Characterizing the Performance of Intel Optane Persistent Memory

– A Close Look at its On-DIMM Buffering

Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, Hong Jiang
The University of Texas at Arlington
Arlington, Texas, USA

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 Optane'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 read and write are managed differently in separate on-DIMM read and write buffers. 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. Furthermore, we discover that the mismatch between cacheline access granularity and the 3D-Xpoint media access granularity negatively impacts the effectiveness of CPU cache prefetching and leads to wasted persistent memory bandwidth.

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

CCS Concepts: • **Hardware → Memory and dense storage**.

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3519556

Keywords: Persistent memory, Optane DCPMM, performance characterization

ACM Reference Format:
Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: – A Close Look at its On-DIMM Buffering. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3492321.3519556

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, Optane 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, 23, 26, 30, 32, 35, 36] have found that Optane DCPMM should not simply be treated as slower, persistent DRAM. Compared to DRAM, Optane DCPMM exhibits complicated behaviors and drastically changing performance depending on access size, access type and pattern.

Since documentation on Optane DCPMM's internal architecture is not yet publicly available, understanding its performance is important to building high-performance and efficient persistent applications. Optane 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 Optane 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, Optane 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 Optane 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 Optane'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 Optane DCPMM with microbenchmarks and have the following findings that were not previously reported.

- Optane 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, adjacent XPLine prefetching into the read buffer due to CPU cache prefetching activities leads to improved sequential read performance but may lead to wasted memory bandwidth for random access patterns. 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 Optane 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 Optane 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 Optane DCPMM without platform support for cache persistence.

We present three optimizations inspired by the insights. 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 fact that 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.

The mismatch between cacheline access granularity and 3D-Xpoint media access granularity incurs a higher CPU prefetching cost in DCPMM compared to that in prefetching from DRAM. A misprediction on memory address pollutes one cacheline in the CPU cache in both DRAM and DCPMM but additionally requires an entire XPLine to be prefetched into the read buffer. Moreover, existing studies [19, 21] have suggested that data accesses in persistent programs should be made sequential and XPLine-aligned to attain a high bandwidth and reduce write amplification. This design almost certainly triggers CPU prefetching due to sequential accesses within an XPLine but results in misprefetching across XPLines if cross-XPLine accesses are not essentially sequential. We demonstrate in a third case study that accessing cachelines using the Advanced Vector Extensions (AVX) in the x86 instruction set effectively avoids misprefetching and reduces the amount of data loaded from the physical media.

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 Optane 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, Optane DCPMM provides both byte-addressability and persistence. Sitting on the memory bus, Optane 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, Optane DCPMM employs an on-DIMM write-combining buffer to merge adjacent small writes. Recent studies [30, 35] on Optane DCPMM 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 Optane 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 (**G2**) Optane DCPMM, the extended ADR (eADR) domain includes CPU caches.

To enforce crash consistency, stores to Optane DCPMM must be properly ordered. Since evictions from the CPU cache hierarchy may not follow store order, in the 1st generation (**G1**) Optane DCPMM, to persist a data object, store instructions are followed by cacheline flushes[1] (e.g., `clflush`, `clflushopt`, or `clwb`) and a memory barrier (e.g., `sfence` or `mfence`). The barrier ensures that cacheline 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 cacheline flush and the following fence instruction together are usually referred as a *persistence barrier*. In G2 Optane DCPMM, 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 G2 Optane DCPMM testbed we evaluated is equipped with the 200 series Optane DIMMs and the latest Intel Xeon scalable processors but with eADR disabled.

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

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 Optane 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 Optane DCPMM in order to infer its internal design. Notably, three architectural characteristics of Optane 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 and prefetching, 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 a 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 the physical media 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 Optane 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 upper bounded by 4 (i.e., $\frac{256B}{64B}$).

Platform. We have two test beds equipped with two generations of Optane 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) Optane DCPMM had dual Intel Xeon Gold 6320 2.1GHz CPUs (20 cores) with 32KB L1i/L1d cache, 1MB L2 cache, and 27.5MB L3 cache, while the 2nd generation (G2) Optane DCPMM server had dual Intel Xeon Gold 5317 3GHz CPUs (12 cores) with 1.1MB L1d/768KB L1i cache, 30MB L2 cache, and 36MB L3 cache. Both were equipped with 192GB DRAM and six 128GB Optane DCPMMs. All the DRAM and Optane DIMMs were installed in one CPU socket to isolate Optane performance 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 a program's address space. We used `ipmwatch` from the Intel VTune profiler to measure data accesses in the 3D-Xpoint media and those 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 [30, 35] 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 [35], 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 along with a RMW buffer (Section 3.2) in Optane DIMMs, and they are separated from each other (Section 3.6). 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 measures 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 cachelines are read in an XPLine. As shown in Figure 2, with a small WSS, RA is inversely proportional to CpX, indicating that loading multiple cacheline in an XPLine only requires 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 is 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 Optane DCPMM** has similar results on read amplification but a slightly larger 22 KB read buffer.

Performance implications. In Figure 2, RA jumps to 4 immediately after 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 Section 3.4, the read buffer is essential for reducing the cost of read-modify-write operations and improving sequential access performance.

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

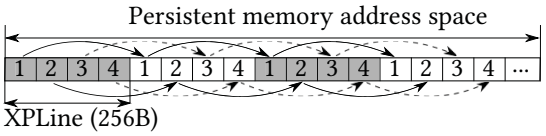

Figure 1. Inferring read buffer capacity with strided read.

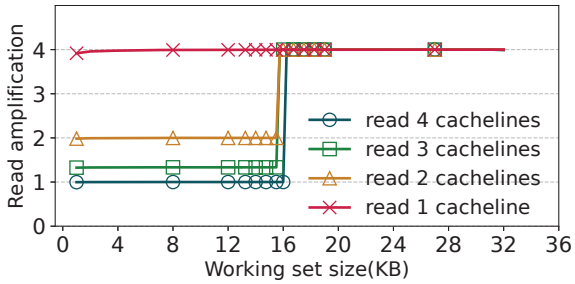

Figure 2. Read amplification due to various access patterns in a single Optane DIMM.

3.2 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. Note that the benchmark bypasses the CPU cache and performs pure writes directly to Optane DCPMM.

Findings. Figure 3 shows write amplification due to different types of write in **G1 Optane DCPMM**. Note that the WA curves as well as write performance (i.e., latency and throughput) are independent of the access pattern across XPlines (i.e., sequential or random) and only depend on the WSS. The figure suggests that the write buffer manages full and partial writes differently in G1 Optane. 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 write buffer is between 12KB and 16KB, and there are two write-back mechanisms in G1 Optane DCPMM. 1) Fully modified XPLines are written back to the media periodically, while 2) partially modified XPLines are retained in the buffer until evicted. We

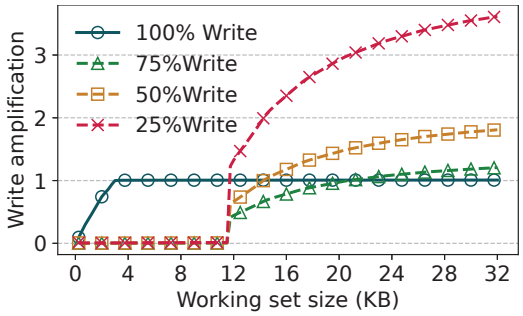

Figure 3. Write amplification in G1 Optane DCPMM.

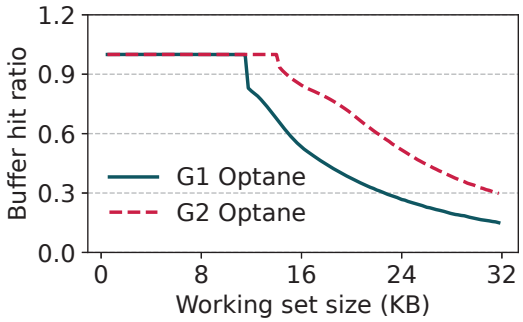

Figure 4. Write buffer hit ratio as WSS increases.

empirically determined that fully-written XPLines are periodically written back to the media approximately every 5,000 cycles. However, in **G2 Optane DCPMM**, periodic XPLine writeback is disabled for full writes.

To study the eviction algorithm used in the write buffer, we issue random partial writes and measure the amount of media write relative to program issued writes to infer the ratio of writes that hit the buffer. Figure 4 shows that 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 write buffer likely employs 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 Optane DCPMM. Ideally, write WSS should not exceed 16 KB to maximize write buffer hit and eliminate media writes, which is quite challenging if not impossible in realistic data stores. Practically, 1) programmers 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) since access sequentiality does not affect write performance or write buffer management, programmers do not need to particularly pursue sequentiality and avoid random access for writes.

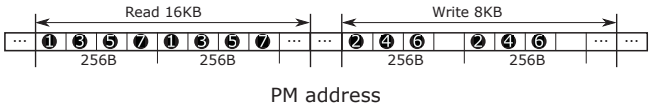

Figure 5. Interleaved read and write operation to check the relation between read and write buffer.

3.3 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 separate, each having a dedicated buffer space. XPLines can be moved between the two, likely via a tagging mechanism.
Benchmark. To study if reads and writes compete for shared buffer space, we set up two non-overlapping regions on Optane DCPMM: a 16KB read and an 8KB write region. As shown in Figure 5, 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 Optane DCPMM and bypass the CPU cache, cachelines are flushed/invalidated immediately after read from Optane, and writes are non-temporal stores.
Findings. We compare the amount of data read from and written to the iMC and the media, respectively, 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 shared 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 separate.

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 DCPMM, thereby likely transitioned to become part of the write buffer. The buffers in **G2 Optane DCPMM** behave similarly.
Performance implications. Hitting the read or write buffer avoids expensive media read or 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 allowing pipelining to hide media access latency. Specifically, a 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.

3.4 Data prefetching to the on-DIMM buffers

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 there exists on-DIMM prefetching and how the on-DIMM buffering interacts with cache prefetching.
Benchmark. The trigger for CPU cache prefetching is usually a sequence of two or more cache misses in a certain pattern or read requests that exhibit sequential or spatial locality [6]. To study the effect of prefetching, we devise a single-threaded benchmark that randomly accessed the Optane DIMM with a granularity of 256B (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 all the cachelines 16 times, thereby invoking data prefetching at various levels. 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 amount of additional data loaded due to prefetching to the on-DIMM buffers. Similarly, We also measure the read ratio for the CPU cache, i.e., the ratio of iMC loaded data over program demanded data. Since accesses across blocks (i.e., XPLines) are random, data prefetched beyond the boundary of an access block does not result in a buffer hit and is thus regarded as wasted.
Findings. Our testbeds allow three CPU cache prefetchers in the Intel scalable processors to be individually enabled/disabled via BIOS configurations. We first disable all three CPU prefetchers to study if there is an on-DIMM prefetching mechanism separate from CPU prefetching. Figure 6 (a) and (e) show that the read ratios for Optane DCPMM and the iMC are both close to 1 in both G1 and G2 Optane DCPMM, suggesting that no noticeable on-DIMM prefetching activities are observed. In contrast, when individual CPU prefetchers are enabled respectively, the read ratio of DCPMM deviates from that of the iMC. There are three regions in each figure according to the WSS.

- *WSS smaller than read buffer (16KB).* The WSS fits entirely in the read buffer, and all data prefetched into the read

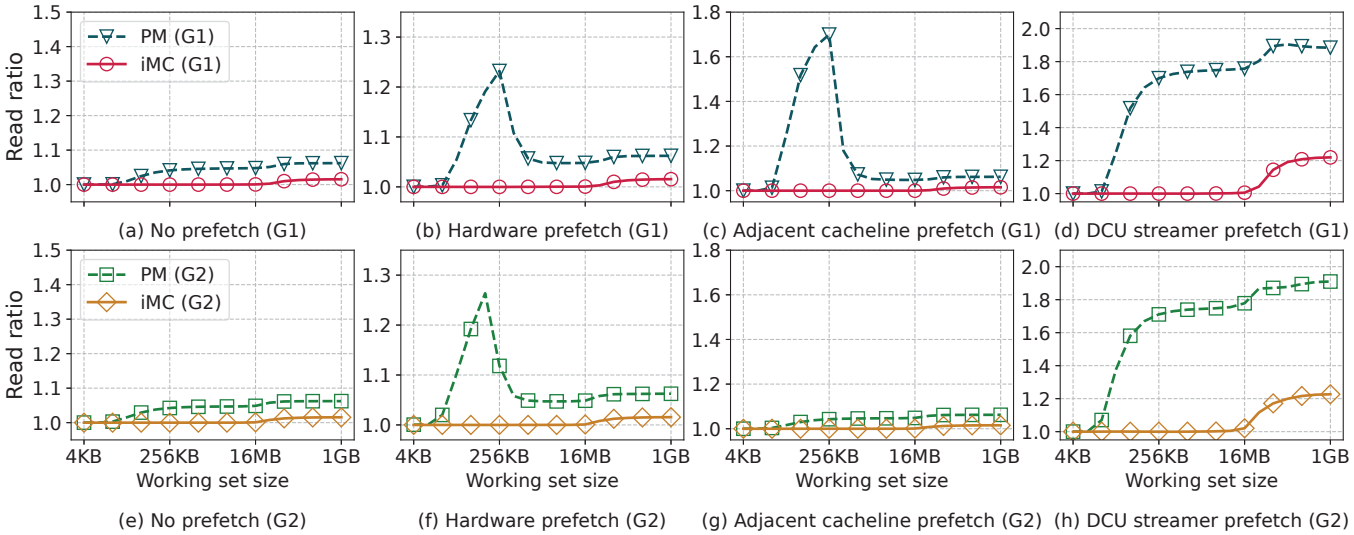

Figure 6. Data prefetching in G1 and G2 Optane DCPMM.

buffer by the CPU prefetcher results in buffer hits in the following accesses. Thus, no additional data other than program demanded data is loaded from the media.

- *WSS larger than read buffer but smaller than L3 cache.* The working set can no longer fit in the read buffer but still fits in the last-level cache (LLC) (27.5MB on the G1 and 36MB on the G2 server). While the read ratio for iMC remains 1 as all CPU prefetched data leads to LLC hits, the read ratio for Optane DCPMM increases significantly. Since the WSS is larger than the read buffer size, prefetched data is evicted before it can be hit in the buffer, resulting in wasted and repetitive loads from the media.
- *WSS larger than L3 cache.* LLC misses due to the large WSS invoke frequent CPU prefetching and cause the read ratios in both Optane DCPMM and the iMC to grow. One notable observation is that DCPMM read ratio is significantly higher than that of the iMC. With an access block of 256B (i.e., 4 cachelines), the iMC at most misprefetches one additional cacheline at the boundary of the access block while DCPMM loads an entire XPLine from the media.

Performance implications. While prefetching is important to the performance of sequential access, it could be detrimental to random access due to the cost of misprefetching. As we demonstrate that the prefetching activity in Optane DIMMs is determined by CPU prefetching, the misprefetching penalty is particularly high in DCPMM than that in DRAM. The mismatch between the cacheline granularity in CPU prefetching and the media access granularity requires 4 cachelines or an XPLine to be loaded from the media upon a misprediction in prefetching. For workloads that are optimized for XPLine-sized and aligned data access, CPU prefetching could account for half of the DCPMM bandwidth. Programmers are advised to carefully weigh the benefit and cost of prefetching in such workloads.

Algorithm 1: Measure read-after-persist latency

input: char *addr (PM 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 cacheline */
10 end
```

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 DCPMM'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, and cross socket access may make it even worse. 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.

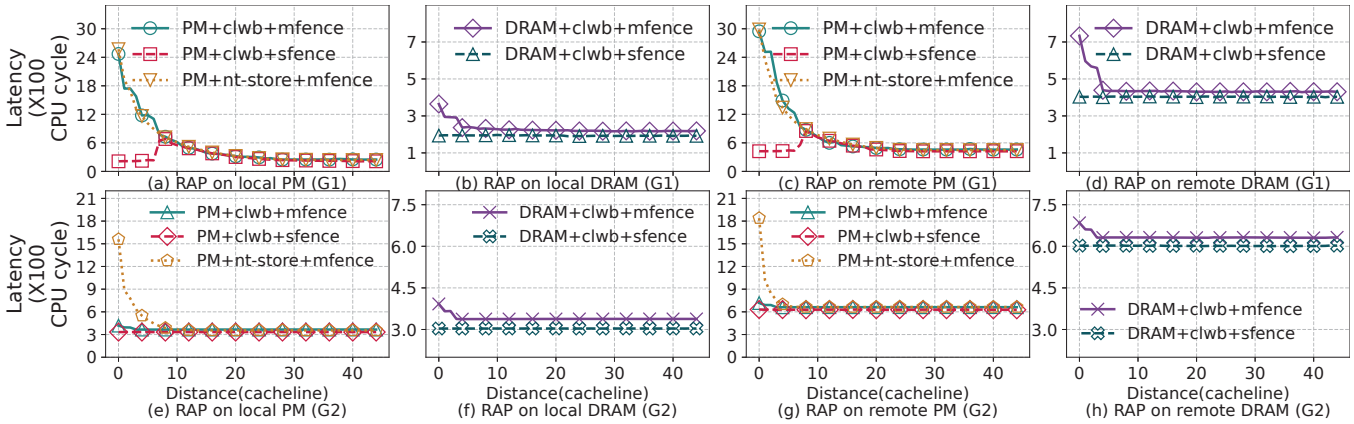

Figure 7. Per-iteration latency in Algorithm 1 as the RAP distance increases. The distance is in the number of cachelines.

Benchmark. Algorithm 1 shows how we quantify the RAP latency with respect to RAP distance. The benchmark accesses data on Optane 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. We place the benchmark program and the working set in a single memory node and two memory nodes, respectively, to study the impact of the non-uniform memory access (NUMA) factor.

Findings. Figure 7 plots the average CPU cycles spent on one iteration of the benchmark as the RAP distance increases in **G1** and **G2 Optane DCPMM**, respectively. We investigate `clwb` and `nt-store` with different types of memory fences in both Optane DCPMM and DRAM. In **G1 Optane**, as shown in Figure 7 (a) and (c), `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 still being persisted, the average per-iteration latency reaches up to 2,500 and 3,200 cycles when the benchmark accesses data from local and remote Optane memory, respectively. As the RAP distance increases, the latency quickly decreases and eventually approaches that of the on-DIMM buffers. The latency gap on Optane 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 800/1,000 cycles on local/remote memory node, 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 DCPMM, `clwb` invalidates the flushed cacheline. Therefore, as the 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.

As shown in Figure 7 (e) - (h), the long RAP delay for `nt-store` still persists while the curves of RAP latency for `clwb` approach those on DRAM on G2 Optane DCPMM. The major difference between G1 and **G2 Optane DCPMM** 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. However, the new `clwb` implementation in the latest Intel processors does not come with no cost. There is a significant increase in the latency of hitting the on-DIMM buffers in DCPMM as well as that of loading data from DRAM. Retaining flushed cachelines in the cache avoids future cache misses but increases the cost of maintaining coherence on the cachelines [12].

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 DCPMM, `nt-store` suffer from it in both generations of Optane DCPMM. 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. Even if there is no contention across sockets and cachelines can be kept in CPU caches, persisting data via remote CPUs should be avoided due to the high overhead to maintain cache coherence.

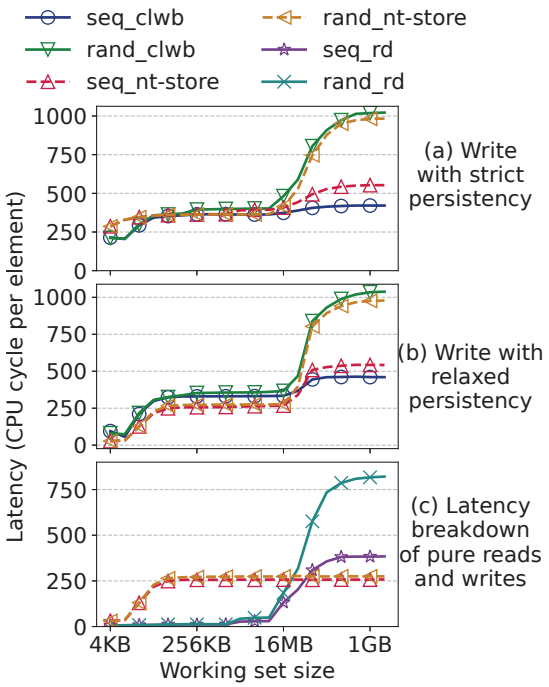

Figure 8. User-perceived latency with various working set sizes in G1 DCPMM. Results are similar on G2 DCPMM.

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 Optane 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 cacheline 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 cacheline. 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 [24], 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 then 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 cacheline. 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 [30] 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.

In contrast, write latency is consistent across different WSSes except for small WSSes that fit in the write buffer. Because 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 DCPMM**, 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 DCPMM 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 write 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 three cases studies of applying the aforementioned insights to three representative workloads.

4.1 CCEH

Cacheline-Conscious Extendible Hashing (CCEH) is an extendible hash table designed for persistent memory [21]. 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.

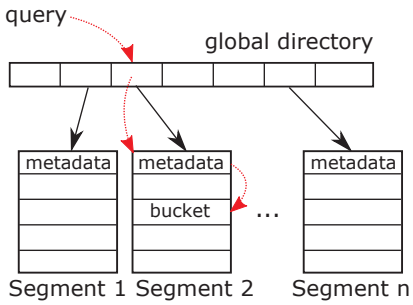

Figure 9. Workflow of accessing a bucket in CCEH.

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%

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 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).

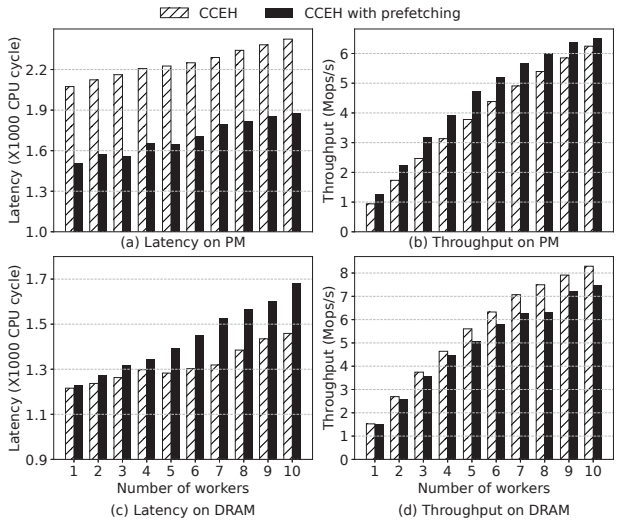

Figure 10. The effect of prefetching in CCEH on Optane DCPMM and DRAM.

Optimization. As discussed in Section 3.6, random reads could be a dominating bottleneck even in write-intensive workloads. 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 Optane DCPMM as well as a comparison with CCEH on DRAM. Optane DCPMM 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 Optane 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 DCPMM**, the results on CCEH are similar. Note that the optimization simultaneously improves latency and throughput but at the cost of consuming more CPU cycles. As long as the prefetcher and the worker does not saturate the read or write bandwidth, both latency and throughput would benefit from the optimization.

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 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.

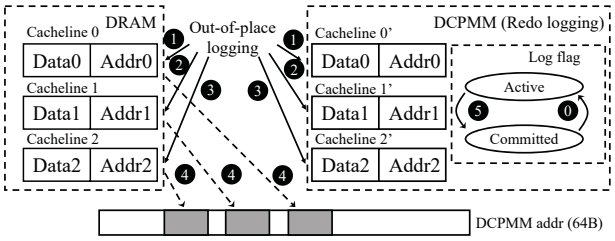

Figure 11. Out-of-place logging to update an internal node in B+-tree.

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

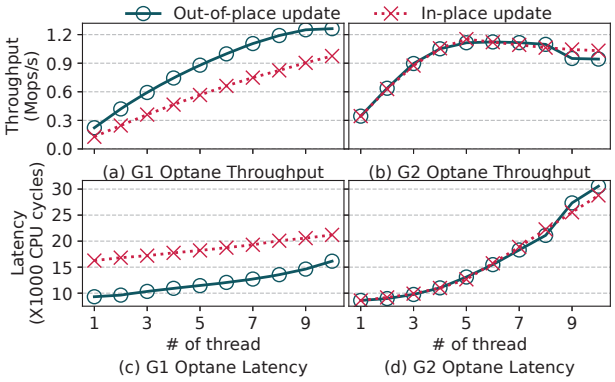

Figure 12. FAST & FAIR performance on a single DIMM.

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 DCPMM, 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 Optane bandwidth.

Figure 12 (b) and (d) compare in-place and out-of-place update on **G2 Optane DCPMM**. 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.

4.3 XPLine-aligned workloads

Due to the mismatch between cacheline access granularity and media access granularity in Optane DCPMM, small writes become XPLine-sized (256B) stores in PM, causing write amplification. Recent studies [19, 21] propose to merge small writes into large, 256B stores and build data structures on 256B, XPLine-aligned data blocks. As demonstrated in Section 3.4, there also exists a mismatch between the *prefetch degree* in DRAM and that in DCPMM – one cacheline is prefetched at each time prefetching is triggered in DRAM while an entire XPLine (4 cachelines) has to be loaded from the media at each trigger. This results in a 4X misprediction penalty. For XPLine-aligned workloads, especially for those using 256B (one XPLine) data blocks, if there is not sufficient sequentiality across blocks, misprefetching could consume up to half of the PM bandwidth.

Optimization. As CPU prefetchers are designed and optimized for DRAM and cacheline access granularity, it is challenging to reduce misprefetching penalty in PM without changing the prefetching hardware. We demonstrate via a simple optimization that an additional data copy from PM to DRAM can effectively avoid the misprefetching penalty in DCPMM. Algorithm 2 shows how sequentially accessing an XPLine in DCPMM can be transformed to reduce misprefetching penalty. Instead of loading an XPLine using an ordinary `load` instruction, the optimization copies the XPLine to 4 cacheline-sized DRAM buffers via streaming single instruction multiple data (SIMD) instructions (e.g., AVX or SSE), from where CPU performs reads/writes on the XPLine. This optimization effectively disables prefetching for XPLines of interest without affecting other PM addresses or DRAM accesses. Note that Algorithm 2 can be extended to enforce crash-consistency using undo or redo logging.

Algorithm 2: Reduce misprefetching per XPLine

```
   input: char *addr (XPLine address)
 1 char buffer[64];
 2 int cacheline = 0, offset = 0;
 3 while cacheline < 4 do
 4 |   512bit_mov [addr + cacheline * 64], [buffer];
 5 |   while offset < 64 do
 6 |   |   load [buffer + offset % 64];
 7 |   |   offset += 1;
 8 |   end
 9 |   cacheline ++;
10 end
```

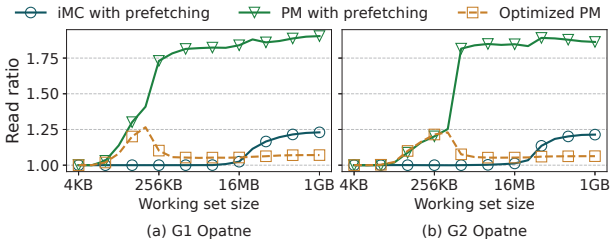

Figure 13. Reducing misprefetching in DCPMM.

We applied this optimization to the benchmark used in Section 3.4 and redirected accesses to each 256B block to a DRAM buffer. Figure 13 shows the read ratios, i.e., the amount of data actually loaded relative to program demanded data through the iMC and from the 3D-Xpoint media. The access redirection optimization effectively reduced the amount of data read and prefetched from the media. However, as shown in 14, the extra data copy from an XPLine in DCPMM to the DRAM buffer indeed incurred significant performance overhead at a small thread count. As the number of threads increased, memory bandwidth contention due to misprefetching outweighed individual threads' performance. Redirecting

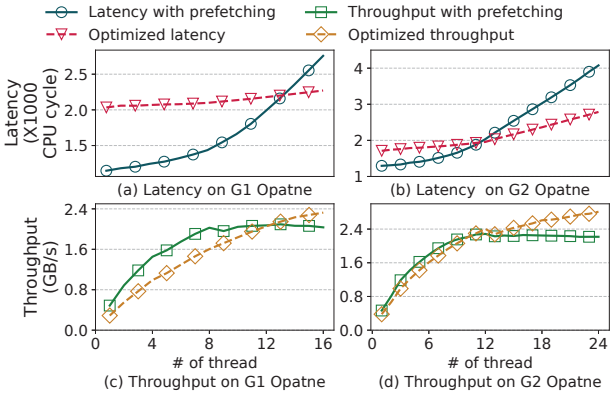

Figure 14. Performance tradeoff due to access redirection.

PM accesses to DRAM buffers outperformed direct PM access in both latency and throughout with 12 or more threads.

5 Related work

In this section, we discuss the existing profiling studies of Optane DCPMM. There have been studies focusing on the performance of Optane DCPMM in the memory mode [7, 9, 26]. While they provide performance characterizations of the whole system, these studies do not offer insights into the internal design of Optane DCPMM.

With the *app-direct* mode, a slew of studies [10, 12, 13, 17, 27, 30, 35, 37] have used both microbenchmarks and realistic applications to delve into the design of Optane DCPMM. Discoveries include the asymmetric read/write performance, a large gap between Optane DCPMM and DRAM performance, sensitivity in access type, pattern and size. Some have offered important insights that motivated this work. Yang et al. [35] presented the first comprehensive empirical study of Optane DCPMM and identified the mismatch between cacheline access granularity and media access granularity and the limited concurrency as culprits for low performance in Optane DCPMM. Kim et al. [14] confirmed the excessive metadata write due to the cache coherence protocol. Beyond these discoveries, Wang et al. [30] 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. [37] used Optane DCPMM-based FPGA to benchmark the DDR-T interface, the address translation path and probe the queue organization in Optane DCPMM. Their results confirmed that writes under DDR-T are asynchronous. Gugnani et al. [8] focused on the idiosyncrasies of Optane DCPMM and sought to identify common performance characteristics of Optane DCPMM 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 Optane 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.

Prior to the release of Optane DCPMM, previous studies [1, 2, 15, 22, 28, 34, 39, 40] treated PM as a slower DRAM. To emulate the higher access latency, lower bandwidth, and asymmetric read/write performance in PM, various techniques were employed, including adding additional latency to read/write accesses via software [29, 34, 40], by BIOS configurations [22], or throttling DRAM bandwidth [15]. Crash-consistent data structures were proposed by enforcing cacheline-level failure-atomicity [1].Since the arrival of DCPMM, there has been a large body of work focusing on evaluating and optimizing data structures designed for volatile memory on persistent memory [7, 16, 17, 25, 32], and observed significant performance gaps and irregularities between Optane DCPMM and DRAM. To adapt persistent programs to Optane 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 ChameleonDB [38] aligned the buckets in the hash table to 256B. ArchTM [33] 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 [20] studied the placement of data structures in Optane DCPMM and DRAM based on the asymmetric read and write bandwidth. LB+Trees [18] reduced the number of line writes rather than the number of written words on persistent memory because the written words do not impact the write performance while persisting lines do. Wei et al. [31] proposed an intermediate layer to absorb access to remote NUMA nodes.

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

6 Discussions

While the existing work has provided valuable suggestions on persistent programming, including coalescing smaller writes and matching program data access granularity with media access granularity to reduce write amplification, the main takeaway in this paper is to separate read and write in performance analysis and optimization. There is more concurrency and bandwidth to reads but a lack of a mechanism to hide read latency. In comparison, writes have consistent latency as they are asynchronous but suffer low bandwidth and concurrency. Given a specific workload, it is important to determine whether read or write is the bottleneck.

The main obstacle to adopting byte-addressable persistent memory is the need for programmers to ensure persistence and crash-consistency, i.e., performing cacheline flushes and properly ordering them. Not only do persistence barriers make persistent programming error-prone, but they also constitute a major performance bottleneck. Strictly ordered load and store instructions limit the memory bandwidth can be attained via a single thread. As demonstrated in the CCEH study, there is ample memory bandwidth available to a single core to perform data prefetching. This discovery suggests that an increased degree of hardware-level parallelism in processor design, i.e., more hyperthreads per core, may be needed to fully exploit memory bandwidth in DCPMM.

The extended ADR (eADR) in the G2 Optane DCPMM is a critical architectural support for persistent programming. However, eADR requires significantly higher stored energy (usually backed by batteries) than that in ADR to keep the CPUs running in order to execute BIOS code that flushes the CPU caches. It also requires a sophisticated power system in the host machine to trigger eADR upon a power loss. At the time of writing, there is only one platform that supports eADR to Intel's specification and a few other platforms for testing only.

7 Conclusions

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 three case studies that show how the decoupling of reads and writes 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 Optane 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 Optane DCPMM. It remains to be seen if the eADR feature will flourish as a sophisticated power system and changes in the platform design are needed for flushing CPU caches upon a power loss.

8 Acknowledgments

We thank our shepherd, Naama Ben-David, and the anonymous reviewers for their insightful comments. This work was supported by NSF under award CCF-1845706.

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⁺-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] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 424–439. https://doi.org/10.1145/3477132.3483589

[15] 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

[16] 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

[17] 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

[18] 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

[19] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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.

[29] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada) *(Middleware '15)*. Association for Computing Machinery, New York, NY, USA, 37–49. https://doi.org/10.1145/2814576.2814806

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

A Artifact Appendix

A.1 Abstract

The artifact provides all the source code to reproduce the graphs in the paper and could run on both G1 Optane DCPMM and G2 Optane DCPMM. The operations are wrapped in a python script "run.py" to simplify the artifact evaluation. The script will automatically determine the path of executable files, collect the results of the experiments and generate the graphs in the paper. More details are included in the file "README.md".

A.2 Description & Requirements

A.2.1 How to access.

- **Artifact link**: https://github.com/lingfenghsiang/Persistent-Memory-Study.
- **Artifact license**: GNU GPL V3.0.
- **Archived DOI**: 10.5281/zenodo.6342303
- **Archived version**: v0.0

A.2.2 Dependencies. Both hardware and software dependencies are included in the Section "Prerequisites" of "README.md".

A.2.3 Benchmarks. YCSB is the only required third-party benchmark in the case studies, and its execution is automated in the "run.py" script.

A.3 Setup

To set up the machine, please follow instructions from the Section "Usage"/"Before run" of file "README.md".

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): Persistent memory DIMM has a read buffer that evicts an XPLine once another is loaded into the CPU cache. This is proven by experiment E1 described in Section 3.1 whose results are illustrated in Figure 2.
- (C2): Optane DCPMMs prefetch data, but it is negligible compared to that caused by CPU prefetcher. This is proved by experiment E2 described in Section 3.4 whose results are reported in Figure 6
- (C3): Persistent memory DIMM has a write buffer that flushes back fully modified XPLines periodically and retains partially modified XPLines until evicted on G1 Optane DCPMM. This is proved by experiment E3 described in Section 3.2 whose results are illustrated in Figure 3.
- (C4): The G2 Optane DCPMM has a different write buffer size and eviction algorithm from the first generation. This is proved by experiment E4 described in Section 3.2 whose results are illustrated in Figure 4.
- (C5): Accessing recently persisted causes significantly high latency on G1 Optane DCPMM, but G2 Optane DCPMM fixes this issue. Further, reaching cache coherence raises high latency overhead. It's proven by experiment E5 described in Section 3.5 whose results are illustrated in Figure 7.
- (C6): With smaller WSS than the write buffer size, write operations for relaxed model have lower latency than strict persistency model, but their performance converges and plateaus as WSS grows. If the WSS overwhelms LLC, the latency of load instructions overshadows the store instructions. This is proven by experiment E6 in Section 3.6 whose results are presented in Figure 8.
- (C7): With helper threads, CCEH achieves 36% lower latency and 34% higher throughput for insert operations on persistent memory while not improving performance on DRAM. This is proven by experiment E7 described in Section 4.1 whose results are illustrated in Figure 10.
- (C8): By employing a redo log, FAST & FAIR achieves 38.8% lower latency and 60.8% throughput on G1 Optane DCPMM but no performance improvement on G2 Optane DCPMM. This is proven by experiment E8 in Section 4.2 whose results are illustrated in Figure 12.
- (C9): Loading data to DRAM via SIMD instructions reduces wasted prefetched data when randomly accessing XPlines and improves scalability for read operations on XPline-aligned workloads. This is proven by experiment E9 described in Section 4.3 whose results are illustrated in Figure 13 and Figure 14.

A.4.2 Experiments. The execution of the code is simplified into four steps. To reproduce the results from this paper, please follow the steps in Section "Usage"/"Reproduce results from the paper" of "README.md". Once all the four steps are finished, the graphs presented in this paper are generated, and the location of corresponding graphs can be found in Section "Matching Paper Results".

Experiment (E1): Read buffer test
Expected outcome. Figure "read_amp.png" plots four lines, among which "read 4 cachelines" is a flat line and the other three have a step at a particular working set size, and the four all end up at read amplification of 4.0. The working set size corresponding to the step may vary on G1 Optane DCPMM and G2 Optane DCPMM.

Experiment (E2): Prefetching test
Expected outcome. As the WSS exceeds the write buffer size, the DIMM reads up to X1.06 data of the desired, while excessively prefetched data by CPU may reach X2.

Experiment (E3): Write buffer amplification test
Expected outcome. The figure "write_buf.png" plots four lines. On G1 Optane DCPMM, 100% write line goes up to 1 and then becomes a horizontal line, and the other three lines, 25%/50%/75%, will bump up near 12KB working set size and gradually approach 4/2/1.33. On G2 Optane DCPMM, write amplification of the four lines will gracefully increase once

the working set size is above a threshold larger than 12KB.

Experiment (E4): Write buffer hit ratio test
Expected outcome. The write buffer hit ratio suddenly drops at 12KB WSS on G1 Optane DCPMM while that on G2 Optane DCPMM declines gracefully, and the turning point exceeds 12KB.

Experiment (E5): Read after persist test
Expected outcome. Reading recently persisted cacheline causes high latency, and "sfence" allows low latency for instantly reading recent one or two cachelines compared to "mfence" on G1 Optane DCPMM. In contrast, only the latency of reading nt-stored cachelines stands out on G2 Optane DCPMM. The overall latency for remote NUMA nodes is higher than that on local NUMA nodes on G1 and G2 servers.

Experiment (E6): Latency test
Expected outcome. Among the three sub-figures, the major difference between the first two figures is that relaxed model has much lower latency than strict persistency. The last figure indicates significantly high read latency and consistent write latency for large working set sizes.

Experiment (E7): CCEH case study
Expected outcome. Helper threads improve the latency and throughput by almost 35% at maximum. The improvement may fade away faster with fewer DIMMs upon multi-threaded insert operations.

Experiment (E8): B+-tree case study
Expected outcome. The throughput is improved by 60%, and the latency is decreased by 40% on G1 Optane DCPMM approximately, but there is no significant difference in performance on G2 Optane DCPMM.

Experiment (E9): Prefetching case study
Expected outcome. With default prefetching, the Optane DCPMM may read up to X2 data of the desired. The optimized case may decrease the read size back to almost X1 and shows better throughput along with latency when the thread number exceeds almost 12 for both G1 and G2 server.