many of these rely on locks internally they are not applicable. Some abstractions, such as PMDK's libpmemobj library [33] do not use locks but rely on per-thread buffers to maintain transaction state. Such abstractions can be used to maintain consistency and lock-freedom. Another recently developed technique is the *dirty-bit* design, as used in [56, 65]. In this approach, data is marked *dirty* when it is updated; threads which find the dirty bit set flush data to PMEM so that only committed data is read. Using per-thread scratch buffers for logging or status tracking is another approach that can be used. Since threads operate on independent memory regions, they can operate in a lock-free manner. Finally, for eADR mode, relying on atomic operations is sufficient to maintain consistency. Note that only using atomic operations may not be sufficient in this case because some information may be required to identify the data structure state on recovery from crash. Other techniques, such as per-thread logging may also be necessary. Also note that in all designs atomic operations are required to maintain mutual exclusion.

In this section, we present designs for two truly lock-free persistent data structures – a multi-producer, multi-consumer ring buffer and a concurrent linked list. We chose these data structures because they are commonly used in database systems for several different purposes. Table 3 shows a summary of the different designs we evaluate. Designs with persistence mode 'both' were designed for ADR mode but can also be run in eADR mode by replacing all persistence barriers with store fences. The main point of this study is not to implement the best possible design but rather to compare and contrast different design approaches.

6.2 Lock-free Linkedlist

We consider a sorted linkedlist with any arbitrary structure as the value. Our lock-free linkedlist designs are extensions of Harris'



(a) ADR (1024 Range) (b) ADR (4096 Range) (c) eADR (1024 Range) (d) eADR (4096 Range)

Figure 11: Lock-free Linkedlist Throughput Evaluation

algorithm [23] and support the same basic operations – `insert`, `delete`, and `contains`. The original algorithm uses the atomic `compare-and-swap` operations to maintain lock freedom. Node deletion is logical and requires subsequent garbage collection to reclaim memory. The logical deletion process involves marking nodes as *removed* by setting the least significant bit of the node's pointers. Our designs build upon the original algorithms to provide persistence. We propose one design for ADR systems, called TLog, and another for eADR systems, called Log-free. Both designs use the same memory allocator and base design.

TLog Design. TLog adds persistence barriers to ensure durability. A new node is flushed to PMEM before being added to the list. In addition, changes to a node's next pointer are also flushed. For maintaining operation atomicity, we rely on per-thread scratch buffers for micro-logging operations. Before initiating a list operation, each thread stores the operation type and its parameters in its scratch buffer and marks it as in-flight. Once the operation is completed, the operation is marked as completed. On recovery, scratch buffers of all threads are examined and any in-flight operations are redone. Recovery is idempotent because insert and delete operations are both idempotent themselves.

Log-free Design. In eADR mode, the volatile cache is in the persistence domain. Insert and delete operations have a single linearization point (the `compare-and-swap` instruction). Therefore, we do not need to use transactions or journaling to maintain atomicity. We can use Harris' algorithm as is, but with store fences at linearization points. However, we still need to prevent PMEM leaks and allow PMEM address space relocation. Our memory allocator solves these two problems.

Memory Allocator Design. The memory allocator consists of a fixed size PMEM slab indexed using a volatile lock-free bitmap. The bitmap relies on atomic `fetch-and-and/or` instructions to retain lock freedom. Our memory reclamation algorithm uses epoch-based reclamation (EBR) [15] to maintain correctness and garbage collect unused memory. To prevent PMEM leaks, we keep the bitmap in volatile DRAM and rebuild its state upon recovery. This is done by walking through the linkedlist and marking the memory region for each node as allocated in the bitmap. In this manner, memory allocation does not require transactional support. Finally, to safeguard against PMEM address space relocation, we use relative pointers and pointer swizzling [7, 50].

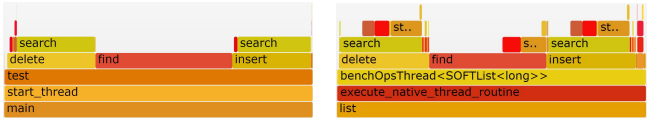
Figure 10 shows an overview of the TLog design. As can be inferred from the figure, linkedlist nodes are allocated from the PMEM slab and use node offsets instead of pointers. In the example shown, two threads operate on the linkedlist. Thread 1 has completed its insert operation and marked the in-flight flag in its log as `false`. Thread 2, on the other hand, has not completed its delete operation and its in-flight flag is still set to `true`. If there is a crash at this moment, then on recovery Thread 2's pending operation will be completed using data in the log. The Lock-free design is the same as shown in the figure, except that the per-thread logs are not required.

Evaluation. We compare our designs with two current state-of-the-art approaches, SOFT and Link-free, proposed in [65]. We measure the total throughput achieved by all implementations while varying the update ratio and value range at full subscription (28 threads) to simulate a heavy load scenario.

Figures 11(a) and (b) compare list throughput in ADR mode for 1024 and 4096 value ranges. TLog outperforms Link-free in all cases and SOFT in all but two cases. Link-free works similar to TLog but does not use per-thread logs for atomicity. Instead, find operations are required to flush the node (if not already persisted) before returning it. This increases the latency of find operations and reduces overall throughput. SOFT is an optimization of Link-free in that it does not flush node pointers but uses a valid flag per-node to indicate which nodes are part of the list. On recovery, all PMEM nodes are scanned to find valid nodes and are added to the list. SOFT reduces the number of persistence barriers required, so insert and delete operations are fast. However, it still requires find operations to flush nodes, increasing find latency. On the other hand, TLog uses micro-logging for atomicity and does not rely on find operations doing persist barriers. Therefore, TLog find latency is much lower than SOFT or Link-free but insert and delete latency is higher. This also explains why TLog performance gets closer to Link-free and SOFT as the update percentage is increased.

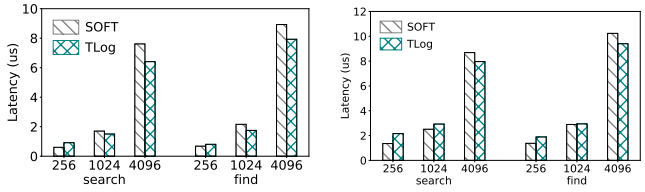
Figures 11(c) and (d) compare list throughput in eADR mode for 1024 and 4096 value ranges. TLog outperforms both Link-free and SOFT in all cases. This is because in eADR mode there is no need to flush cache-lines. So, insert and delete operations do not incur persistence overheads. SOFT and Link-free optimize insert/delete over find and are hence outperformed by TLog. We also observe that Log-free outperforms TLog in all but two cases, albeit only by a small margin. The Log-free design takes advantage of the eADR mode to avoid the micro-logging operation which TLog performs. By eschewing logging, Log-free achieves better throughput, particularly for 1024 value range. The Log-free design's improvement over TLog is only marginal because a majority of time is spent in search and find operations as opposed to persistence barriers.

To make complete sense of the ADR results, we first analyze the time-wise breakdown of each function call using a flame graph. Figure 12 shows the flame graphs for TLog and SOFT with 1024 value range and 50% updates. The stacked boxes show the function



(a) TLog (b) SOFT

Figure 12: Flame Graph of Persistent Linkedlist Implementations with 1024 Value Range and 50% Updates



(a) 5% Updates (b) 50% Updates

Figure 13: Lock-free Linkedlist Latency for Search and Find

call trace and the width of each box is proportional to total time spent in that function. For both implementations, a majority of time is spent in iterating over the list (search and find operations). Persistence barriers and other operations comprise a very small fraction of overall time. Increasing value range or decreasing update percentage will further reduce this fraction of time. According to [24], a typical application using list-based sets performs 90% reads. This indicates that optimizing linkedlist traversal is more important than minimizing persistence operations. Both SOFT and Link-free focus on optimizing persistence, resulting in a sub-optimal design. On the other hand, TLog does not change search and find operations (as compared to Harris' algorithm), and shows better performance for find intensive cases. To verify this reason, we measure the latency of search and find operation while varying value range (256, 1024, and 4096) and update percentage (5 and 50). Results are shown in Figure 13. We find that on increasing value range SOFT latency increases more for both operations as compared to TLog. These results confirm the reasons for the performance trends in Figure 11.

6.3 Lock-free Ring Buffer

We base our designs on Krizhanovsky's algorithm [37] for volatile lock-free ring buffers.

Volatile Design. The original algorithm uses per-thread head and tail pointers to maintain lock freedom. To perform a push or pop operation, each thread increments the global head/tail pointer using a `fetch-and-add` instruction and stores the old value in its local head/tail pointer. The local pointer indicates the queue slot which the thread will operate on. Before operating on the location, the thread must make sure that it is safe to push or pop at that slot. To ensure this, two global variables (last tail and last head) are maintained to indicate the earliest push and pop operations still in-flight. Threads compute theses variables for each operation by iterating over all thread-local head/tail pointers. These variables are used to ensure that we do not push at a slot still being popped or vice-versa. Once each thread completes its operation, it sets its local head/tail pointer to `INT_MAX`. This allows other threads to push/pop

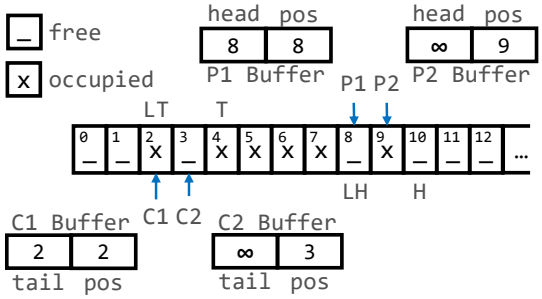


Figure 14: Ring Buffer Design Overview with 2 Producers and 2 Consumers: LT is last tail, T is tail, LH is last head, and H is head.

that slot. Essentially, this store instruction is the linearization point of an operation where it becomes visible to other threads.

Persistent Design Overview. We propose two designs for our lock-free persistent ring buffer – a transaction-based design, TX, for ADR systems and a transaction-free design, TX-free, for eADR systems. For both designs, the ring buffer, global head/tail pointers, and thread-local pointers are placed in PMEM. Last head and last tail pointers are placed in DRAM instead of PMEM because they can be computed using thread-local pointers and hence do not need to be persisted. We also add two thread-local pointers in PMEM (push and pop position) to indicate the slot position where the thread is operating on. These pointers are used to identify the slots for which push/pop operations were interrupted in case of a crash. Memory management is fairly straightforward since the ring buffer is of a fixed size. We allocate PMEM for all necessary structure and pointers statically on application startup. The simplified memory allocation avoids PMEM leaks. In addition, we use indices instead of real pointers to allow PMEM address space relocation. Figure 14 shows an overview of the ring buffer design with 2 producers and 2 consumers. As shown in the figure, each producer/consumer has a private buffer which contains the local head/tail and position pointers. The position variable always indicates the slot which was being operated on. On recovery, the head/tail pointer is examined. If it is not `INT_MAX` (∞), then the corresponding operation was left incomplete. In this manner, all incomplete operations can be detected by scanning the buffers of all producers and consumers and appropriate steps can be taken to restore the state to produce a consistent view of the ring buffer.

TX Design. In this design, we wrap the critical section of push/pop operations with transactions for atomicity. We use PMDK's libpmemobj transactional bindings for this purpose, which use a combination of redo and undo logging to achieve atomicity. The use of transactions guarantees that there are no partially completed operations in case of a crash. On recovery, we examine the push/pop position pointers to determine which buffer slots were being operated on at the time of crash. We use this information to consolidate the buffer, i.e., copy data to remove holes, which can occur as a result of a subset of threads initiating their operations at the time of crash. Using PMDK transactions does not compromise lock-freedom because transactions use thread-local buffers to store internal state and avoid synchronization.

TX-free Design. The TX-free design follows the same principle as the TX design but does not require the use of transactions. Only

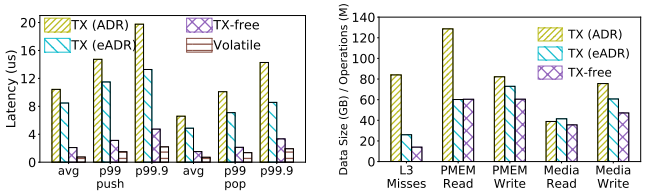
Figure 15: Effect of flushing data with low temporal locality in eADR mode – Results show ratio of latency when no data is flushed to latency when only slot data is flushed.

store fences are required to implement persistence barriers. This is because the volatile cache is within the persistence domain in eADR mode. Further, there is only a single linearization point, which involves setting the push/pop position to `INT_MAX`. On recovery, checking the values of the position pointers for each thread enables us to identify in-flight operations and roll them back. After the roll-back is complete, we consolidate the buffer, just as in the TX design.

Evaluation. To the best of our knowledge, TX and TX-free are the first lock-free persistent ring buffer solutions available. Thus, we compare the performance of the proposed designs with the volatile implementation in both ADR and eADR persistence modes. We use 4KB slot size and 32K slots so that the working set is 3-4 times larger than the LLC size.

We first examine the effects of flushing data with low temporal locality in eADR mode. We modify our designs to flush slot data (but not per-thread buffers or transaction state) on each push operation and measure latency. Figure 15 shows the normalized latency for TX and TX-free designs in eADR mode. Except for average push TX latency, all other latency values are similar or higher when slot data is not flushed from the cache. As we observed from I5 results, not flushing the cache can lower bandwidth when using DCPMM. Here, we find that not flushing non-critical data can increase latency as well. The reason for this is that data in the ring buffer has low temporal locality and competes with other critical data for cache space. This reduces the cache hit rate of the per-thread buffers and transaction state which have high temporal locality and increases overall latency. Push latency is less affected than pop latency because not flushing the slots removes an expensive persistence operation off the critical path for push operations. This is also the reason why TX has slightly better push latency. Finally, we can also observe that TX-free is less affected as compared to TX. This is because TX-free does not use transactions and hence has a lower overall memory footprint, so the cache hit rate of critical data is less affected. **Overall, flushing data with low temporal locality is good for lowering latency**. Although this observation appears trivial and has been observed in prior literature [4, 64] with the use of non-temporal stores, its impact is more pronounced with PMEM because its latency is much higher than DRAM. Therefore, applying this observation can significantly impact performance. Based on these results, we flush slot data in eADR mode for all subsequent experiments.

We also measure the latency of each operation with 50% pops and 50% pushes at full subscription (28 threads). Figure 16(a) shows the average, p99, and p99.9 latency of different implementations. We observe that TX in ADR mode shows the worst performance.



(a) Operation Latency (b) PMEM Hardware Counters

Figure 16: Lock-free Ring Buffer Evaluation – (a) Latency for 50% pops and 50% pushes with 4KB slot size at 28 threads, (b) Performance trends via low-level PMEM counters. The y-axis represents number of operations (in millions) for L3 misses and data size (in GB) for PMEM and media read/write.

This is expected because it requires expensive transactions and cache-line flushes to ensure atomicity and durability. All changes to PMEM data need to be logged and as we have seen from I1 and I2 results, PMEM writes have high overhead. We can also observe that in eADR mode, TX-free has much better performance than TX. Even though cache-line flushes can be avoided for TX in eADR mode, the overhead of transactions is high. However, TX-free avoids both cache flush and transaction overheads and achieves near volatile performance. In fact, P99 latency for TX-free is only 2x that of the volatile design. To better understand the reasons for these performance trends, we examine low-level PMEM counters for the entire duration of our experiments. Results are presented in Figure 16(b). PMEM read/write represent data exchanged with the PMEM controller while media read/write represent data exchanged with the internal 3D-XPoint media in DCPMM. We find that TX (ADR) has the highest number of L3 misses and PMEM operations. In comparison, TX (eADR) significantly reduces L3 misses and PMEM reads. This is because we do not flush transaction data and per-thread buffers to PMEM, so load operations are more likely to be serviced from the CPU cache. PMEM and media writes are also reduced by avoiding some cache-flush operations. Finally, TX-free further reduces PMEM and media writes because transactional PMEM updates are not required anymore. An interesting observation is that the media reads remain largely unchanged for all implementations. This implies that reads are mostly serviced from the XPBuffer in DCPMM.

6.4 Key Insights

The main takeaways from our analysis are twofold. First, transactions on PMEM have high overhead; so, **transactions should be avoided to the extent possible**. Per-thread logging and dirty-bit design are two alternatives that can be used to avoid transactions. As we observed from our evaluation, per-thread logging is more optimal for read-heavy workloads while the dirty-bit design is better for update-heavy workloads. Also, in some cases it may not be possible to avoid transactions. Therefore, choosing the correct design technique is important for both performance and correctness. Second, we conclude that ADR-based designs do not necessarily provide the best performance in eADR mode. **To attain optimal performance, algorithms should be specifically designed for**

Table 4: System Design Recommendations. Performance Impact – N/A: No impact, Low: <1.5x, Moderate: 1.5-3x, High: >3x.

ID	Rule	Recommendation	Impact
R1	Avoid p-stores	Eliminate unnecessary stores and/or cache stores in DRAM	High (C1–C4)
R2	Avoid concurrent access to PMEM for p-stores	Limit number of threads writing to PMEM	High (C4), N/A (C1–C3)
R3	Avoid small-sized IO	Reduce small-sized IOs and use store+cache-flush, if unavoidable	High (C3,C4), Moderate (C1,C2)
R4	Limit NUMA access	Put data structures with higher EBR on remote NUMA PMEM	High (C2,C4), Moderate (C1,C3)
R5	Use p-stores for non-critical data in eADR mode	Flush data with low temporal locality from cache	High (C2,C4), Low (C1,C3)
R6	Avoid random IO	Use techniques like write buffering and log structuring	High (C2,C4), Moderate (C1,C3)
R7	Avoid transactions	Use techniques like per-thread logging or a dirty-bit design	High (C1–C4)

the eADR mode by taking advantage of the immediate persistence of any visible operation. Our results show that this is more applicable for update-heavy workloads.

7 DISCUSSION

In this section, we discuss interesting results from our experimental analysis. Further, we combine the lessons we learnt from our analysis to propose PMEM system design recommendations.

7.1 ISA Support

The types of persistence instructions available play an important role in PMEM performance. There are three categories of persistence instructions – ① cache-flush (such as `clflush` and `clflushopt`), ② write-back (such as `clwb`), and ③ non-temporal (such as nt-store). ① and ② require data to be in the cache while ③ bypasses the cache. ① invalidates the cache-line from all levels and writes dirty data to PMEM. ② does the same but does not invalidate the cache-line, and thus improves performance by avoiding the need to read the cache-line back on a subsequent load and increasing cache hit rate. By bypassing the cache hierarchy, ③ achieves higher bandwidth compared to ① and ② but has higher latency. Therefore, it may be useful for large IO sizes or data with low temporal locality. In all other cases, ② will likely perform better. ① should be used only if it is known that flushed data will not be read again. This is because it reduces the cache hit rate, which adversely affects performance, as we have shown in Figure 3(a).

The `clwb` write-back instruction is now part of the ISA on new Intel CPUs but as of now its behavior is the same as cache-flush instructions [53]. Therefore, we were unable to test the performance benefits of write-back over cache-flush. Once it is implemented as expected, it will be an interesting avenue for future research.

7.2 Persistence Mode

As we have observed in §6, the persistence mode has both performance and design implications. In eADR mode, the cache does not need to be flushed, so the cost of persistence is significantly lowered. However, our results (see Figures 6 and 15) have shown that avoiding the flush operation reduces bandwidth and increases latency when using DCPMM. This is particularly true if data with low temporal locality is not flushed. This observation shows that even in eADR mode, some data should be flushed to achieve good performance. We expect that this trend will mostly be applicable to DCPMM.

In terms of design implications, eADR mode allows programmers to eliminate or reduce the requirement of transactions. Our results show that for applications where persistence barriers comprise a significant portion of overall time, optimizing the system design specifically for eADR mode has useful performance gains. For instance, our Log-free linkedlist design for eADR shows marginal improvement over the ADR design, TLog. This is because linkedlist workloads are read heavy. On the other hand, our TX-free ring buffer design for eADR significantly outperforms the ADR TX design. In this case, the ring buffer workloads are update heavy, which is why we see drastic improvement. Therefore, it is quite important to consider persistence mode while designing a PMEM storage system.

7.3 Internal Cache

DCPMM uses an internal cache to buffer PMEM reads and writes. Our experiments have shown that the internal cache is actually the root cause behind many of its idiosyncrasies (including I4, I5, I6, and I7 from Table 1). The reason is that the XPBuffer in DCPMM uses the XPPrefetcher to preload lines. As a result, there are two prefetchers (one in CPU cache and one within DCPMM) in the critical path. In general prefetching logic is more suited for sequential IO than random IO. Therefore, DCPMM sequential IO is much faster than random IO. Any PMEM access where sequential IO is converted to random IO (such as using stores instead of p-stores or NUMA access to PMEM) will perform poorly. Based on these observations, we conclude that **future PMEM which contains an internal cache with a prefetcher will likely show idiosyncrasies I4, I5, and I6, just like DCPMM**. In addition, the ability to control the prefetching logic of the cache in software will be useful in optimizing latency critical IO. The current generation of DCPMM does not provide this support, but if it is available in the future, it will be interesting to see its performance impact.

7.4 IO Amplification

The mismatch between the DDR-T protocol and 3D-XPoint access granularities in DCPMM causes IO amplification, resulting in reduced bandwidth utilization. This can be mitigated to some extent by using large access sizes but remains a relevant performance anomaly. We expect that **future PMEM with a mismatch in access granularities will result in similar idiosyncratic behavior (such as I3)**. To avoid this problem, PMEM should be designed to either match or minimize the difference between access granularities.

7.5 Recommendations

Based on our observations in this paper, we present a set of system design recommendations as well as their performance impact on different classes of PMEM. Table 4 shows the consolidated set of recommendations along with their impact. We compute the impact of each recommendation based on the combined quantitative impact of the idiosyncrasies that the recommendation helps manage, as observed in our empirical evaluation (§4,§5,§6). The recommendations are as follows. R1 follows directly from I1 and I2 – p-stores must reach PMEM and are more expensive than loads. Reducing unnecessary stores is, therefore, a good design philosophy. R2 is based on results we observed during I2 experiments. We achieve maximum PMEM bandwidth with just 8 threads as opposed to 28 threads with DRAM. Limiting concurrent writers to PMEM is an efficient way to maximize bandwidth utilization. R3 follows directly from I3 – using small-sized IO results in lower bandwidth, particularly for DCPMM due to internal IO amplification. A good programming practice should be to combine several smaller stores into a single large store. In cases where small stores cannot be avoided, a store+cache-flush should be preferred over nt-store because it has been shown to have lower latency [53]. R4 is inferred from I4 as well as our NUMA study with MongoDB. As we concluded from the study, placing data structures with higher EBR on remote NUMA results in maximum capacity utilization with minimal performance loss. R5 is a result of I5 and our ring buffer evaluation – p-stores achieve higher bandwidth than regular stores and using p-stores on non-critical data improves latency. So, there is merit in flushing irrelevant data from the cache, even in eADR mode. R6 follows directly from I6 – random IO is worse than sequential IO, particularly for DCPMM. There are several existing techniques which can be applied to fix this issue, including log structuring and write buffering. Both techniques help convert small random IO into larger sequential IO. R7 is based on results we observed in our lock-free case study. This recommendation is more applicable to eADR mode where it is easier to avoid transactions. There are several techniques which can be used to avoid transactions. One is per-thread logging, which we use in our linkedlist design and is also used by PMDK. Another is using a *dirty-bit* design, as used in [56, 65]. We believe that these recommendations will serve as guidelines to any researcher designing PMEM-based storage systems.

8 RELATED WORK

In this section, we discuss relevant related work on PMEM evaluation and lock-free persistent data structure design.

PMEM Evaluation Studies. There are several studies [18, 25, 26, 35, 39, 48, 53, 58, 62, 63] which evaluate both low-level as well as system-level performance characteristics of DCPMM. Unlike this paper, all of these studies present guidelines that are only applicable to DCPMM. This work was done in parallel with [63], so there is overlap in some of the analysis. In particular, idiosyncrasies I2, I3, and I4 are presented in both works, but others (I1, I5-I7) are unique to this paper. While their work was only targetted for DCPMM, our work seeks to understand the root cause of each idiosyncrasy and understand its applicability to future PMEM. Even though some idiosyncrasies were discussed in prior literature [5, 57, 64], their impact is more pronounced for DCPMM, a fact that was not known before. Therefore, the novelty of our work is in not only identifying the larger performance impact of these idiosyncrasies but also in pinpointing the exact reason for this additional impact and understanding their applicability to other classes of PMEM. In addition, while prior literature recommends completely avoiding NUMA accesses, our results show that carefully placing data on remote NUMA can help maximize capacity utilization with minimal impact on performance. We proposed a new metric, called EBR, to guide NUMA-aware data placement (data structures with high EBR can be placed on remote NUMA). Unlike EWR proposed in [63], EBR is applicable to other classes of PMEM.

Lock-free Persistent Data Structures. Several works have presented the design of lock-free persistent data structures. [16] presents the design of lock-free queues with different durability requirements. The focus of their work is on ensuring a consistent view of the queue in the event of failure. Zuriel et al. [65] extend their work to design lock-free durable sets, including linkedlist, skiplist, and a hashmap. They improve performance by reducing the number of persistence barriers required to ensure consistency. We compare with their linkedlist implementation in §6. NVC-Hashmap [49] reduces persistence overhead by only flushing cache-lines at linearization points. `PMwCAS` [56] is an easy to use multi-word `compare-and-swap` primitive for PMEM which simplifies the design of lock-free data structures. Log-free data structures [10] uses dirty bits and write buffering to avoid the need for logging and improve data structure performance. CDDS [54] uses a multi-version approach to implement persistent data structures and avoid logging. All of these works evaluate with emulated PMEM and only consider the ADR persistence mode. Therefore, our work is unique in evaluating with real PMEM and also considering both persistence modes for data structure design.

9 CONCLUSION

In this paper, we proposed PMIdioBench, a micro-benchmark suite to categorize the idiosyncrasies of real PMEM through targeted experiments and further understand the applicability of its characteristics on other classes of PMEM. Based on generic PMEM characteristics, we conducted two in-depth studies, one to guide the placement of data structures on NUMA-enabled systems to maximize capacity utilization, the other to guide the design of lock-free data structures on both ADR and eADR systems. Our analyses lead us to make interesting observations about PMEM behavior which highlight the challenges of programming with PMEM. We distilled the information gathered as empirically verified technology agnostic system design recommendations. We believe that this paper will be useful to a wide range of researchers with different specializations. In the future, we plan to continuously enrich PMIdioBench benchmarks and tools for emerging PMEM platforms. We also plan to evaluate with more storage systems.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their valuable feedback. We would also like to thank Tuong Nguyen, Abel Gebrezgi, Jonathan Burton, and Timothy Witham from Intel for their support and help in access to the test platform. This work was supported in part by NSF research grant CCF #1822987.

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98 (2010), 2237–2251.

[2] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.

[3] Daniel Castro, Paolo Romano, and Joao Barreto. 2019. Hardware Transactional Memory meets Memory Persistency. *J. Parallel and Distrib. Comput.* 130 (2019), 63–79.

[4] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. In *Proceedings of the VLDB Endowment*, Vol. 8. 497–508.

[5] Feng Chen, David A Koufaty, and Xiaodong Zhang. 2009. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *SIGMETRICS'09*. 181–192.

[6] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS'20*. 1077–1091.

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *ASPLOS'11*. 105–118.

[8] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-Addressable, Persistent Memory. In *SOSP'09*. 133–146.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC'10*. 143–154.

[10] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *USENIX ATC'18*. 373–386.

[11] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *SOSP'19*. 478–493.

[12] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *EuroSys'14*. 15.

[13] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *EuroSys'18*. 42.

[14] FAST'19. 2019. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. 221–234.

[15] Keir Fraser. 2004. *Practical Lock-Freedom*. Ph.D. Dissertation. University of Cambridge. UCAM-CL-TR-579.

[16] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *PPoPP'18*. 28–40.

[17] Kaan Genç, Michael D Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-Compatible Persistent Transactions. In *POPL'20*. 59–74.

[18] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets using Intel Optane DC Persistent Memory. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1304–1318.

[19] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. 2016. Using Storage Class Memory Efficiently for an In-Memory Database. In *SYSTOR'16*. 21.

[20] Brendan Gregg. 2016. The Flame Graph. *Commun. ACM* 59, 6 (2016), 48–57.

[21] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance with 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.

[22] Swapnil Haria, Mark D Hill, and Michael M Swift. 2020. MOD: Minimally Ordered Durable Data Structures for Persistent Memory. In *ASPLOS'20*. 775–788.

[23] Timothy L Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *International Symposium on Distributed Computing*. 300–314.

[24] Maurice Herlihy and Nir Shavit. 2008. The Art of Multiprocessor Programming.

[25] Takahiro Hirofuchi and Ryousei Takano. 2019. The Preliminary Evaluation of a Hypervisor-based Virtualization Mechanism for Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1907.12014* (2019).

[26] Takahiro Hirofuchi and Ryousei Takano. 2020. A Prompt Report on the Performance of Intel Optane DC Persistent Memory Module. *IEICE Transactions on Information and Systems* 103, 5 (2020), 1168–1172.

[27] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. 8, 4 (2014), 389–400.

[28] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *USENIX ATC'18*. 967–979.

[29] Intel. 2014. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html (accessed Dec. 2020).

[30] Intel. 2014. PMDK. https://github.com/pmem/pmdk (accessed Dec. 2020).

[31] Intel. 2016. Persistent Memory Storage Engine for MongoDB. https://github.com/pmem/pmse (accessed Dec. 2020).

[32] Intel. 2017. Processor Counter Monitor. https://github.com/opcm/pcm (accessed Dec. 2020).

[33] Intel. 2018. libpmemobj-cpp. https://github.com/pmem/libpmemobj-cpp (accessed Dec. 2020).

[34] Intel. 2019. 2019 Annual Report. https://annualreport.intc.com/Y2019/default.aspx (accessed Dec. 2020).

[35] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019).

[36] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *SOSP'19*. 494–508.

[37] Alexander Krizhanovsky. 2013. Lock-Free Multi-Producer Multi-Consumer Queue on Ring Buffer. *Linux Journal* 2013, 228 (2013), 4.

[38] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA'09*. 2–13.

[39] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019), 574–587.

[40] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *ASPLOS'17*. 329–343.

[41] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1147–1161.

[42] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: An RDMA-enabled Distributed Persistent Memory File System. In *USENIX ATC'17*. 773–785.

[43] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *ASPLOS'20*. 757–773.

[44] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *ASPLOS'20*. 789–806.

[45] MongoDB Inc. 2009. MongoDB. https://www.mongodb.com/ (accessed Dec. 2020).

[46] Dushyanth Narayanan and Orion Hodson. 2012. Whole-System Persistence. In *ASPLOS'12*. 401–410.

[47] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. 2017. SQL Statement Logging for Making SQLite Truly Lite. 11, 4 (2017), 513–525.

[48] Ivy B Peng, Maya B Gokhale, and Eric W Green. 2019. System Evaluation of the Intel Optane Byte-Addressable NVM. In *MEMSYS'19*. 304–315.

[49] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap for Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. 1–8.

[50] Steve Stargall. 2019. *Programming Persistent Memory: A Comprehensive Guide for Developers*.

[51] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The Missing Memristor Found. *Nature* 453, 7191 (2008), 80.

[52] AA Tulapurkar, Y Suzuki, A Fukushima, H Kubota, H Maehara, K Tsunekawa, DD Djayaprawira, N Watanabe, and S Yuasa. 2005. Spin-Torque Diode Effect in Magnetic Tunnel Junctions. *Nature* 438, 7066 (2005), 339.

[53] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *DaMoN'19*. 1–7.

[54] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST'11*, Vol. 11. 61–75.

[55] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS'11*. 91–104.

[56] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *ICDE'18*. 461–472.

[57] Yongkun Wang, Kazuo Goda, Miyuki Nakano, and Masaru Kitsuregawa. 2010. Early Experience and Evaluation of File Systems on SSD with Database Applications. In *International Conference on Networking, Architecture, and Storage*. 467–476.

[58] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *SC'19*. 1–19.

[59] Xiaojian Wu and AL Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *SC'11*. 39.

[60] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX ATC'17*. 349–362.

[61] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST'16*. 323–338.

[62] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2019. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. *arXiv preprint arXiv:1908.03583* (2019).

[63] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST'20*. 169–182.

[64] Xi Yang, Stephen M Blackburn, Daniel Frampton, Jennifer B Sartor, and Kathryn S McKinley. 2011. Why Nothing Matters: The Impact of Zeroing. In *OOPSLA'11*. 307–324.

[65] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. In *OOPSLA'19*. 1–26.