Characterizing the Performance of Intel Optane Persistent Memory

– A Close Look at its On-DIMM Buffering

Abstract

We present a comprehensive and in-depth study of Intel Optane DC persistent memory (DCPMM). Our focus is on exploring the internal design of DCPMM's on-DIMM read-write buffering and its impacts on application-perceived performance, read and write amplifications, the overhead of different types of persists, and the tradeoffs between persistency models. While our measurements confirm the results of the existing profiling studies, we have new discoveries and offer new insights. Notably, we find that there are separate on-DIMM read and write buffers under different management in DCPMM. Comparable in size, the two buffers serve distinct purposes. The read buffer offers higher concurrency and effective on-DIMM prefetching, leading to high read bandwidth and superior sequential performance. However, it does not help hide media access latency. In contrast, the write buffer offers limited concurrency but is a critical stage in a pipeline that supports asynchronous write in the DDR-T protocol. Surprisingly, in addition to write coalescing, the write buffer delivers lower than read, and consistent write latency regardless of the working set size, the type of write, the access pattern, or the persistency model.

Our proposition is to decouple read and write in the performance analysis and optimization of persistent programs. We present two case studies based on this insight and demonstrate considerable performance improvements with two representative persistent data structures. We verify the results on two generations of DCPMM.

ACM Reference Format:

. 2021. Characterizing the Performance of Intel Optane Persistent Memory: – A Close Look at its On-DIMM Buffering. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As the first non-volatile memory DIMM becomes commercially available with the release of Intel Optane DC persistent memory (DCPMM), there is a growing interest in characterizing its performance and understanding the implications for programming persistent memory (PM) and designing persistent data structures. Along with Intel's latest processors, DCPMM provides both byte-addressability and persistence. It offers 8x capacity and significant energy savings compared with dynamic random-access memory (DRAM), though incurring higher access latency and sustaining lower bandwidth. However, recent studies [8, 13, 21, 24, 27, 29, 32, 33] have found that DCPMM should not simply be treated as slower, persistent DRAM. Compared to DRAM, DCPMM exhibits complicated behaviors and drastically changing performance depending on access size, access type and pattern.

Since documentation on DCPMM's internal architecture is not yet publicly available, understanding its performance is important to building high-performance and efficient persistent applications. DCPMM differs from DRAM in several ways. First, there is a mismatch between CPU cacheline access granularity (64-byte) and the 3D-Xpoint media access granularity (256-byte XPLine) in DCPMM, which leads to write or read amplifications if data access size is smaller than 256 bytes. Second, to bridge the gap in access granularity, DCPMM is equipped with complex on-DIMM buffering to support read-modify-write operations and write combining for small writes. Third, the new DDR-T protocol that connects DCPMM and the integrated memory controller (iMC) supports asynchronous stores to hide the long write latency to the physical media while the DDR4 protocol used by DRAM is synchronous for both loads and stores.

These differences could lead to various performance implications for persistent software designs, which have not been thoroughly studied. This paper seeks to understand the design of DCPMM's on-DIMM read-write buffering and its interactions with the CPU caches as well as the DDR-T protocol. Most importantly, we aim to understand how on-DIMM buffering affects application-perceived performance. We evaluate the existing two generations of DCPMM with microbenchmarks and have the following findings that were not previously reported.

- DCPMM manages read and write separately with dedicated on-DIMM buffer spaces. The read buffer is exclusive to the CPU caches and has no significant impact on the performance of random reads since CPU caches are several orders of magnitude larger than the read buffer. However, read buffering and its adjacent XPLine prefetching is important to improving sequential read performance. Most importantly, it helps hide write latency by allowing direct updates to XPLines already in the read buffer, avoiding the " read" in expensive read-modify-write operations.

- The write-combining (write) buffer is effective in merging small writes and reducing write amplification. Given its small size (16 KB), however, it is challenging for programs to exploit locality and hit the write buffer. Notably, together with the asynchronous DDR-T protocol, the write buffer helps absorb writes at a rate much faster than the underlying media and effectively sustains write latency at a low level comparable to that of DRAM. Write latency is consistent across various working set sizes (WSS), even when the WSS is an order of magnitude larger than the capacity of the write buffer.
- Contrary to popular belief that write performance is worse than read in DCPMM, we find that for data stores with weak or little locality that comprise of frequent random reads (often in the form of pointer chasing) followed by writes and persists, a typical access pattern found in linked lists, hash tables, and balanced search trees, the overall latency is bottlenecked by expensive random media reads.
- Due to the asynchronous DDR-T protocol, cacheline flushes or ordinary writes return when they reach the write pending queue in the iMC to hide the long media write latency. Fence instructions, which order read and write operations for crash-consistency, only guarantee that flushes are globally visible but not necessarily completed. Thus, reading a recently flushed cacheline after fence instructions return could experience almost an order of magnitude longer latency as the read needs to wait for the flush to complete.

Note that the latest Intel Xeon scalable processors and the 2nd generation DCPMM propose a significant change to make the CPU caches persistent upon a crash. If enabled by the platform (e.g., the motherboard), no cacheline flushes are needed for persistence. While this new feature is still being evaluated by vendors, the new platform is not yet widely available. We verify the aforementioned findings in the 2nd generation DCPMM without platform support for cache persistence.

We present two optimizations inspired by the insights in two representative persistent data structures. Unlike the existing work that mostly focuses on optimizing write performance, we demonstrate that random reads could become a major bottleneck in large data stores with weak locality. Given that persistent writes are constrained by fence instructions and cannot fully utilize memory bandwidth, we devise a *speculative helper thread* approach to prefetch data for a worker thread before a random access to the data occurs. The helper thread, which is constructed by extracting load instructions from a worker thread, has a 100% prefetching accuracy and is independent of as well as faster than the worker thread. Experiments with the cacheline-conscious extensible hash table (CCEH) show significant improvements in latency and throughput for write-intensive workloads.

B+-trees that store sorted keys in contiguous memory spaces are susceptible to long read-after-persist delays. Key insertions using in-place updates require on average half of the keys in a node to be shifted and cause repeated reads and writes to the same cacheline. Frequent cacheline flushes and fences after key shifts are considered a major performance bottleneck. We demonstrate that using out-of-place redo logging can effectively improve the latency and throughput of insertions in B+-tree despite the logging leads to doubled PM writes. This case study suggests that the trade-off between packing data items for exploiting locality and reducing the overhead of persistence should be examined when designing persistent data structures.

2 Background and Motivation

In this section, we provide background on Intel's Optane persistent memory and discuss the configurations of our testbed, the tools used to profile DCPMM's performance, and the methodology to design the benchmark programs.

2.1 Optane persistent memory

Intel Optane persistent memory is the first commercially available non-volatile DIMM. Along with Intel's latest scalable processors, DCPMM provides both byte-addressability and persistence. Sitting on the memory bus, DCPMM connects to the integrated memory controller (iMC) through a new DDR-T communication protocol. Similar to DRAM DIMMs, Optane DIMMs provide the processor with cacheline (64-byte) access granularity, but the physical 3D-Xpoint media access granularity is 256 bytes. The mismatch in access granularity causes write amplification because writes smaller than 256 bytes become read-modify-write operations and result in 256-byte writes to the physical media. To reduce write amplification, DCPMM employs an on-DIMM write-combining buffer to merge adjacent small writes. Recent studies [27, 32] on Optane PM independently verified that the size of the write buffer is 16 KB.

To ensure persistence, the iMC maintains an asynchronous DRAM refresh (ADR) domain. CPU stores that reach the ADR domain will be persisted to DCPMM upon a power failure. The iMC also maintains separate read (RPQ) and write pending queues (WPQ) for each Optane DIMM. Both the WPQ and the write-combining buffer belong to the ADR domain. In the 2nd generation Optane, the extended ADR (eADR) domain includes CPU caches. There are two modes of operation in DCPMM: *memory* mode and *app-direct* mode. The memory mode does not support persistence, and Optane

DIMMs are used as volatile DIMMs. In *app-direct* mode, Optane DIMMs appear as persistent memory, and applications can directly access them using CPU load/store instructions.

To enforce crash consistency, stores to DCPMM must be properly ordered. Since evictions from the CPU cache hierarchy may not follow store order, in the 1st generation Optane, to persist a data object, store instructions are followed by cache line flushes [1] (e.g., `clflush`, `clflushopt`, or `clwb`) and a memory barrier (e.g., `sfence` or `mfence`). The barrier ensures that cache line flushes are accepted to the WPQ and reach the ADR domain. The return of a barrier instruction guarantees that all stores prior to the barrier are persisted. The cache-line flush and the following fence instruction together are usually referred as a *persistence barrier*. In the 2nd generation Optane, cacheline flushes are not needed since CPU caches belong to the eADR domain and are persistent. However, eADR requires platform support and a much larger power reserve. As of the time of writing, platforms that support eADR are not widely available. The 2nd generation Optane testbed we evaluated is equipped with the 200 series Optane DIMMs and the latest Xeon scalable processors but with eADR disabled.

2.2 Known performance characteristics

Previous studies found that the behavior of Optane DIMMs is different from that of DRAM DIMMs in several ways: **1)** Read and write performance in Optane DIMMs are asymmetric, and the performance gap is much larger than that in DRAM DIMMs. Maximal read bandwidth is 3x of the maximal write bandwidth, and write performance does not scale beyond a small thread count. **2)** Surprisingly, read latency on Optane DIMMs is significantly higher than write latency because writes commit once data reaches the ADR domain while reads need to fetch data from the 3D-Xpoint media. **3)** Optane DIMM performance is strongly dependent on access size. Reads and writes smaller than the 256-byte physical media access granularity are inefficient. Small writes cause write amplification in which the amount of data written to the media is larger than that issued by the iMC.

2.3 Motivation

As documentation on the architecture of DCPMM is scarce, it is important to study how the integration of Optane DIMMs in the memory hierarchy, in addition to multi-level CPU caches and the DRAM, affects application-perceived performance. We seek to connect high-level performance, such as throughput and latency, to low-level statistics on DCPMM in order to infer its internal design. Notably, three architectural characteristics of DCPMM deserve investigation.

- The mismatch between cacheline and 3D-Xpoint media access granularity not only causes write amplifications but also read amplifications. While write amplification is an indication of inefficiency, read amplification opens up opportunities for potential data reuse as the additional data not demanded by the iMC may be buffered on Optane DIMMs and can be accessed at a lower cost. Read buffering has a slew of implications for performance, including its interactions with the CPU caches, its impact on `clwb` and `nt-store` as well as the potential benefit of reducing the cost of read-modify-write operations.
- There are still unknowns about write buffering, including its eviction and write-back policies. Since write amplification can have salient impact on performance, understanding its internal management helps reason about performance anomalies and fluctuations in write. Beyond reducing write amplification, it is also important to investigate how well write buffering, together with the asynchronous DDR-T protocol, bridges the latency gap between CPU caches and the physical media.
- The performance implications of persistence barriers are not thoroughly studied. While store fences are extensively used for weakly-ordered memory models in volatile memory, they only guarantee global visibility of stores to ensure memory consistency. Cacheline flushes and non-temporal stores, however, still take substantial time to reach PM after fences return. The delay can have a sizable impact on the performance of the following data accesses.

We are also interested in how these architectural characteristics evolve in different generations of DCPMM.

2.4 Methodology

We design micro-benchmarks to generate various controlled access patterns in PM and measure the following metrics to infer the internal working of Optane DIMMs.

Metrics. *Write amplification* (WA) is the ratio of the number of bytes written to the 3D-Xpoint media divided by the bytes issued by the iMC. Similarly, *read amplification* (RA) is the ratio of the actual data read from the media divided by data requested by iMC. A WA or RA value larger than 1 indicates that more data is written to or read from the media than requested due to the mismatch between CPU access granularity (64B) and media access granularity (256B). On the other hand, WA and RA can also be smaller than 1 when repeated writes/reads hit the on-DIMM (write-combining) buffer(s), avoiding accessing the physical media. In general, WA and RA should be bounded by 4 (i.e., $\frac{256B}{64B}$).

Platform. We have two test beds equipped with two generations of DCPMM, respectively. The two machines only differ in the processors and share the memory as well as the software configurations. The server with the 1st generation (G1) DCPMM had dual Intel Xeon Gold 6320 2.1GHz CPUs with 32KB L1i/L1d cache, 1MB L2 cache, and 27.5MB L3 cache, while the 2nd generation (G2) Optane server had dual Intel Xeon Gold 5317 3GHz CPUs with 1.1MB L1d/768KB

[1]Non-temporal stores can be used to bypass the cache and avoid cache line flushes.

L1i cache, 30MB L2 cache, and 36MB L3 cache. Both were equipped with 192GB DRAM and six 128GB Optane DIMMs. All the DRAM and Optane DIMMs were installed in one CPU socket to isolate Optane performance characteristics from the non-uniform memory-access (NUMA) factor. We used Ubuntu 20.04 as the operating system and the `DAX` mode in the `ext4` file system to mount PM to the program's address space. We used `ipmwatch` from the Intel VTune profiler to measure data access in the 3D-Xpoint media and issued by the iMC.

3 Read-Write Buffering in Optane DIMMs

This section seeks to understand the internal working of the Optane DIMM by observing its behavior running carefully crafted micro-benchmarks.

3.1 Read buffering

Existing studies [27, 32] have confirmed the existence of a 16KB read-modify-write (RMW) buffer in Optane DIMMs where adjacent writes could be temporarily stored and merged. In [32], experiments demonstrated that reads can also be cached and compete for space in the RMW buffer. In what follows, we demonstrate that there exists a 16 KB read buffer separate from the RMW buffer in Optane DIMMs. The read buffer is *exclusive* with regard to the CPU caches, i.e., only containing data that is not present in the CPU caches.

Benchmark. To prove the existence of an on-DIMM read buffer and estimate its size, we design a single-threaded, micro-benchmark that repeatedly reads from a contiguous persistent memory region and measure read amplification (RA). As shown in Figure 1, the benchmark uses strided read aligned with the 256B 3D-Xpoint access granularity (referred as an *XPLine*). Note that each XPLine contains four cachelines which can be read from the media in one transaction. A stride of 256B ensures that two consecutive reads hit two separate XPLines and require two media reads. The benchmark reads one cacheline from each XPLine at a time until reaching the end of the memory region and repeats this pattern by reading another cacheline from each XPLine. The benchmark has two configurable parameters: the number of Cache lines read per XPLine (CpX) and the size of the memory region, i.e., the working set size (WSS).

Findings. To ensure data is actually read from the Optane DIMM, cachelines are invalidated [2] using `clflushopt` immediately after they are read from PM. Figure 2 shows RA as CpX and WSS change. We made three observations: 1) It is evident that there exists read buffering in Optane DIMMs, otherwise RA should always be equal to 4 regardless of how many cache lines are read in an XPLine. As shown in Figure 2, with a small WSS, RA is inversely proportional to CpX, indicating that multiple cacheline reads in an XPLine only

[2]Non-temporal load is not currently implemented in Intel scalable processors and cannot bypass the CPU caches.

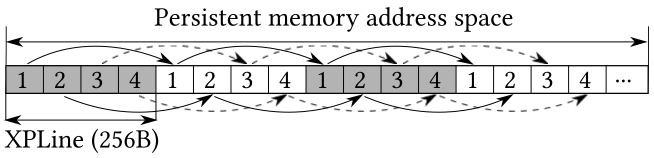

Figure 1. Inferring read buffer capacity with strided read.

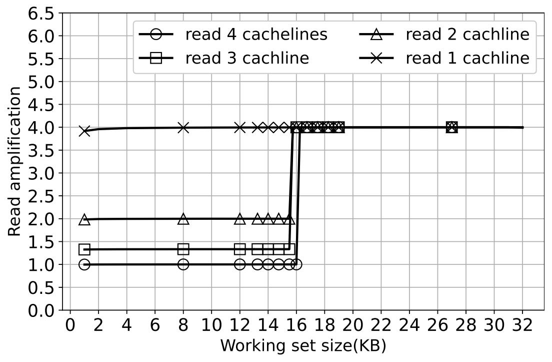

Figure 2. Read amplification due to various access patterns in a single Optane DIMM.

require one media load because later cacheline accesses hit an on-DIMM read buffer. 2) RA jumps to 4 when the WSS exceeds 16KB, suggesting that following cacheline accesses miss the read buffer. Thus, the capacity of the read buffer should be 16 KB. 3) RA never drops below 1 even when the WSS fits in the read buffer (16 KB). It suggests that a cacheline is evicted from the read buffer once it is loaded into the CPU caches, otherwise RA would be 0, i.e., all recurring reads would hit the read buffer instead of the media, when the WSS is smaller than 16 KB. **G2 DCPMM** has similar results on read amplification and a 16 KB read buffer.

Performance implications. In Figure 2, RA jumps to 4 immediately when the WSS exceeds the read buffer capacity. Similar results are observed if the access pattern is changed. This indicates that the read buffer likely employs a simple first-in-first-out (FIFO) eviction policy. Since the read buffer is exclusive to the CPU caches, which are at least two orders of magnitude larger, there is not much locality to exploit in the read buffer. However, as will be demonstrated in Section 3.2 and 3.3, the read buffer is essential for improving sequential access performance and reducing the cost of read-modify-write operations.

3.2 Data prefetching to read buffer

Cache prefetching is an effective mechanism to reduce cache miss rate and penalty, which makes sequential data access much faster than random access. This section investigates whether the Optane DIMM devises a similar prefetching mechanism to hide load latency from the media.

Benchmark. The trigger for CPU cache prefetching is usually a sequence of two or more cache misses in a certain pattern [6]. To reveal the access pattern that may trigger data prefetching to the read buffer, we devised a single-threaded benchmark that randomly accessed the Optane DIMM with a granularity ranging from 256B to 1 KB (referred as an *access block*). Because access blocks align with the XPLines, there is no read amplification. Within each access block, the benchmark sequentially accessed multiple cache lines each multiple times. After a block is accessed, it is flushed from the CPU cache to ensure that the next visit to the same block always requires access to the Optane DIMM. We measure the ratio of data loaded from the 3D-Xpoint media and that actually demanded by the benchmark to infer the *prefetching depth* (PD) of Optane DIMMs. We also measure the read ratio for the CPU cache, i.e., the ratio of iMC loaded data over program demanded data. The hardware prefetcher and the adjacent cache line prefetcher for the CPU caches were disabled in the BIOS.

Findings. Figure 3 shows that in most cases, more data is loaded from the media than actually requested by the program, indicating data prefetching to the read buffer. There are three regions in the figure according to the WSS.

- *WSS smaller than 16KB.* The prefetched data can fit in the read buffer (16KB). Unlike program-requested data that is not kept in the read buffer, the prefetched data stays in the buffer, and the following prefetch requests hit the read buffer, thereby requiring no media access.
- *WSS between 16KB and 32MB.* The working set can no longer fit in the read buffer but still fits in the last-level CPU cache (27.5MB). The read ratio of 1 on iMC confirms that there is no prefetching into the CPU cache while more data is loaded from the media than requested. The additional data is due to data prefetching to the read buffer. Note that the read ratio is bounded at different levels for different block sizes, which infers that one additional XPLine is prefetched regardless of the block size.
- *WSS larger than 32MB.* The read ratio continues to grow in the Optane DIMM, mainly due to prefetching activities in the CPU cache, as demonstrated by the larger-than-one read ratio in iMC. Since hardware and adjacent line prefetchers are disabled on the CPU, the prefetching activities are likely due to software prefetching from the compiler in response to cache misses.

We verified that there is no change to the read buffer prefetching mechanism in **G2 DCPMM**.

Performance implications. While read buffer prefetching is expected, an important finding is that on-DIMM prefetching is independent from the CPU cache prefetching as we observe similar trends when the CPU prefetchers are enabled. The CPU prefetcher works at the cacheline granularity, and the prefetch depth ranges from one to a few additional cachelines. Thus, CPU prefetching may not cross the boundary of XPLines or help hide media latency. As will be discussed in Section 3.6, read buffer prefetching is responsible for the

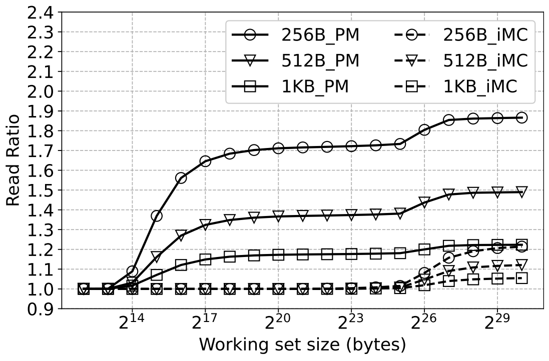

Figure 3. Inferring prefetching depth in Optane DIMMs.

large performance discrepency between sequential and random accesses in DCPMM. Thus, it is recommended that programmers use a block size of at least one XPLine (256B) to benefit from on-DIMM prefetching.

3.3 Write buffering

This section explores the design of the write-combining buffer beyond capacity analysis. We investigate whether the write buffer employs a write-back mechanism that flushes XPLines down to the media as well as its eviction algorithm.

Benchmark. We performed both full (256B) and partial writes in the Optane DIMM using non-temporal stores to compare the write-back mechanism for different types of writes. Full writes update all four cachelines in an XPLine, while partial writes only update a subset of cachelines leaving the remaining untouched. The benchmark sequentially updates cachelines within an XPLine and can be configured to either sequentially or randomly update across XPLines.

Findings. Figure 4 shows write amplification due to different types of write in **G1 DCPMM**. Note that the WA curves are independent of the access pattern across XPlines (i.e., sequential or random) and only depend on the WSS. It is evident that the write buffer manages full and partial writes differently. For partial writes, WA remains 0 until the WSS exceeds 12KB, indicating that all writes are absorbed by the write buffer, and no data is written to the media. WA jumps after the WSS goes beyond 12KB and gradually approaches the theoretical write amplification for each write pattern, e.g., a WA of 4 for writing one cacheline out of four cachelines. In contrast, full writes always lead to media write and write amplification rises to 1 even when the WSS is small (less than 4KB).

The results suggest that the size of the writer buffer is between 12KB and 16KB, and there are two write-back mechanisms in the write buffer. 1) Fully modified XPLines are written back to the media periodically, while 2) partially modified XPLines are retained in the buffer until evicted. To further understand the periodic write back mechanism, we devise a benchmark to study the relationship between

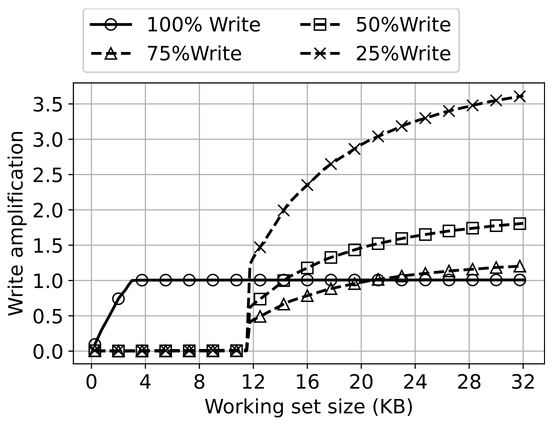

Figure 4. Write amplification in G1 Optane.

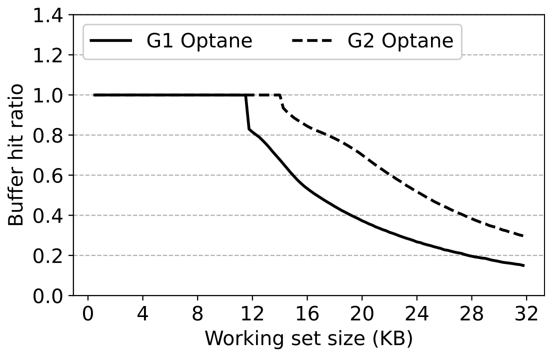

Figure 5. Write buffer hit ratio as WSS increases.

the inter-reference (IR) interval (in CPU cycles) to the same fully-written XPLine and write amplification. If the IR interval is sufficiently long, subsequent accesses to a XPLine miss the write buffer since it has been written back to the media, which will result in a WA of 1. The results suggest that fully-written XPLines are periodically written back to the media approximately every 5,000 cycles.

G2 DCPMM disables periodic XPLine write back and manages full and partial writes uniformly with the same eviction algorithm. To study the eviction algorithm, we issue random partial writes [3] and measure the amount of media write relative to program issued writes to infer the ratio of writes that hit the buffer. Figure 5 shows the write buffer hit ratio drops gracefully as the WSS exceeds buffer capacity. Contrary to the sharp climb of RA in Figure 2, WA also gracefully increases beyond the read buffer capacity. This suggests that the write buffer is managed using a different eviction algorithm than the one used for the read buffer. Since the access pattern is pure random, the hit ratio curves in Figure 5 suggest a *random* XPLine eviction algorithm.

Performance implications. Unlike reads whose performance largely depends on the effectiveness of CPU caching, writes such as cacheline flushes or non-temporal stores rely heavily on write buffering in DCPMM. Ideally, write WSS should not exceed 16 KB to maximize write buffer hit and eliminate media write, which is quite challenging if not impossible in realistic data stores. Practically, programmers 1) should coalesce small writes to form XPLine-sized writes rather than exploiting temporal locality across XPLines if the WSS cannot fit in the buffer since random eviction is employed; 2) do not need to particularly pursue sequential and avoid random access for writes because the write buffer does not preserve spatial locality across XPLines.

[3]We use partial writes because full writes are subject to periodic write back in G1 Optane.

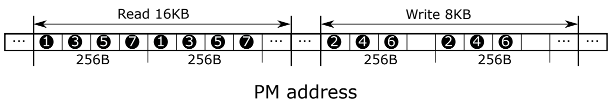

Figure 6. Interleaved read and write operation to check the relation between read and write buffer.

3.4 The relationship between read and write buffers

This section studies whether the read and write buffers share a 16KB space or are separate. Our results show that they are independent, and each has a dedicated buffer space. XPLines can be moved between the read and write buffers, likely via a tagging mechanism.

Benchmark. To study if reads and writes compete for a shared buffer space, we set up two non-overlapping regions on DCPMM: a 16KB read and an 8KB write region. As shown in Figure 6, the benchmark issues interleaved reads and writes by jumping between the two regions. The numbers indicate the sequence cachelines are read or written. To ensure data accesses reach DCPMM and bypass the CPU cache, cachelines are flushed/invalidated immediately after read from DCPMM and writes are non-temporal stores.

Findings. We compare the amount of data read from and written to the iMC and the media in the benchmark with those in two baseline programs, which access the read and write region separately. Since the individual read (16KB) and write (8KB) working sets fit in the read (16KB) and write (12-16KB) buffer, respectively, but the aggregate WSS (16KB + 8KB) does not, reads and writes would compete for buffer space and interfere with each other if the read and write buffers were a shared space. However, our results show that the benchmark with interleaved read/write accesses incurs no read amplification (RA = 1), causes no data written to the media, and behaves the same as the baseline programs. It suggests that the read and write buffers are independent.

Next, we study if data can be moved between the read and write buffers. We modify the benchmark to issue one non-temporal store to the first cacheline of an XPLine, followed by three reads to the remaining three cachelines in the same XPLine. To bypass the CPU caches, all reads are immediately flushed from the caches. We set the WSS to 8KB to guarantee that all data fits in the read or write buffer. Our measurements show that the amount of data loaded from and written to the media is far less than the amount of data issued by the iMC, which indicates that most reads and writes hit the buffers. Since reads and writes are interleaved, read can directly load data from the write buffer and write can update XPLines in the read buffer, avoiding expensive read-modify-write operations from the media. Further experiments show that XPLines updated in the read buffer are subject to the write-back policy in G1 Optane, thereby likely transitioned to become part of the writer buffer. The buffers in **G2 DCPMM** behave similarly.

Performance implications. Hitting the read or write buffer avoids expensive media read and write operations and can lead to significant performance improvements. However, it is difficult for programs to exploit locality in such small buffers. The separation and XPLine transition between the read and write buffers offer a different perspective. The expensive read-modify-write operation on PM can be broken into separate read and write operations to the same address, thereby able to be pipelined to hide media access latency. Specifically, write could hit the read buffer if the target address is prefetched. Since the read and write buffers are independently managed, it allows for great flexibility to design collaborative approaches to pipeline reads and writes.

3.5 Read-after-persist Latency

To ensure crash consistency, persistence barriers need to be placed between data accesses. A persistence barrier usually includes one or multiple `clwb` or `nt-store` followed by a `mfence` or `sfence` instruction. While memory barriers have been widely adopted for memory consistency, they may behave differently under the Optane's DDR-T protocol, which supports asynchronous command and timing. With DDR-T, memory barriers only ensure that cacheline flushes are globally visible but not necessarily completed by the time a fence instruction returns. Following accesses to addresses that have been previously persisted (flushed) may experience longer delays. As writes are asynchronous, we investigate how reads, which are synchronous under DDR-T, are affected by persistence barriers. Our primary metric is the read-after-persist (RAP) latency, which is defined as the data load time to a recently persisted address.

Benchmark. Algorithm 1 shows how we quantify the RAP latency with respect to RAP distance. The benchmark accesses data on DCPMM from low to high addresses in the granularity of cachelines. In each iteration, it first flushes one cacheline followed by a fence instruction and accesses a cacheline at a lower address that was previously persisted. The RAP distance is the distance of a cacheline the benchmark reads at a lower address relative to the current address being persisted. A small RAP distance indicates the read and persist occur in temporal proximity. The benchmark has a 4KB WSS that fits in the on-DIMM buffers.

Algorithm 1: Measure read-after-persist latency

```
   input: char *addr (PMM address), int distance, int
          wss (working set size)
1  offset=0;
2  while offset < wss do
3      mov 0x00, [addr+offset];
4      clwb [addr+offset];
5      /* Line 3-4 can be changed to nt-store */
6      mfence or sfence;
7      mov [addr+offset+(wss-distance)%wss], register0;
8      offset+=64;
9      /* move to the next cache line */
10 end
```

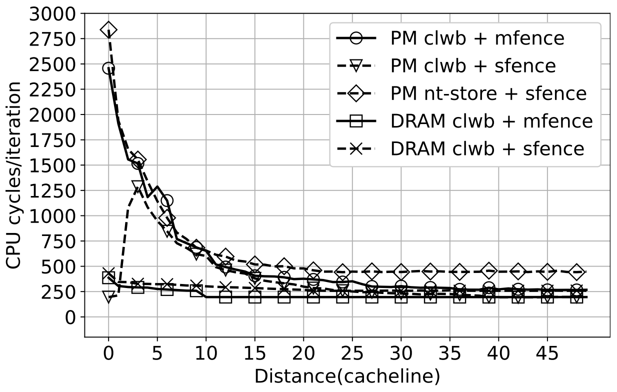

Figure 7. Per-iteration latency in Algorithm 1 as RAP distance increases. The distance is in the number of cache lines.

Findings. Figure 7 plots the average CPU cycles spent on one iteration of the benchmark as the RAP distance increases on **G1 Optane**. We investigate `clwb` and `nt-store` with different types of memory fences in both DCPMM and DRAM. As shown in Figure 7, `clwb` and `nt-store` cause significant delays to following accesses to the same address. When the RAP distance is small, i.e., accessing an address may still being persisted, the average per-iteration latency reaches up to 2,800 cycles. As RAP distance increases, the latency quickly decreases and eventually approaches that of the on-DIMM buffers. The latency gap on DCPMM is significant, by as much as 10X, while the gap on DRAM is 2X.

Interestingly, the combination of `clwb` and `sfence`, a common implementation of a persistence barrier, presents a different RAP latency profile. Reading recent persisted addresses (RAP distance ≤ 1) results in low latency comparable to that in accessing more distant addresses. However, RAP latency quickly jumps to 1,250 cycles, from where it gradually converges to the initial latency as the read moves further away from the current persisted address. Note that reads are not ordered with respect to `sfence` nor the flush, thereby able to load directly from the CPU caches. In G1 Optane, clwb invalidates the flushed cacheline. Therefore, as RAP distance increases, reads have to wait for persists to complete if they have been started, missing the opportunity to bypass the flush and load from the caches.

A major difference between G1 and **G2 Optane** is that the latter does not invalidate and allows a cacheline to remain in the cache after clwb. This effectively eliminates the RAP issue for clwb as the following read can always fetch the latest value of an address from the CPU caches. We re-test the results in Figure 7 on G2 Optane and find that the long RAP delay for nt-store still persists while the curves of RAP latency for clwb approach those on DRAM.

Performance implications. Programs that repeatedly persist and read adjacent cachelines may suffer long RAP latency, typically in data structures that pack items in contiguous memory spaces. While clwb only has the issue in G1 Optane, nt-store suffer from it in both generations of Optane. A similar problem could occur when read-write sharing a cacheline on PM across CPU sockets, e.g., multiple threads on different sockets competing for a persistent lock or a critical section. Handing over the lock between threads requires a shared cacheline to be invalidated and flushed back to PM, immediately followed by a read from another thread. Optimizations should be devised to avoid such contentious accesses to flushed cachelines.

3.6 Interactions between CPU caches and on-DIMM buffers

Previous sections focus on exploring the design of the on-DIMM read-write buffers and use benchmarks that bypass the CPU cache to measure read or write amplification. This section evaluates the performance of the DCPMM considering the CPU caches. We are particularly interested in the interactions between the caches and the on-DIMM buffers.

Benchmark. Inspired by [6], we developed a benchmark with the following building block called an *element*.

```
typedef struct  working_set_unit
{
        struct working_set_unit *next;
        uint64_t pad[NPAD];
} working_set_unit_t;
```

The benchmark consists of a predefined number of elements, each containing a pointer to the next element and a data area pad. We set the size of an element to 256B and align it with XPLines. The working set is constructed by connecting elements either sequentially or randomly to form a circular linked list. The benchmark updates one cache line in the pad area from each element and follows the next pointer to traverse the entire working set. This pattern is repeated a large number of times to obtain statistically significant results. Throughout the tests, the structure of the linked list remains unchanged. The accessed data and the pointer do not belong to the same cache line. This separation ensures that only data changed in the pad area is persisted, avoiding invalidating cached element pointers in the CPU cache.

The benchmark performs sequential/random reads and writes and reports the average latency per element. We use two types of writes (clwb and nt-store) and implement strict and relaxed persistency. In strict persistency, each write is followed by a persistence barrier (e.g., flush + fence). We adopt a simple implementation of relaxed persistency, in which all writes are allowed to be reordered and occur in parallel until the completion of the working set. A fence instruction is inserted after finishing the last element to separate two passes to the linked list. While there are many other relaxed persistency models, such as epoch and strand persistency [22], our purpose is to compare write performance with and without ordering constraints. Therefore, we choose the most strict and relaxed models.

Findings. Figure 8 (a) and (b) show the latency per element with different types of persists. Both figures show three levels of latency: 1) Latency begins with a low level and gradually ramps up as the WSS approaches 16KB – the size of the read-/write buffers; 2) latency plateaus at around 400 cycles before 3) it starts to grow drastically when the WSS exceeds 16MB. The latency of random accesses is up to more than 1000 cycles for both clwb and nt-store, which is 10X compared to that when the WSS fits in the on-DIMM buffers.

Note that the benchmark traverses the linked list via pointer chasing. Therefore, data access to each element always begins with a *read* to the first cache line. This is a common access pattern in many data structures such as trees and hash tables. Even for write-intensive workloads, e.g., those with frequent persists, the initial read could significantly affect performance. To study how read and write contribute to the overall latency, we separate reads from writes in the original benchmark. *Pure reads* only perform pointer chasing in the linked list without touching the pad data area. *Pure writes* use an array in DRAM to store the addresses of elements and perform stores directly to the pad area without reading any data from PM. The address array is randomized for random access.

Figure 8 (c) shows the latency breakdown between pure reads and writes. For WSS smaller than 16 MB, read latency remains low at 6-40 cycles, comparable to the latency of L1-L3 CPU caches. Latency climbs up to 400 and 800 cycles for sequential and random reads, respectively, as the WSS grows beyond 16 MB. A similar read latency increase at 16 MB was also observed in [27] due to the overflow of the address indirection translation (AIT) buffer in Optane DIMMs. We conjecture that the overflow of the AIT buffer and the last-level cache (27.5 MB L3 cache in our testbed) both contribute to the drastic latency increase, though it is difficult to isolate their respective effects. As the WSS exceeds cache capacity, reads must be served from the physical media, incurring at least an order of magnitude longer latency.

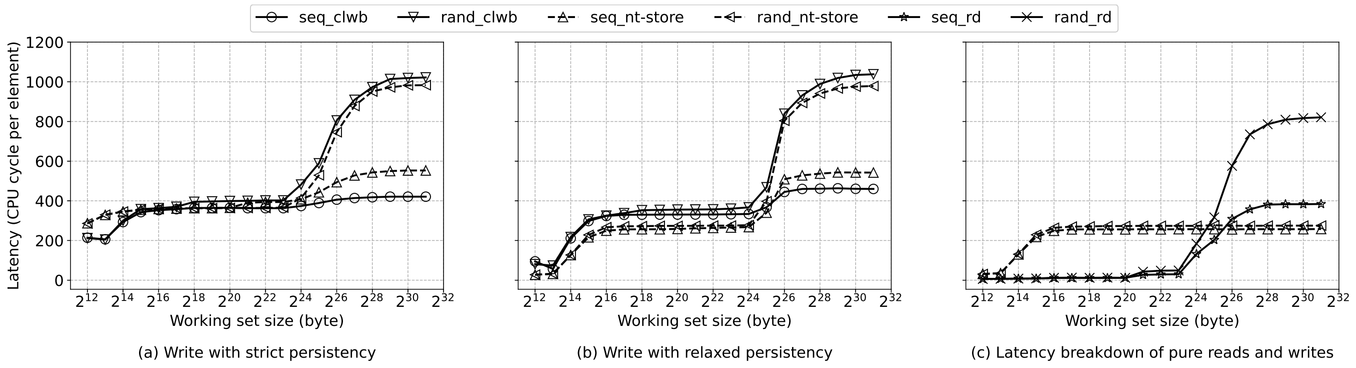

Figure 8. User-perceived latency with various working set sizes in G1 Optane. Results are similar on G2 Optane.

In contrast, write latency is consistent across different WSSes except for small WSSes that fit in the write buffer. While the SRAM-based write buffer is able to bring write latency close to that of the CPU caches, the asynchronous DDR-T protocol effectively hides write latency and keeps it below 300 cycles regardless of the WSS. As Figure 8 suggests, write outweighs read when the WSS is smaller than 16 MB while read latency dominates the overall performance thereafter. Sequential access incurs much lower latency than random access does mainly due to the reduction in read latency thanks to data prefetching to the on-DIMM read buffer. Another interesting observation is that the performance trends for `clwb` and `nt-store` are similar, though `clwb` requires one additional read in the pad area. It suggests that the first read to an XPLine is especially expensive compared to following reads to the same XPline, which would certainly hit the read buffer.

In **G2 Optane**, we observe similar performance trends except that 1) the overall latency as well as the pure read latency (in CPU cycles) beyond the L3 cache is even higher due to a higher CPU frequency in the G2 Optane server, and 2) the performance of `clwb` and `nt-store` converges when the WSS is smaller than the L3 cache size while diverging thereafter.

Performance implications. We have important takeaways from the experiments. First and foremost, the latency to load data from the media, regardless of the access pattern, outweighs write latency when the WSS exceeds the size of the last-level cache. Given that write latency is consistent regardless of the access pattern and the WSS, optimizations should be focused on reducing read latency for large workloads. Second, the performance of different types of writes (i.e., `clwb` and `nt-store`) and different types of persistency models could converge – 1) when the WSS fits in the CPU caches, the cost of `clwb` and `nt-store` are similar since reading from the cache is almost negligible compared to media write; when the WSS is larger than the L3 cache, the difference of the two is overshadowed by the first read to an element. 2) Reads are not constrained by a store fence, thereby its performance not affected by the persistency model; for `clwb` and `nt-store`, the persistency model also has a limited impact. Although the writer buffer has sufficient capacity to accept concurrent writes from a relaxed persistency model than a strict persistency model, both are bottlenecked by the limited concurrency of media write. Thus, it would be more effective to employ relaxed persistency models to reduce persists to the same XPLine than reducing the number of XPLines persisted.

4 Case Studies

In this section, we present two cases studies of applying the aforementioned insights to two representative persistent data structures.

4.1 CCEH

Cacheline-Conscious Extendible Hashing (CCEH) is an extendible hash table designed for persistent memory [19]. Extendible hashing dynamically allocates memory space for buckets on demand and allows the hash table to grow (or shrink) incrementally. For this goal, extendible hashing manages buckets in a hierarchical manner in which a top-level *directory* stores bucket addresses. Buckets can be inserted or deleted from the directory without affecting other buckets, and re-hashing can be done by changing the size of the directory. CCEH adopts cacheline-sized buckets to minimize cacheline accesses and introduces an intermediate level of indexing, *segments*, to reduce the size of the directory.

As shown in Figure 9, CCEH consists of a global directory and a large number of 16 KB segments. Each segment contains 256 cacheline-sized buckets (64B) plus 16B metadata. Key lookup in CCEH requires at least three cacheline accesses – 1) indexing a segment in the directory, 2) addressing a bucket in a segment, and 3) accessing data in the bucket. Since the three-level indexing is essentially a tree structure covering a large memory region, key insertion in CCEH likely entails three random reads followed by a write and a persistence barrier. Table 1 lists the time breakdown of key insertions in CCEH. We used YCSB [4] to insert 16 million 16B key-value pairs into the hash table and reported the time breakdown using `perf`. We configured CCEH to use 1 or

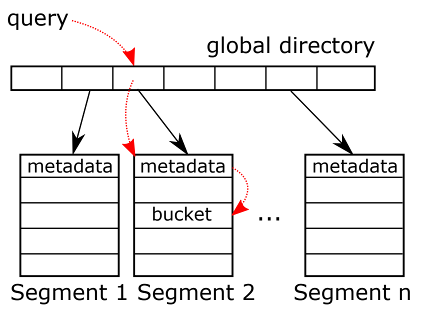

Figure 9. Workflow of accessing a bucket in CCEH.

Table 1. Time breakdown of key insertion in CCEH.

Thread/DIMM	Segment metadata	Persists	Misc.
1T/1-DIMM	51.1%	22.56%	26.34%
5T/1-DIMM	51.6%	20.94%	27.46%
1T/6-DIMM	47.2%	25.13%	27.67%
5T/6-DIMM	42.8%	26.12%	31.08%

5 threads and tested with a single non-interleaved Optane DIMM or 6 interleaved DIMMs.

The results suggest that while persists (CCEH uses `clwb` followed by a memory fence after each bucket update) are expensive, the bottleneck is accessing segment metadata, one of the three random reads. The access to segments accounts for approximately 50% of key insertion time, regardless of the number of threads or DIMMs. Among the three random access locations, the global directory fits in the CPU caches and has a strong temporal locality because it is frequently queried. CCEH employs linear probing to prevent premature segment split due to hash collision. It searches up to four adjacent buckets upon a collision, thereby exhibiting spatial locality when accessing individual buckets. This amortizes the per-bucket read latency as some likely hit the on-DIMM read buffer. Therefore, the first access to a segment, i.e., reading the metadata, is the most expensive random read among the three and requires to load directly from the 3D-XPoint media (around 800 cycles latency, as discussed in Section 3.6).

Optimization. To hide the long latency of random read from the media, we devise a general approach that uses a helper thread to prefetch the needed data before the worker thread actually accesses it. The prefetcher helps load the data and its associated XPLine in the AIT buffer, the read buffer, and CPU caches. Long media access latency can be avoided when the following accesses hit these buffers. Unlike traditional hardware and software prefetching that uses access history or program analysis to predict future access patterns, the helper thread speculatively visits the directory entries, segments, and buckets for key-value pairs that have not yet been inserted to prefetch the associated XPLines. Specifically, we construct the helper thread by only retaining data loads and instructions necessary for indexing the three-level hash table, e.g., calculating the hash, from the worker thread. All stores, computation, and synchronization are removed in the helper. We currently manually modify the worker code to create the helper thread and leave automatic construction of the helper using compiler techniques for future work.

The rationale behind this design is that insertions in CCEH, like many other persistent data structures, include expensive cacheline flushes and memory barriers to ensure persistence and crash-consistency. The CPU pipeline and the memory bandwidth are both under-utilized due to the strict ordering of stores. The helper thread, which is independent of the worker thread, is not restricted by the memory barriers and able to utilize the unused bandwidth. Since the helper only contains a subset of the worker's instructions, it is faster than and always stays ahead of the worker. However, prefetching too aggressively leads to the overflow of the on-DIMM buffers and CPU caches. We empirically determined that a prefetch depth of 8 key-value pairs resulted in the best performance. To avoid using additional CPU resources for prefetching, we bound the helper thread to the sibling hyperthreads on the core where the worker runs.

Figure 10 shows the performance benefits of the speculative helper thread in CCEH on DCPMM as well as a comparison with CCEH on DRAM. Optane results on a non-interleaved single DIMM and on 6 interleaved DIMMs were similar, and thus we only present the single-DIMM case. Prefetching from the helper thread helped improve key insertion latency by up to 36% and achieved consistent latency improvement across different numbers of workers. A similar trend was observed in CCEH throughput though the improvement was not as significant by up to 34%. In contrast, as shown in Figure 10 (c) and (d), the helper threads led to no performance improvements but degradations in both latency and throughput when CCEH ran on DRAM. Note that the DRAM version of CCEH retains the persistence barriers and only differs in the underlying memory device. It suggests that the dominance of random reads in the performance of write-intensive workloads is a unique problem facing DCPMM because 1) the 3D-Xpoint media incurs a much higher latency than DRAM, and 2) write latency can be effectively hidden in the DDR-T protocol. We found that prefetching using helper threads is generally effective for workloads with large WSSes and weak or no spatial locality. Since random media reads are also expensive on **G2 Optane**, the results on CCEH are similar.

4.2 B+-Tree

B+-trees are balanced search trees that have high fanout and store records only in the leaf nodes. The high fanout allows for trees of low height and superior search performance with fewer data accesses. Since keys must be stored in internal nodes in a sorted order to exploit cacheline locality, in-place key insertions are particularly challenging in persistent B+-trees. While in-place insertion maintains sorted

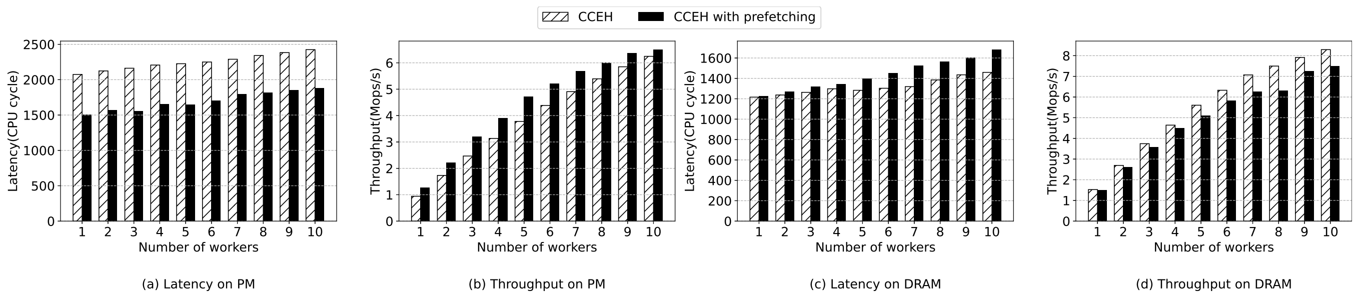

Figure 10. The effect of prefetching in CCEH on DCPMM and DRAM.

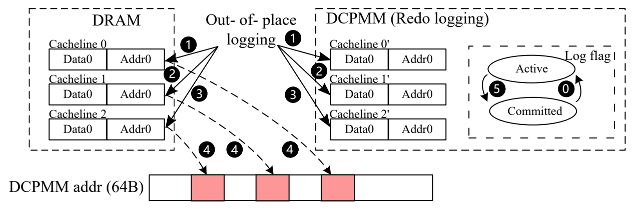

Figure 11. Out-of-place logging to update an internal node in B+-tree.

keys in internal nodes and thus preserves data locality, it requires on average half of the keys in a node to be shifted. A persistent barrier is needed after each key shift to ensure persistence and the proper ordering of stores. As a large number of keys are packed in one node, shifting keys residing on the same cacheline leads to repeated flushes and loads on the cacheline, causing long read-after-persist delays.

The FAST & FAIR algorithm [11] relaxes the requirement for cacheline flush and memory fence after each key shift and inserts persistence barriers only when the shift operation crosses cacheline boundaries. Although key shifts within a cacheline is not failure-atomic, FAST & FAIR can tolerate transient duplicate pointers due to a crash because B+-tree nodes do not allow duplicate pointers and can detect the inconsistency during recovery. However, cacheline flushes (64-byte stores) are not atomic on x86 processors, and one flush may contain multiple updates to a cacheline. Therefore, new updates flushed from volatile memory may over-write old values on PM, which will be permanently lost in a crash.
Optimization. We demonstrate the performance benefit of avoiding read-after-persist delays for in-place key insertion. The baseline is a B+-tree that performs a cacheline flush and store fence after each shift operation. We borrow the node design in FAST & FAIR and add a persistence barrier after each key shift to build the baseline. As shown in Figure 11, we employ redo logging to re-direct in-place updates to a cacheline within a FAST & FAIR node to a logging area on PM. Each log entry contains the address, the value, and the length of a single update. Separate updates are recorded in different cachelines with one entry per cacheline. For a fair comparison with the baseline, we persist each log entry immediately after it is written. Thus, the number of writes in the redo log matches that in the baseline. The difference is that updates to the cacheline are written out-of-place to the redo log, avoiding read-after-persist delays when shifting multiple keys within a cacheline. Once all updates for a cacheline are logged, and before moving to the next cacheline, the redo log for the cacheline is committed. We use atomic write to an 8-byte flag to indicate the completion of logging a cacheline. Upon a crash, committed redo logs can be used to recover lost updates. In addition to logging on PM, updates are also logged in the same format on DRAM. After logging is committed on PM, the logged updates on DRAM are written back to the original cacheline, after which the flag on the corresponding PM log is cleared, and the log can be reclaimed.

Note that our objective is not to design an efficient redo logging scheme. Instead, we seek to demonstrate that an intuitive implementation of redo logging, which is considered more expensive than in-place update due to duplicate PM writes to the log, leads to significant performance improvements due to avoiding read-after-persist on a single cacheline. We use YCSB to insert 16 million key-value pairs to the B+-tree on a single Optane DIMM and 6 interleaved DIMMs, respectively. Figure 12 (a) and (c) show the latency and throughput of insertions on a single DIMM on G1 Optane, and the results on 6 DIMMs are similar. We can observe that out-of-place write in redo logging consistently improves latency and throughput compared to in-place shift operations in the baseline by up to 38.8% and 60.8%, respectively. Since redo logging introduces additional PM writes, the performance benefits decline as the number of threads increases due to contentions on the DCPMM bandwidth.

Figure 12 (b) and (d) compare in-place and out-of-place update on **G2 Optane**. As expected, multiple shift operations on a single cacheline do not cause long RAP delays in the baseline because reads can directly load data from the caches rather than waiting for the persists to complete. Consequently, out-of-place redo logging does not offer any performance benefit. It is worth noting redo logging does not cause noticeable slowdowns either except for slight performance degradations with a large thread count.

5 Related work

In this section, we discuss the existing profiling studies of DCPMM. There are studies focusing on the performance

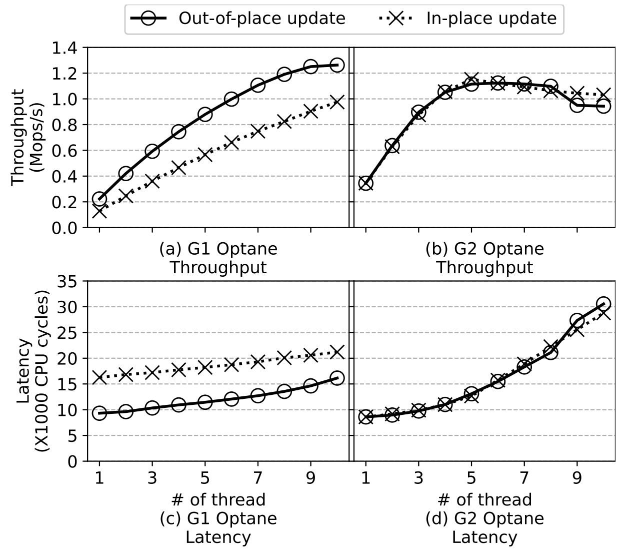

Figure 12. FAST & FAIR performance on a single DIMM.

of Optane in the memory mode [7, 9, 24]. While they provide performance characterization of the whole system, these studies cannot offer insights into the internal design of DCPMM since DRAM acts as a cache for PM in the memory mode and the caching is managed by the memory controller.

With the *app-direct* mode, a slew of studies [10, 12, 13, 16, 25, 27, 32, 34] have used both microbenchmarks and realistic applications to delve into the design of DCPMM. Discoveries include the asymmetric read/write performance, a large gap between DCPMM and DRAM performance, sensitivity in access type, pattern and size. Some have offered important insights that motivated this work. Yang et al. [32] presented the first comprehensive empirical study of DCPMM and identified the mismatch between cacheline access granularity and media access granularity and the limited concurrency as culprits for low performance in DCPMM. Wang et al. [27] developed the LENS profiling framework and offered further insights into the internal design of Optane DIMMs, including the multi-level buffer structure and their access granularities and management policies. Zhang et al. [34] used Optane-based FPGA to benchmark the DDR-T interface, the address translation path ,and probe the queue organization in DCPMM. Their results confirmed that writes under DDR-T are asynchronous. Gugnani et al. [8] focused on the idiosyncrasies of DCPMM and sought to identify common performance characteristics of Optane that may persist in the future generations of persistent memory.

This work differs in several ways and offers new insights. We have new discoveries that read-write buffering is managed separately and differently in DCPMM. We attribute the asymmetric read/write performance, sensitivity to access patterns, and the high cost of persistence barriers to the tangling of read and write in workloads. Instead, we design benchmarks that isolate the effect of read and write in order to pinpoint the bottlenecks in different types of workloads. Furthermore, we discuss the performance implications and provide programming guidelines.

There has been a large body of work focusing on evaluating and optimizing data structures designed for volatile memory on persistent memory [7, 15, 16, 23, 29]. Unlike previous designs [1, 2, 14, 20, 26, 31, 36, 37] that treated persistent memory as a slower DRAM before the release of DCPMM, these studies observed significant performance gap and irregularities between DCPMM and DRAM. To adapt persistent programs to DCPMM, studies mainly focused on improving write performance by aligning write to XPLines and merging small writes to reduce RMW operations.

FlatStore [3] combined the issued write into full XPLines as much as possible, and ChamberDB [35] aligned the buckets in the hash table to 256B. ArchTM[30] proposed a PM transaction framework that avoids small writes (smaller than 256B) and encourages sequential writes by coalescing them. Sage [5] proposed graph algorithms in which persistent memory only serves reads while write operations are issued on DRAM because of the asymmetry between reads and writes. AsymNVM [18] studied the placement of data structures in DCPMM and DRAM based on the asymmetric read and write bandwidth. LB+Trees [17] reduced the number of line writes rather than the number of written words on persistent memory because the the written words do not impact the write performance while persisting lines does. Wei et al. [28] proposed an intermediate layer to absorb access to remote NUMA nodes.

In this work, we offer a new perspective on optimizing persistent programs on DCPMM. We propose to leverage the separate read and write buffers to decouple and pipeline the execution of read and write. We present two case studies on two generations of DCPMM and demonstrate this general approach can lead to significant performance improvements.

6 Conclusion

This paper presents an in-depth study of Optane persistent memory with a focus on its on-DIMM buffering. Through controlled, carefully crafted microbenchmarks, we discover the existence of two separate and distinctly managed on-DIMM buffers for reads and writes, respectively. The discovery inspires us to treat data loads and persists differently in performance analysis and optimization. It also leads to two case studies that show how the decoupling of reads and write helps improve performance. However, we acknowledge that the criticality of the on-DIMM buffers relies on the fact that data has to be written back to DCPMM for persistence. Except for random media reads still being expensive, the read-after-persist delay does not seem to be an issue on the 2nd generation of DCPMM. It remains to be seen if the eADR feature which persists CPU caches will flourish as we have heard hesitancy from manufacturers to support this feature due to reliability concerns.

References

[1] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 553–565. https://doi.org/10.1145/3164135.3164147

[2] Shimin Chen and Qin Jin. 2015. Persistent B<sup>+</sup>-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947

[3] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[5] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *Proc. VLDB Endow.* 13, 9 (May 2020), 1598–1613. https://doi.org/10.14778/3397230.3397251

[6] Ulrich Drepper. 2007. What every programmer should know about memory. *Red Hat, Inc* 11 (2007), 2007.

[7] 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. *Proc. VLDB Endow.* 13, 8 (April 2020), 1304–1318. https://doi.org/10.14778/3389133.3389145

[8] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 626–639. https://doi.org/10.14778/3436905.3436921

[9] Mark Hildebrand, Julian T. Angeles, Jason Lowe-Power, and Venkatesh Akella. 2021. A Case Against Hardware Managed DRAM Caches for NVRAM Based Systems. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 194–204. https://doi.org/10.1109/ISPASS51385.2021.00036

[10] Takahiro Hirofuchi and Ryousei Takano. 2020. A Prompt Report on the Performance of Intel Optane DC Persistent Memory Module. *IEICE Trans. Inf. Syst.* 103-D, 5 (2020), 1168–1172. https://doi.org/10.1587/transinf.2019EDL8141

[11] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang

[12] Intel®. 2021. Intel® 64 and ia-32 architectures optimization reference manual. *URL:https://software.intel.com/content/dam/develop/external/us/en/documents-tps/64-ia-32-architectures-optimization-manual.pdf* (2021).

[13] 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. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[14] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 257–270. https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon

[15] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[16] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (2019), 574–587. https://doi.org/10.14778/3372716.3372728

[17] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1078–1090. https://doi.org/10.14778/3384345.3384355

[18] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 757–773. https://doi.org/10.1145/3373376.3378511

[19] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

[20] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[21] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 288–303. https://doi.org/10.1145/3357526.3357541

[22] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 265–276. https://doi.org/10.1109/ISCA.2014.6853222

[23] Ivy Peng, Kai Wu, Jie Ren, Dong Li, and Maya Gokhale. 2020. Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 916–925. https://doi.org/10.1109/IPDPS47924.2020.00098

[24] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System Evaluation of the Intel Optane Byte-Addressable NVM. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 304–315. https://doi.org/10.1145/3357526.3357568

[25] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, Thomas Neumann and Ken Salem (Eds.). ACM, 12:1–12:7. https://doi.org/10.1145/3329785.3329930

[26] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th*

USENIX Conference on File and Stroage Technologies (San Jose, California) *(FAST'11)*. USENIX Association, USA, 5.

[27] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 496–508. https://doi.org/10.1109/MICRO50266.2020.00049

[28] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 523–536. https://www.usenix.org/conference/atc21/presentation/wei

[29] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 76, 19 pages. https://doi.org/10.1145/3295500.3356159

[30] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. 2021. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 141–153. https://www.usenix.org/conference/fast21/presentation/wu-kai

[31] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[32] 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 *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[33] Vinson Young, Zeshan A. Chishti, and Moinuddin K. Qureshi. 2019. TicToc: Enabling Bandwidth-Efficient DRAM Caching for Both Hits and Misses in Hybrid Memory Systems. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. 341–349. https://doi.org/10.1109/ICCD46524.2019.00055

[34] Jialiang Zhang, Nicholas Beckwith, and Jing Jane Li. 2021. GORDON: Benchmarking Optane DC Persistent Memory Modules on FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 97–105. https://doi.org/10.1109/FCCM51124.2021.00019

[35] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. *ChameleonDB: A Key-Value Store for Optane Persistent Memory*. Association for Computing Machinery, New York, NY, USA, 194–209. https://doi.org/10.1145/3447786.3456237

[36] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 421–434. https://doi.org/10.14778/3372716.3372717

[37] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo