Arbiter: Alleviating Concurrent Write Amplification in Persistent Memory

Bolun Zhu, Yu Hua
Huazhong University of Science and Technology
Corresponding author: Yu Hua (csyhua@hust.edu.cn)

***Abstract*—Persistent memory (PM) is able to bridge the gap between the high performance and persistence, thus receiving many research attentions. The concurrency in PM is often constrained due to limited concurrent I/O bandwidth. The I/O requests from different threads are serialized and interleaved in the memory controller. Such concurrent interleaving unintentionally hurts the locality of PM's on-DIMM buffer (XPBuffer) and thus causes significant performance degradation. Existing systems either endure performance degradation caused by the concurrent interleaving or leverage dedicated background threads to asynchronously perform I/O to PM. Unlike conventional designs, we present a non-blocking synchronous I/O scheduling mechanism that can achieve high performance and low I/O amplification. The key insight is that inserting a proper number of delays to I/O can mitigate the I/O amplification and improve the effective bandwidth. We periodically assess the system states and adaptively determine the number of delays to be inserted for each thread. Evaluation results show that our design can significantly alleviate the I/O amplification and improve system performance for concurrent applications.**

I. Introduction

The emerging memory expansion technologies, such as Persistent Memory (PM) [1] and CXL [2], have garnered increasing attentions due to the limitations of existing memory systems, which are challenging to scale in terms of low capacity limits, high costs, and restrictions on the number of hardware slots/pin counts. PM, in particular, is considered to construct memory systems due to salient features of large capacity, near-DRAM performance, byte-addressability, and non-volatility [1]. It can be utilized to accelerate applications like in-memory databases and key-value stores [3], [6], [7]. However, despite PM promising performance comparable to DRAM, practical applications based on PM are difficult to match the performance of DRAM, e.g., significantly degraded performance in indexing. Apart from PM inherently achieving lower performance than DRAM, an important contributing factor is the read-write amplification of PM.

The read-write amplification of PM comes from the granularity mismatch between PM and the CPU cache (i.e., 256B vs. 64B). To mitigate the read-write amplification, PM incorporates a small cache internally to merge adjacent writes. Ideally, adjacent sequential writes can be cached and merged. However, when multiple cores access PM concurrently, adjacent sequential writes from the same core are interleaved with requests from other cores, thereby disrupting locality. Consequently, the cache hit rate of PM's internal cache significantly decreases under high concurrency, which further exacerbates write amplification and consequently decreases system performance. The write amplification caused by concurrency is referred to as concurrent write amplification. As a result, it is widely recognized that PM exhibits limited concurrency performance and exacerbates the concurrency of PM applications. Table I illustrates that PM applications are significantly impacted by the concurrent write amplification: when scaling from 4 to 16 threads, most PM applications experience a higher write amplification, which indicates a lower effective bandwidth.

TABLE I: Write Amplification (WA) of different PM indexes. CWA refers to concurrent write amplification, see II-C.

Indexes	**WA(4 Threads)**	**WA(16 Threads)**	**CWA**
FAST&FAIR	1.45	1.84	0.39
BwTree	1.69	2.23	0.54
Masstree	2.02	2.51	0.49
Adaptive Radix Tree	1.95	2.30	0.35
Height Optimized Trie	1.78	2.17	0.39
CCEH	1.64	1.82	0.18
Level Hash	1.67	1.92	0.25
CLHT	1.92	2.33	0.41

Efficiently mitigating concurrent write amplification is crucial for constructing scalable PM systems. However, there are several challenges. First, ensuring program correctness is paramount. Addressing concurrent write amplification needs to reorder the read and write requests to the PM, which possibly compromises the program correctness. Many PM applications are designed based on specific system models to provide high performance and crash consistency [9], [10]. Controlling PM access for these applications may disrupt their desired properties.

The second challenge is how to accurately measure concurrent write amplification. Concurrent write amplification refers to the increase in write amplification when multiple threads access the PM concurrently compared to sequential execution [11], [12]. Unfortunately, accurately measuring this concurrent write amplification during program execution is not feasible in practice. Although the overall read and write amplification can be aware through hardware counters provided by Intel [4], it is difficult to accurately and quantitatively measure the amplification from concurrency.

Due to the inability to measure concurrent write amplification, OrdinFS [8] can be only used for sequential read and write operations with known access patterns before execution. If the access patterns of the workload cannot be determined beforehand, it becomes impossible to distinguish whether the observed write amplification comes from the workload's inherent pattern (e.g., a large number of random writes) or concurrent interference.

In this paper, we explore a synchronous non-blocking system to mitigate concurrent write amplification in PM and improve concurrency performance without the needs of any hardware modifications. Our contributions are summarized as follows:

- Synchronous and Wait-Free Concurrent Control: PM applications often leverage system model assumptions, such as the Total Store Order (TSO) model or persistence model [13]. To preserve these model assumptions, we adopt synchronous and wait-free concurrent control. Unlike thread delegation, each thread still directly executes its own requests. By using this synchronous approach, there is no reordering of read and write requests within a single thread, thus maintaining the memory model. Moreover, our approach avoids placing any blocking synchronization operations on the read-write path and ensures that each request is processed within a finite time.
- Delay-based Automatic Control: We discover that inserting appropriate delays can control concurrency and efficiently mitigate write amplification. However, the amount of delays to insert depends on multiple factors, such as concurrency level, thread locality, and hardware performance. This value changes dynamically with the workloads. Therefore, we propose an automatic control system to determine the amount of delays to insert for each thread's request. By continuously observing the system states and estimating the current level of concurrent write amplification, this system dynamically adjusts the frequency of thread access to the PM, effectively mitigating write amplification and significantly improving system performance.
- Experimental validation with real-world workloads and applications. We plan to release the open-source codes for public use in the near future.

II. Background

A. Persistent Memory (PM)

The byte-addressable persistent memory (PM) provides access speeds comparable to DRAM, while guaranteeing persistence. Unlike DRAM, PM is read-write asymmetric. Specifically, the read bandwidths are much higher than the write bandwidths and one write consumes the power of PM three times than that of one read [15].

PM can run in two operating modes, including App Direct and Memory modes. In the memory mode, the system treats PM as a volatile memory like DRAM and uses the DRAM as the cache for PM. In the App Direct mode, PM is persistent and byte-addressable. Applications can access PM directly using load and store instructions without a DRAM cache.

PM resides on the same bus as the DRAM, both of which are controlled by the processor's integrated memory controllers(iMCs). Each iMC maintains separate write and read pending queues (WPQ and RPQ) for each Optane DIMM [16]. Intel proposes Asynchronous DRAM Refresh (ADR) to contain the WPQ in the iMC within the power-failure protection domain [17]. Once data arrives in the WPQ, it will be flushed into PM even if a crash occurs. The Enhanced Asynchronous DRAM Refresh (eADR) extends the protected domain to include the cache [5].

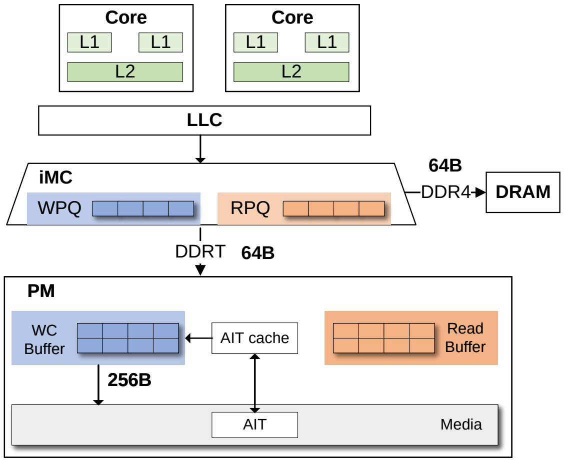


Fig. 1: Overview of Persistent Memory platform.

B. Read and Write Amplification

The iMC accesses the PM with the 64B cache line access granularity using the DDRT protocol [15], [16]. First, the iMC access arrives at the on-DIMM controller. The controller then translates the CPU address to the media address by accessing the Address Indirection Table (AIT). During this process, the PM performs a wear-leveling algorithm to improve its lifetime. Moreover, the actual access to the 3D-xpoint media occurs. However, unlike the PM access granularity of 64B, the 3D-Xpoint media access granularity is 256B. The granularity mismatch requires the controller on the DIMM to perform a read-modify-write operation to translate the 64B access to 256B, ultimately resulting in the Write Amplification (WA). Similarly, the granularity mismatch also leads to read amplification (RA).

The Intel's performance monitor unit (PMU) provides a set of registers to collect the number of bytes written to the 3D-Xpoint media (media writes), the number of bytes issued by the iMC (iMC write), the actual data read from the media (media reads), and data requested by iMC (iMC reads). The WA and RA can thus be calculated by:

$$WA = (media\, writes)/(iMC\, writes)$$

$$RA = (media\, reads)/(iMC\, reads)$$

Existing studies have confirmed that there is a 16KB write-combing buffer (i.e., XPbuffer) and a 16KB read buffer on the DIMM. PM uses the XPbuffer to combine multiple smaller writes to the adjacent memory locations into a single larger (256B) write operation, thus mitigating write amplification.

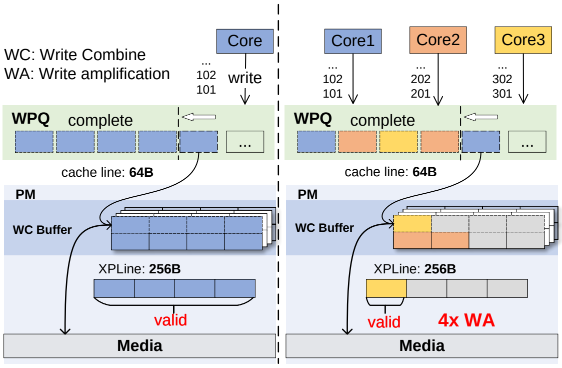


Fig. 2: Examples of (left) how single thread writes data to PM and (right) how concurrent threads cause write amplification on PM.

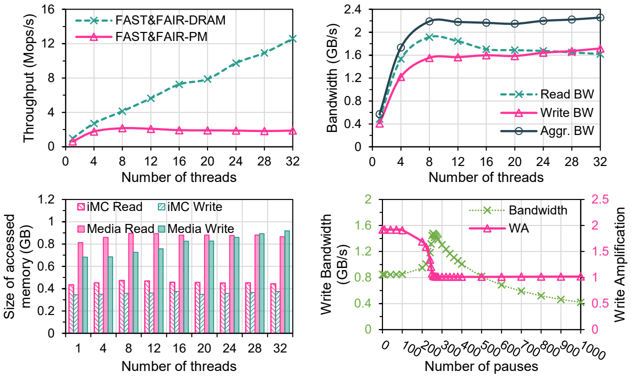


Fig. 3: (a) Throughputs of PM and DRAM based indexes. (b) Read bandwidth, write bandwidth and aggregated bandwidth of PM index ($Aggr.\ BW = ReadBW/3 + WriteBW$). (c) Memory footprint of PM index. The iMC read and write represent the sizes of memory requests issued by CPU, and the media read and write represent the size of data accessed in PM media. (d) The numbers of inserted *_mmpause()* and effective bandwidth of PM.

C. *Concurrent Write Amplification*

Since the XPbuffer contains 64 (16KB/256B) entries, concurrent threads competing for limited resources will lead to more evictions and increased accesses to 3D-Xpoint media. Even if each thread can write sequentially and has good spatial locality, these concurrent writes will eventually be interleaved. When the number of concurrent threads increases, the overall accesses appear more randomly. The interleaved memory access leads to more XPbuffer evictions, increases the number of accesses to 3D-xpoint media, and worsens write amplification.

III. Design Space

A. *Analysis of PM Applications*

To understand how the concurrent write amplification impact the PM applications, we select the PM index, fastfair [10] as the application, use nvalloc [18] as the memory allocator of the index, and test via ycsba [19] as the workload. Our testbed machine is a dual-socket server with two Intel Xeon Gold 6230R CPUs. The system is equipped with 512GB DRAM and 768GB PM (six non-interleaved 128GB Intel Optane DC PM modules are plugged on one socket to avoid NUMA effect).

- (1) The performance of PM based index is limited by PM's small bandwidth. As shown in Fig 3(a), the experimental results present that PM based FastFair achieves a maximum throughput of 2 Mops on a single DIMM with 8 threads, which is much lower than that of DRAM ($\approx$ 10 Mops). Combined with Fig 3(b), we argue that the performance is limited by the PM's bandwidth (2.2 GB/s ideal write bandwidth).
- (2) The performance starts to collapse when the number of threads increases. As shown in Fig 3(a), the throughput of PM index with 32 threads is 20 percent lower than that of 8 threads.
- (3) The performance degradation is caused by the concurrent write amplification. Fig 3(c) shows that when the number of threads increases, only the media writes increase from 0.7 GB to 0.9 GB. In the 8 threads case, there are 0.9GB media reads and 0.7GB media writes. In the 32 threads case, there are 0.9GB media reads and 0.9GB media writes. According to the Intel's manual [15], the cost of a write is approximately 3x that of a read. We thus translate all media reads to media writes and calculate the equivalent media writes with $equi_{writes} = media_{writes} + media_{reads}/3$. The equivalent media writes of 8 and 32 threads are 1.0 GB and 1.2 GB respectively, which means the 32-thread case needs 20 percentage more time to process the requests. This result confirms that the concurrent write amplification is the root cause of the performance degradation.

Here we select the PM index to illustrate the more general case. Note that the impact of the concurrent write amplification in other applications can be much higher than that of the PM indexes. Because the concurrent write amplification mainly affects the sequential I/O while a large portion of memory accesses in indexes are not sequential.

B. *Existing Approaches*

To control concurrent accesses to PM, two intuitive approaches are initially considered: concurrency control and thread delegation.

Concurrency control based approach. The concurrency control approach protects each PM DIMM with a lock. Before accessing the PM, each PM access must acquire the corresponding lock. If the current concurrency of PM DIMM accesses exceeds a specified threshold, incoming PM accesses will be blocked until preceding threads complete their accesses.

Thread delegation based approach. The thread delegation based approach utilizes background threads to access PM. Instead of performing I/O operations by themselves, application threads push their memory requests to a queue (usually a ring buffer) and allow the background threads to delegate their memory accesses. By properly controlling the concurrency of background threads, the thread delegation based approach can eliminate read amplification and write amplification. For instance, each DIMM only allows 2 to 4 background threads to access concurrently.

We summarize the limitation of these approaches as: *bad performance* and *limited applicability*.

Limited performance. To illustrate the limitation of these two approaches, we implement these two approaches and conduct an experimental evaluation. We allow multiple threads to perform sequential write to one DIMM at a 64B granularity and evaluate their performance in terms of two metrics: the write bandwidth and the latency. The write amplification (WA) for sequential write is expected to be 1.0, while the actual WA can be higher than 1.0 due to concurrent interleaving.

We conduct the evaluation with three schemes:

Baseline refers to the plain scheme where each thread directly performs the memory requests.

Lock is a simplified concurrency control based scheme, implemented by a global semaphore. An early version of $Lock$ increments the semaphore whenever a thread needs to access a new XPBuffer and decrement the semaphore when the XPBuffer is no longer accessed. However, the overhead of this early version is unacceptable higher than other two schemes. So we increase and decrease the semaphore every 32 XPBuffers (32x256B) to avoid frequent synchronization operations. We manually implement the semaphore with CAS and FAA operations, since the overhead of pthread semaphore is much higher and cannot guarantee the fairness. Note that in $Lock$, we do not consider how to identify the next access to a new XPBuffer. This is because this scheme is not for practical use but rather for evaluating the optimal performance under sequential write workload, where the access pattern is known at advance.

Delegation is implemented with 4 ring buffers, each configured with a dedicated background thread. This configuration achieves optimal performance in our preliminary experiments. Threads send the memory write requests to background threads via the associated ring buffer.

Fig 4a and 4b show the iMC and media bandwidth of three schemes on concurrent sequential write workload. The iMC bandwidth represents the effective bandwidth, since the iMC requests are generated by the CPU. The media bandwidth represents the total bandwidth consumed in the PM's 3D-XPoint media. The larger the media bandwidth compared to the iMC bandwidth becomes, the larger the write amplification is.

As shown in the Fig 4a and 4b, the effective bandwidth of $Delegation$ are much lower than baseline and $Lock$ due to frequent ring buffer push and pull operations. Thread delegation is unable to fully utilize the bandwidth of persistent memory (PM) for small write operations due to the limitations of the low throughput of the ring buffer. The throughput of a ring buffer typically ranges from 1e6 to 1e7 operations per second (op/s). However, to saturate a single DIMM with 64B writes, a significantly higher throughput of approximately 3e7 op/s is required.

Fig 4c and 4d show the median and tail latency of three schemes. $Lock$ achieves similar latency as baseline while the latency of $Delegation$ is unacceptably higher than other two schemes. Because some threads may starve for a long time, when multiple threads compete for the ring buffer. To avoid high latency, implementations of delegation based scheme should consider how to provide fairness guarantee in their ring buffers.

Our results indicate that $Delegation$ is not feasible for high concurrency applications due to the high overhead of the ring buffer. $Lock$ performs better than $Delegation$, since we reduce the synchronization frequency of $Lock$. Otherwise, the performance of $Lock$ would be unacceptably lower than $Delegation$ and $Baseline$. In summary, both of these two approaches are limited by the bad performance due to high synchronization overheads.

Limited applicability. Despite of their high overheads, the applicability of these two approaches is also limited.

The read and write operations in delegation based approaches are not completed immediately, but need to wait in the queue. Therefore, programs that require data freshness cannot adopt thread delegation. Unfortunately, most in-memory applications need to read/write the latest submitted data (e.g., data synchronization). Only a few file operations and log writing, such as Flat-Store [7], can use the thread delegation.

Some programs heavily rely on the program's concurrent access and data competition behavior, such as lock-free trees, lock-free hashes, etc. Adding blocking synchronization operations to these programs will unintentionally change the original semantics of these programs. Therefore, lock-based schemes cannot be utilized in these programs.

In summary, both of these two approaches suffer from high synchronization overheads and limited applicability.

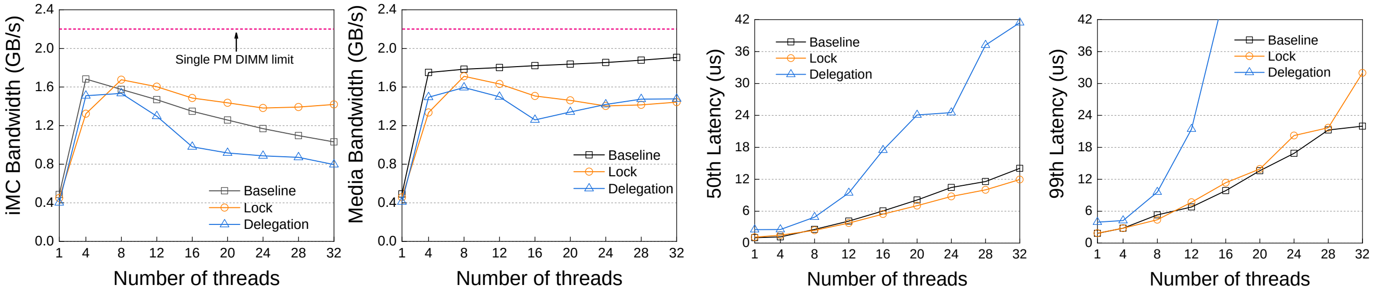
IV. Design

In this paper, we propose a delay insertion mechanism to mitigate concurrent write amplification. Specifically, the system determines a delay, D, for each memory access instruction issued by the working thread. The working thread then waits for this predefined delay, D, before executing the memory access instruction.

The implementation of this delay strategy offers the following advantages:

(1) Performance Enhancement: By deciding on a single waiting duration upfront, the system eliminates the overhead of frequent synchronization operations, thereby significantly improving overall system performance.

(2) Wide Applicability: Managing memory access behavior through a finite waiting period imposes minimal interference



(a) iMC bandwidth (b) Media bandwidth (c) Median latency (d) Tail latency

Fig. 4: Comparisons of existing schemes.

on the data contention behavior of programs, making this strategy suitable for a broader range of application scenarios. The implementation avoids deadlocks caused by blocking-based concurrency control and prevents the memory access order from being changed by background threads.

Unlike blocking-based infinite waiting, the finite waiting strategy employed in this paper ensures that, even if certain conditions are not met during the waiting period, the thread can continue execution after the predetermined delay, thus preventing potential deadlock issues.

Unlike delegation based approaches, the memory access operations in this strategy are executed directly by the working thread, without delegating the handling of data contention logic to a delegation thread. This approach avoids additional interference with data contention and ensures the directness and efficiency of memory access operations.

To further illustrate how the insertion of delays addresses the issue of concurrent write amplification, we present a simplified experimental setup.

In this scenario, multiple threads perform sequential writes to persistent memory (PM), where each thread waits for a predetermined delay, D, before executing its write operation.

The impact of inserting the delay D on bandwidth is depicted in Figure 3(d), resembling a hook function. Initially, the concurrency of multiple threads results in an effective bandwidth that is less than the consumed bandwidth. However, when the inserted delay increases, the concurrent interference is mitigated, and the effective bandwidth gradually rises to a level close to the consumed bandwidth.

At this point, the influence of concurrent interference has become minimal. Further increasing the inserted delay slows down the memory access speed, leading to a continual decrease in the effective bandwidth. The experimental results demonstrate that by selecting an appropriate delay D, the effects of concurrent interference can be mitigated, leading to improved concurrent performance of persistent memory.

Consequently, our task needs to determine an optimal delay D, given the current working threads and memory access instructions, such that the effective bandwidth of PM is maximized.

Adaptive delay insertion. The system periodically monitors the number of concurrent threads accessing PM and the write amplification factor obtained from hardware performance monitor counters. Once the write amplification factor and the number of concurrent threads are both higher than $WA_{threshold}$ and $Thd_{threshold}$, the system needs to insert delay to these threads. The objective of the delay insertion mechanism is to select an appropriate D value that maximizes the effective bandwidth B. The relationship between D and the effective bandwidth B, as illustrated, exhibits a concave function with D representing the inserted pause time on the x-axis and the effective bandwidth on the y-axis. There exists an optimal point D_i where the effective bandwidth reaches its peak.

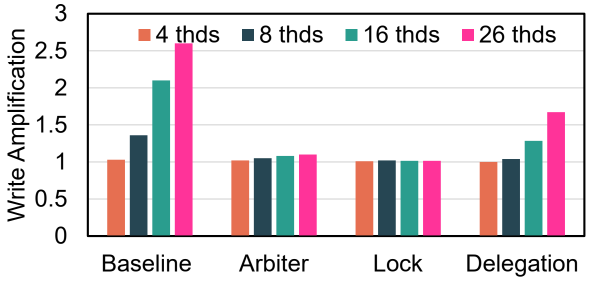


Fig. 5: Write amplification factor with various thread numbers. Write amplification factor is calculated by $Media\_Bandwidth/iMC\_Bandwidth$.

To this end, we employ the simplest form of an exponentially weighted moving average (EWMA) to estimate B and a PID (Proportional-Integral-Derivative) controller to adjust D. The PID controller's parameters are manually tuned to meet our requirements.

V. Performance Evaluation

A. Experimental Setup

Configuration. We evaluate Arbiter on a dual-socket server with two Intel Xeon Gold 6230R (26 core, 52 hyperthread) and 6×128GB Intel Optane DC Persistent Memory. All Optane PMs are plugged on the first socket and configured as non-interleave mode. We carry out all memory allocations and threads to the first socket.

Benchmarks. We leverage a *multi-threaded sequential write* as the micro-benchmark to show the performance improvement of using Arbiter under ideal cases. In the *multi-threaded sequential write*, each single thread writes data sequentially

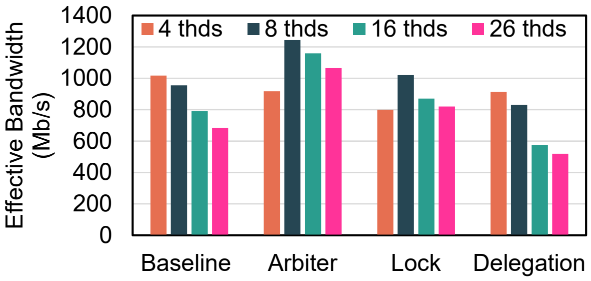


Fig. 6: Effective write bandwidth with various thread numbers. Effective write bandwidth (iMC Bandwidth) is obtained from the hardware counters of iMC-level memory requests.

to PM, which is expected to obtain a 1.0 write amplification. When multiple threads concurrently write, the performance slowdown comes from concurrent write amplification, which shows how our system eliminates the concurrent write amplification. To avoid the impact of NUMA nodes, the maximum number of threads is set to 26, which is exactly the number of physical cores within one socket in our machine.

Comparisons. We compare Arbiter with existing state-of-the-art schemes that can mitigate concurrent write amplification. Baseline is the basic scheme that does nothing to mitigate concurrent write amplification (CWA) in most CWA-unaware PM applications. Delegation is an approach that only uses background threads to access PM to avoid CWA. The main design comes from Ordinfs [8]. We implement Delegation by ourselves since the thread delegation is only a part of the file system in Ordinfs. Lock tries to strictly control the concurrency of PM by using lock-based operations. We implement the lock-based scheme for comparisons.

B. Micro-Benchmark Results and Analysis

Write Amplification. The main design goal of Arbiter is to eliminate the concurrent write amplification. For single threaded sequential write workload, the iMC-level memory write is equal to media-level memory write, so the write amplification factor is 1.0. When multiple threads perform sequential writes to PM concurrently, all increased write amplification comes from concurrent write amplification. In this case, we compare the write amplification factor (WA) in Baseline and Arbiter. As shown in Fig 5, when the number of concurrent threads increases, the WA of Baseline grows rapidly. Arbiter eliminates CWA by carefully controlling the concurrency of PM accesses with delay insertion and thus keeps WA close to 1.0. The evaluation results show that Arbiter can effectively eliminate the CWA.

Effective Bandwidth. In addition to eliminating CWA, end-to-end performance is also an important metric. To check if Arbiter can eliminate CWA while also improve the performance, we select the effective bandwidth as the performance metric, because most PM applications are memory-bounded due to PM's limited write bandwidth [15], [16]. Higher effective bandwidth often indicates better end-to-end performance. As shown in Fig 6, when concurrency increases, Baseline suffers from significant performance slowdown due to CWA. Arbiter also suffers from CWA but the performance decrease is small. Arbiter achieves at most 47 percent performance improvement compared to the Baseline under high concurrency workloads.

VI. Related Work

Optane DCPMM [1] aims to provide high performance. Many schemes [11], [16], [20] confirm that the write amplification scenario of Optane DCPMM is caused by the mismatched access granularity between CPU cache and the PM's internal XPBuffer. Based on these insights, there are many efforts in preventing applications from the impact of PM's write amplification. Most schemes focus on the design of the PM-aware applications, like hash-tables [21], [22], [30], B-trees [10], [28], key-value stores [6], [7], [23], transactional memories [24], memory allocators [18], [29], and algorithms [25]. To mitigating the write amplification, they either merge small PM writes or align the data structures for the specific applications. While successfully reducing the number of PM writes, these optimizations are highly dependent on the programmers, requiring deep understanding upon both PM and applications to be optimized.

To be more general, some schemes leverage the application-level I/O schedules to improve PM's performance. Specifically, Ordinfs [8] mitigates PM's write amplification at the file system level by allowing only the background thread to access PM. MT2 [26] alleviates interference between multiple processes by regulating the memory bandwidth of PM and DRAM. Dicio [27] utilizes the CPU scheduler to mitigate interference between latency-critical and best-effort processes. These schemes are effective in mitigating write amplification caused by multiple processes and overlook the write amplification caused by multiple threads, which is more possible to exist within the same process and requires more fine-grained control of PM accesses.

Unlike them, our work emphasizes the significance of the concurrent write amplification (CWA) and proposes a more general I/O scheduling system to mitigate the CWA.

VII. Conclusion

This paper presents Arbiter, a non-blocking synchronous I/O scheduling mechanism that mitigates the concurrent write amplification in the persistent memory. The key insight is to avoid thread interleaving caused by high concurrency. To alleviate concurrency, Arbiter adaptively inserts a proper number of delays into PM applications. The insertion mechanism allows Arbiter to be applied to more PM applications without changing the program semantics. Performance evaluation shows that Arbiter can mitigate the concurrent write amplification and improve the effective bandwidth of PM under highly concurrent cases.

VIII. Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. U22B2022 and 62125202. The authors thank Zhangyu Chen, Yujie Shi, and Aoyang Tong for their constructive comments and discussions .

References

[1] RYAN SMITH. Intel Announces Optane Storage Brand For 3D XPoint Products. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for3d-xpoint-products, 2015.

[2] CXL Consortium. Compute Express Link Specification Revision 3.1. https://www.computeexpresslink.org/download-the-specification, 2024.

[3] The Volatile Benefit of Persistent Memory. https://memcached.org/blog/persistent-memory/, 2020.

[4] ipmctl. https://github.com/intel/ipmctl, 2021.

[5] INTEL. Intel® Optane persistent memory 200 series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optanepersistent-memory/optane-persistent-memory200-series-brief.html, 2020.

[6] Wang, Jing, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. "Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory." In ATC 2020.

[7] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient logstructured key-value storage engine for persistent memory. In ASPLOS 2020.

[8] Diyu Zhou, Yuchen Qian, Vishal Gupta, and Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. Odinfs: Scaling PM Performance with Opportunistic Delegation. In OSDI 2022.

[9] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free Concurrent Level Hashing for Persistent Memory In ATC 2020.

[10] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-tree. In FAST 2018.

[11] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, Hong Jiang. Characterizing the Performance of Intel Optane Persistent Memory. In Eurosys 2022.

[12] Jinyoung Oh, Youngjin Kwon. Persistent Memory Aware Performance Isolation with Dicio. In APSys 2021.

[13] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. Extending Intel-x86 Consistency and Persistency: Formalising the Semantics of Intel-x86 Memory Types and Non-temporal Stores. In POPL 2022.

[14] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[15] INTEL. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[16] Jian Yang, Juno Kim, Morteza Hoseinzadeh, and Joseph Izraelevitz. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In FAST 2020.

[17] Andy Rudoff. Deprecating the PCOMMIT instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, 2016.

[18] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators. In ASPLOS 2022.

[19] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing, 2010.

[20] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and Modeling NonVolatile Memory Systems. In MICRO 2020.

[21] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. In VLDB 2020.

[22] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In FAST 2019.

[23] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. ChameleonDB: A Key-Value Store for Optane Persistent Memory. In Eurosys 2021.

[24] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. ArchTM: Architecture Aware, High Performance Transaction for Persistent Memory. In FAST 2021.

[25] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel SemiAsymmetric Graph Algorithms for NVRAMs. In VLDB 2020.

[26] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In FAST 2022.

[27] Jinyoung Oh and Youngjin Kwon. Persistent Memory Aware Performance Isolation with Dicio. In APSys 2021.

[28] Jinlei Hu, Zijie Wei, Jianxi Chen, and Dan Feng. RWORT: A Read and Write Optimized Radix Tree for Persistent Memory. In ICCD 2023.

[29] Xiangyu Xiang, Yu Hua, and Hao Xu. PMA: A Persistent Memory Allocator with High Efficiency and Crash Consistency Guarantee. In ICCD 2023.

[30] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, Jiajie Sheng. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems.In FAST 2023.