PATS: Taming Bandwidth Contention between Persistent and Dynamic Memories

Shucheng Wang*, Qiang Cao*✉, Ziyi Lu*, Hong Jiang† and Yuanyuan Dong‡
*Wuhan National Laboratory for Optoelectronics, HUST, ‡Alibaba Group,
†Department of Computer Science and Engineering, University of Texas at Arlington

***Abstract*—Emerging persistent memory (PM) with fast persistence and byte-addressability physically shares the memory channel with DRAM-based main memory. We experimentally uncover that the throughput of application accessing DRAM collapses when multiple threads access PM due to head-of-line blockage in the memory controller within CPU. To address this problem, we design a PM-Accessing Thread Scheduling (PATS) mechanism that is guided by a contention model, to adaptively tune the maximum number of contention-free concurrent PM-threads. Experimental results show that even with 14 concurrent threads accessing PM, PATS is able to allow only up to 8% decrease in the DRAM-throughput of the front-end applications (e.g., Memcached), gaining 1.5x PM-throughput speedup over the default configuration.**

***Index Terms*—Persistent Memory, Memory Contention, Thread Scheduling**

I. Introduction

Persistent memory is considered as 'persistent DRAM' and became commercially available for the first time in April 2019 with the launch of the Intel Optane DC Persistent Memory Model (PM) [1]. Therefore, DRAM-PM architecture has been widely used in data-intensive scenarios such as PM-based file systems [2] and key-value stores [3]–[5]. However, unlike DRAM, PM has asymmetric read and write performances that are lower than DRAM, with limited access parallelism [1].

In the current DRAM-PM architecture, DRAM and PM physically share high bandwidth memory channels, e.g., about 60GB/s bandwidth per channel at 2.6GHz [6]. Despite a potential memory-channel contention between DRAM and PM, there has not been any study mentioning, much less studying, such contentions in the literature to the best of our knowledge. We speculate that this oversight may in part stem from the fact that applications on the traditional DRAM-disk architecture rarely incur significant I/O interference between DRAM and disk. To understand and study this issue, we conduct an experiment in a real DRAM-PM platform with three DRAMs and a 128GB PM, detailed in Table I. The PM is configured in the AppDirect mode and formatted by the Ext4-DAX file system. MBW [7], a memory benchmark, performs memory copy operations (i.e., *memcpy()*) upon the DRAM. Meanwhile, a set of threads sequentially read/write data on the PM with 4KB request size. Fig.1 reports that the throughput of MBW drops dramatically as the number of PM-accessing threads (*PA-threads*) increases. Specifically, the MBW throughput decreases by more than 90% at 14 PM write-threads. In contrast, the aggregated throughput of read PA-threads increases quickly by 2.2 times until 5 PA-threads, and then decreases marginally to about 2 times. The

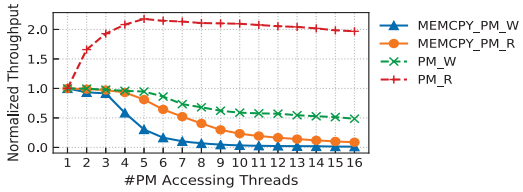


Fig. 1. Throughput of a *memcpy()* thread accessing DRAM while an increasing number of threads accessing PM. Both DRAM and PM throughputs with multiple PM-threads are normalized to that of a single PM-thread.

experimental result reveals that multiple threads accessing PM can seriously degrade the performance of applications accessing main memory, such as Memcached [8].

We experimentally and holistically analyze the DRAM-PM contention behavior, uncover that slow PM-accesses could severely impede the performance of fast DRAM-accesses under concurrent PM-threads. Previous researches focus on inter-thread interference upon shared DRAM by prioritizing different threads via operating system and scheduling requests based on their priority by the memory controller, to tradeoff between performance and fairness [9]. However, they not only modify the hardware memory-controller, but also introduce extra request handling overhead in the request critical path.

In this paper, we first build a contention model to dynamically determine the maximum contention-free concurrency of PM-threads under a given hardware setting and workload. We further propose *PM-Accessing Thread Schedule mechanism* (PATS), a PM-side thread dynamic scheduling mechanism integrating within Persistent Memory Development Kit (PMDK) library, to ensure the performance of both accessing DRAM and PM by effectively taming the DRAM-PM contention. PATS avoids the costly request-level hardware-assisted scheduling and extra scheduling delay for fast DRAM accesses.

This paper makes the following contributions:

- We experimentally reveal that multiple threads accessing PMs can cause performance collapse of DRAM.
- We propose an effective request scheduling scheme, PATS, guided by a contention model, to effectively harness the parallelism of concurrent PA-threads.
- We implement a PATS prototype. The evaluation shows that PATS can dynamically maintain DRAM-access performance independent of PM accessing behaviors. Meanwhile, PATS effectively improves the overall throughput of PM by up to 45% under multi-threaded mode.

The rest of paper is organized as follows. Section II presents the background for Optane PM and the motivations for PATS.

978-3-9819263-6-1/DATE22/©2022 EDAA

Section III analyzes the behaviors of DRAM-PM contention. Section IV describes PATS's design. We evaluate PATS in Section V and describes related works in Section VI . VII concludes this paper.

II. Background and Motivation

A. Optane PM

Emerging persistent memory is attractive because of its near-DRAM latency, byte-addressability and durability. The Intel Optane DC Persistent Memory Model (PM) [1] is the first commercialized persistent memory product. In a PM-based system, shown in Fig.2, a CPU's integrated memory controller (iMC) communicates with the Optane PM and DRAM via memory channels. Each memory channel can achieve up to 60GB/s bandwidth at 2.6GHz. The Intel Xeon Salable Processor, as the only CPU supporting PM at the present, has two internal iMCs, each of which supports up to three memory channels. Each memory channel can deploy DRAM DIMM and/or PM DIMM. For example, iMC1 connects DRAM DIMMs located in A1/B1/C1 and PM DIMMs in A2/B2/C2, respectively.

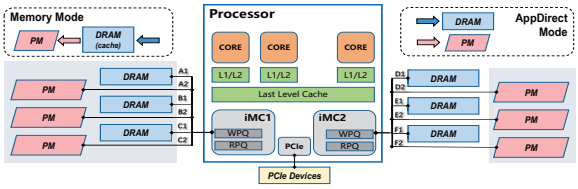


Fig. 2. Overview of an Intel Optane platform.

The Optane PMs work in either the Memory mode or the AppDirect mode. The Memory mode merely utilizes DRAM and PM in memory hierarchy where PM is a backend larger 'DRAM' with DRAM as its cache and is transparent to users and applications. In the AppDirect mode, PM is a storage device accessed by applications via memory mapped interface (mmap) [10]. The DAX-mapped I/O can be executed with loads/stores and bypass OS page cache [11], thus reducing software overhead and context switches between user and kernel space. In the rest of this paper, we employ the AppDirect mode with DAX-mmap to access PM, unless otherwise noted.

B. Accessing DRAM and PM simultaneously

The bandwidth of PM is 1-2 orders of magnitude lower than DRAM. Therefore, modern computer architectures still depend on DRAM-based main memory to guarantee the performance of data-intensive applications while utilizing PM as fast disk for persistence. The emerging DRAM-PM storage architecture with the PM AppDirect mode has been increasingly applied in practical scenarios, such as file systems [2], [10], [11], key-value stores [3], and HPC systems [12]. In these cases, both PM and DRAM are intensively accessed simultaneously.

To better understand the interplay between PM and DRAM when they are simultaneously accessed, we design a test approach. Specifically, a thread continuously inserts data into Memcached in DRAM mimicking typical accessing behaviors of data-intensive applications. Meanwhile, FIO [13] continuously issues 4KB-size write requests into the PM with a varying number of PM-accessing threads (**PA-threads**). The IO-engine is libpmem provided by PMDK [14]. The test results are shown in Fig.1, depicting the collapses of the memcached throughput.

TABLE I
Evaluation Platform Specifications

Components	Configurations
Processor	Single Socket Intel Xeon Gold 5218, 16 Cores, 32MB LLC 2 iMCS × 3 channels
Memory	32GB 2666MHz DDR4 × 3 128GB PM in AppDirect mode × 1
Operating System	Ubuntu 20.04 LTS with Linux kernel 5.1.0

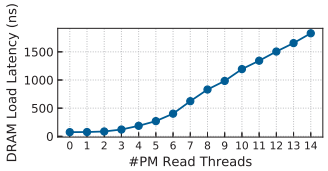


Fig. 3. The read latency of DRAM under multi-threaded PM reads.

We further measure the DRAM read latency using LMBench [15] when reading from PM with multiple threads. The results, shown in Fig.3, indicate a sky-rocketing of the average DRAM latency from about 75 ns at 2 PA-threads to 1820 ns at 14 PA-threads. This clearly illustrate a drastic increase of DRAM latency as a result of the bandwidth contentions.

These interesting findings motivate us to further explore their root causes, behaviors, triggering conditions, and finally an effective solution in the emerging DRAM-PM architecture.

III. Analysis and Contention Model

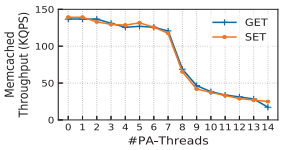
A. Experimental Setup

We conduct our experimental analysis on a single-socket server with configurations specified in Table I. We adopt Memcached and MBW as data-intensive applications accessing DRAM. Memcached is a widely used in-memory key-value store. A program continuously launches GET or SET requests to Memcached with a single thread and calculates its average throughput. MBW is an in-memory benchmark tool to accurately evaluate the performance of different memory operations. MBW has three basic operations: MEMCPY, DUMB, and MCBLOCK. MEMCPY invokes the *memcpy()* function to copy a whole data array to another memory area. MCBLOCK cuts the array into a series of 256KB-sized blocks and copies them one by one. DUMB executes an assignment operation between two array elements. MBW also runs in a single thread. To avoid the dataset being completely cached in the CPU, we set the dataset size to 10GB that far exceeds the cache capacity. In addition, we use FIO to generate a sequential read/write workload upon PM. The request size is set from 512B to 4KB. The maximum number of PA-threads is set to 14.

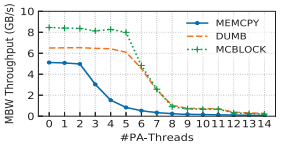
We bind the threads of FIO, Memcached and MBW to different CPU cores, thus avoiding the overhead of thread scheduling and resource contention among CPU cores.

B. The Problem and Its Root Cause

We invoke MBW and Memcached and measure their performances. At the same time, we generate 1-14 PA-threads to perform continuous write upon PM in the background. The I/O size of write requests is fixed at 4KB. The experimental results,



(a) Throughput of Memcached



(b) Throughputs of MBW with different memory test operations

Fig. 4. Memcached and MBW with different numbers of PA-threads.

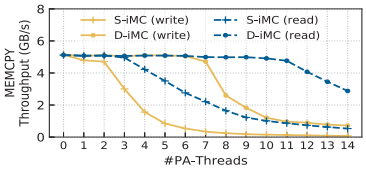


Fig. 5. Throughput of the MEMCPY operation with PM and DRAM located in the same and different iMCs.

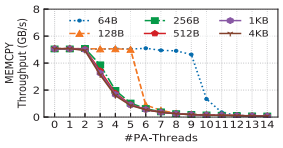


Fig. 6. Throughput of MBW with multiple threads access PM in different requests I/O sizes.

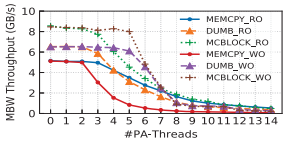
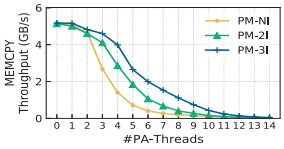
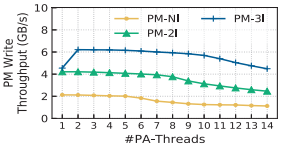


Fig. 7. MBW throughput when employing read-only and write-only workloads on PM.



(a) MEMCPY operation in MBW



(b) Sequential PM write

Fig. 8. MBW and PM throughput with different numbers of interleaved PMs.

shown in Fig.4, indicate that the throughput first decreases slightly until it reaches a threshold, or a "knee" point, where it starts to drop precipitously in a "waterfall" region until an "ankle" point whereon it flattens out again smoothly reaching a floor. More specifically, at 14 PA-threads, the throughputs of the GET and SET operations in Memcached and three memory operations in MBW decrease more than 78% from their peak levels when PM is not accessed by any PA-thread.

The sharpness of the knee and the steepness of the waterfall region are dependent on workloads, especially for MBW. The Memcached throughput with 7 PA-threads is only 8% lower than the peak throughput while dropping to 65% with 9 PA-threads. The SETs and GETs behave similarly due to read/write symmetry on DRAM. MBW suffers a steeper throughput drop than Memcached with relatively lighter DRAM-access load. With 8 PA-threads, the MBW performance drops to about 85% lower than its peak, while that percentage in Memcached is less than 55%. The peak throughputs of the three operations in MBW are different significantly. The floor throughput of the MEMCPY operation at 14 PA-threads is only 5% of its peak throughput. MCBLOCK has the highest peak throughput and the highest knee point, but is also the steepest in the waterfall region between 5 and 8 PA-threads. In the experiment, the LLC miss rates of MBW remain unchanged at about 99.2%.

1) iMC: We conduct experiments to analyze the role iMC plays in the resource contention. In previous experiments, the PM and three DRAMs are installed in channel A2 and channels A1, B1 and C1 respectively, while the iMC2 is totally idle, an Intel recommended [1] iMC configuration referred to as `S-iMC`. In this experiment, we move the PM DIMM from channel A2 in iMC1 to channel D2 in iMC2, and leave all DRAMs in iMC1, an iMC configuration referred to as `D-iMC`.

Fig.5 shows the performance of the MEMCPY operation with `D-iMC` and `S-iMC`. Both cases have a similar "knee-ankle" profile, but `D-iMC` has higher knee points. When multiple threads read PM, the throughput knee of MEMCPY is 7 PA-threads under `D-iMC`, and the throughput drops by 35% at 14 PA-threads. In `S-iMC`, the knee point is 3 PA-threads and the performance drops by up to 80% at 14 PA-threads.

2) Request Size: We investigate the effect of request size using MBW. The request size ranges from 64B to 4KB. Fig.6 shows that a workload with requests larger than 256B triggers the performance collapse with a smaller number of PA-threads. Previous studies [1] confirm that PM with a 256B-sized built-in XP-Buffer saturates its bandwidth with I/Os larger than 256B.

3) PM-accessing Patterns: We explore the effect of PM's read and write patterns. The experiment issues sequential reads and writes to PM with a fixed request size of 4KB. Fig.7 reports that multi-threaded writes to PM cause more severe memory throughput degradation than reads in the waterfall region. The write throughput of MBW drops by more than 70% when PM serves two more threads after the knee point, while the read throughput decreases by only 30% in the same case.

4) PM Devices: In this experiment, we finally vary the number of PM devices from 1 to 3 in the platform. A single PM only uses the non-interleaved mode (denoted as `PM-NI`), while multiple PMs are configured in the interleaved mode (denoted as `PM-2I` and `PM-3I`). Results in Fig.8 show that the platform with more interleaved PM devices suffers less bandwidth contention under the same multi-threaded workloads. This is because the interleaved mode spreads the requests as evenly as possible across multiple PM devices. As shown in Fig.8(b), `PM-3I` has nearly 3x bandwidth than that in `PM-NI`. The PM in the interleaved mode improves capability to serve concurrent requests and makes them less likely to be blocked.

5) Summary: Due to the fact that an iMC has limited channels and queue-length, when a large-sized PM request sits at the head of the queue, its relatively slow memory-transaction will heavily hamper the fast DRAM-requests behind it in the queue, resulting in the notorious head-of-line blockage. Through extensive experiments, we find that it is the number of PA-threads exceeding a threshold that primarily triggers the performance collapse of DRAM applications, with the detailed and complex queueing-behavior likely playing a secondary role. We also test the impact of other factors on the DRAM-PM bandwidth contention, such as different PM-aware file systems, different DIMM placement policies, ratio of DRAM:PM capacities and CPU LLC hit ratios, and find that they basically do not affect the severity of the contention.

TABLE II
MODEL PARAMETERS AND CORRESPONDING DESCRIPTIONS

	Paremeters	Descriptions
Input	T_{mc}	Threshold of tolerable memory contention impact
	N	Number of PMs in interleaved mode
	S	Average request size
	Wr	Ratio of write requests
Output	T_S	Threshold of runnable PA-Threads count

C. Contention Model

It is clear that finding or projecting the aforementioned "knee" point, i.e., the threshold number of PA-threads beyond which the DRAM performance collapses, for a given workload and platform configuration, is critical for taming the bandwidth contention. Based on the above observations and analysis, we build a contention model, expressed in Eq.1, that provides a contention profile projecting this "knee" point. The output T_S of the model is the maximum contention-free concurrency of runnable PA-threads, which is determined by workload patterns, the number of PMs, and the threshold T_{mc}.

$$T_S = \lfloor (1 + 2T_{mc}) * P_{rw} + (1 + T_{mc}) * P_N + P_{size} \rfloor \quad (1)$$

The key parameters are described in Table II. T_{mc} represents the performance degradation severity predefined by the user, which has range of $[0, 1]$, where 0 and 1 mean lower-bound and upper-bound contention on DRAM, respectively. The default value is set as 0.2 for a trade-off between PM and DRAM performance.

P_{rw} represents the read/write ratio of the workload on the bandwidth contention. Results in Fig.7 show that writes to PM incur the performance collapse of DRAM applications at a smaller number of PA-threads than reads.

$$P_{rw} = 3 - Wr \quad (2)$$

N is the number of PM devices. Fig.8 shows that more PM devices help weaken the bandwidth contention under the same multi-threaded workloads.

$$P_N = N - 1 \quad (3)$$

P_{size} represents the impact of the average request size.

$$P_{size} = \begin{cases} 0 & S \geq 256Byte \\ \frac{2*256B}{S} - 1 & S < 256Byte \end{cases} \quad (4)$$

Finally, we validate the accuracy of the contention model. For example, the evaluation shown in Fig.6 runs with a write-only workload (i.e., $Wr = 1$), with an average request size of 4KB (i.e., $S = 4096B$) and a single PM (i.e., $N = 1$). With these inputs, the model could shape the actual results of the knee point and the waterfall region. For example, the model outputs $T_S = 2$ when T_{mc} is 0.1 and $T_S = 4$ when T_{mc} is 0.5. In addition, we further evaluate the accuracy of the contention model on rehabilitating memory performance in Section V.

IV. DESIGN

The architecture of PATS is shown in Fig.9. PATS has two dedicated scheduling threads, Collector and Selector, and maintains three global data structures, Req-Info array, Running-Lock array, and Serving-Thread count. The Req-Info array links each PA-thread's own Req-Info that records the request read/write type and length. A PA-thread accesses PM only after obtaining the Running-Lock.

PATS assigns each PA-thread a globally unique thread ID (TID), which is less than the global array length value. A

Algorithm 1 PATS scheduling

Input: N, T_{mc}, T, S, Wr, defined in Table II

```
1: Function Selector()
2: for each blocked PA-thread with thread ID as TID do
3:     if T_S >= T then: unblock(Running-Lock-Array[TID])
4:     end if
5: end for
6: Function Collector()
7: for each PA-thread with thread ID as TID do
8:     S, Wr += Req-Info-Array[TID]
9: end for
10: S, Wr = average(S, Wr)
11: T_S = model(S, Wr, T_mc, N)
```

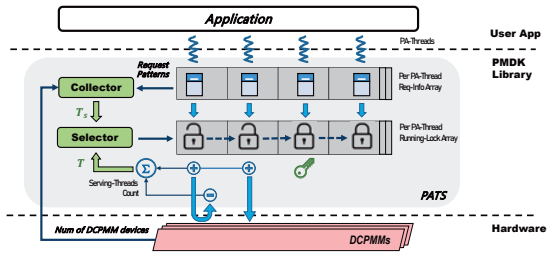


Fig. 9. The architecture of PATS scheduling.

PA-thread uses its own TID to locate its corresponding Req-Info and Running-Lock. The per-thread structures are updated by their owner threads. When a PA-thread is terminated, PATS recycles its TID and clears its corresponding data structures.

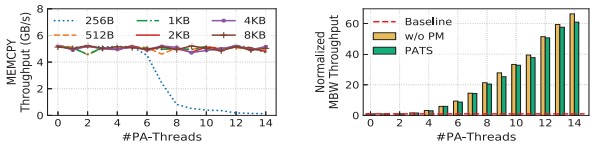
At runtime, a PA-thread sets its running-lock to "1" (i.e., the locked state) at initialization. Before launching a PM request, the PA-thread updates its Req-Info in the global Req-Info array, and is locked. Once unlocked by Selector, the PA-thread adds 1 to the Serving-Thread count and then performs its relevant PM request. After request completion, the PA-thread subtracts 1 from the counts and clears the request info from Req-Info array. Collector traverses the current Req-Info array within a time-window and calculates the maximum number of runnable PA-threads (i.e., parameter T_S in the model). Selector monitors Serving-Thread count at runtime (i.e., parameter T in the model) and decides when to release blocked PA-threads.

As shown in algorithm 1, collector traverses all the non-empty Req-Info arrays to collect request patterns of all PA-threads periodically and calculates T_S according to the contention model. At runtime, Selector handles each Running-Lock in a round-robin manner, which ensures fairness among PA-threads and avoids starvation. PATS does not block PA-threads with very small request sizes (e.g., less than 256B by default) to avoid high scheduling overhead. We implement and integrate the PATS prototype in PMDK to perform scheduling on PA-threads invoked by applications.

V. EVALUATION

A. Experiment Setup

We run experiments in the platform as described in Section III-A. We use FIO to access PM with a varying number of threads and different block sizes, while executing MBW and Memcached in DRAM with a single thread. The **Baseline** is accessing PM and DRAM simultaneously without **PATS**. The value of N and T_{mc} are 1 and 0.2, respectively.



(a) In PATS (b) Normalized to Baseline

Fig. 10. Throughput of the MEMCPY operation with different numbers of PA-threads and varying request sizes.

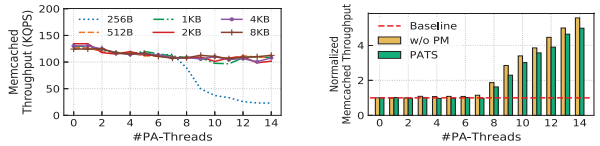


Fig. 11. Throughput of Memcached with different numbers of PA-threads and varying request sizes.

B. Performance

1) DRAM: The design goal of PATS is to mitigate the DRAM performance collapses when accessing PMs and DRAMs simultaneously. In experiments, we measure the operational throughput of MBW and Memcached, and compare the results with those in the Baseline. We set the request size from 256B through 8KB, and invoke up to 14 PA-threads.

The results of the Baseline are shown in Fig.4 and Fig.6. The MEMCPY throughput degrades from the peak of 5.1 GB/s to the bottom of 500 MB/s with 14 PA-threads. Compared to the Baseline, the PATS results shown in Fig.10(a) indicate that the MEMCPY throughput in PATS remains stable independent of the PM accessing behaviors. With request size and number of PA-threads varying, the fluctuation of the MEMCPY throughput in PATS is controlled within a 6% margin. It indicates that PATS can adapt to different workload characteristics. However, workloads with 256B-sized PM requests still incur DRAM performance collapse. This is because PATS does not schedule PM requests smaller than 256B by default, to ensure that the PM performance does not suffer from PATS overhead.

Moreover, we define the MEMCPY throughput without using PM (`w/o PM`) as the peak throughput and compare it with that in PATS, both of which are normalized to Baseline. Fig.10(b) shows that the MEMCPY throughput increases to up to 60x that of Baseline, while dropping by less than 5% from `w/o PM` at 14 PA-threads. Fig.11 demonstrates an identical trend. With 14 PA-threads, the Memcached throughput increases to up to 5x that of the Baseline, while slightly decreased by less than 8% from the peak throughput.

2) PM: We measure the write throughput of PM in PATS, as shown in Fig.1, the peak write throughput of PM sustains with less than 5 PA-threads and drops by over 45% with 14 PA-threads. Fig.12(a) shows PM write throughput under the PATS scheduling. When PM requests size is larger than 1KB, the throughput remains at its peak regardless of the number of PA-threads. Workloads with requests smaller than 256B will not be scheduled by PATS, thus performing similarly as the Baseline. Moreover, Fig.12(b) shows that 4KB-sized requests with 14 PA-threads obtain 1.6x throughput gain in PATS over the Baseline. This is because PATS can effectively control the number of concurrent requests and mitigate the impact of bandwidth contention on PM's own performance.

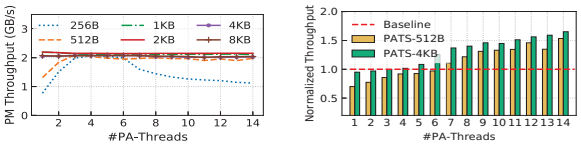
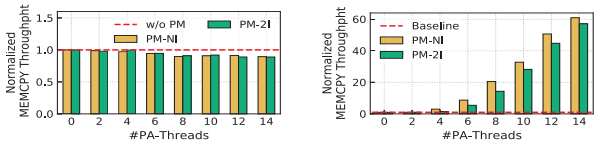


Fig. 12. PM throughput with different numbers of PA-threads and varying request sizes.



(a) Normalized to w/o PM (b) Normalized to Baseline

Fig. 13. MEMCPY throughput with different number of PM devices.

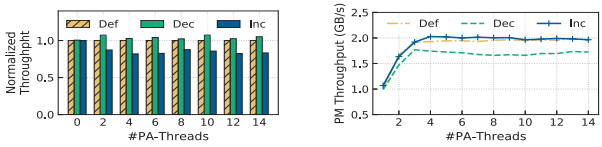
We evaluate the overhead of PATS on PM performance. We select PM throughput results in PATS with two typical request sizes of 512B and 4KB and normalize them to the Baseline. Fig.12(b) shows that with the 512B request size, the PM throughput at one or two PA-threads is decreased by up to 25% of the Baseline. This is because small size requests are more sensitive to the scheduling overhead of PATS. It is for this reason that PATS does not schedule requests with extra small size. However, we observe that PM throughput with 512B-sized requests is still better than the Baseline beyond 6 PA-threads. This indicates that the PATS scheduling overhead can be hidden with multiple PA-threads to some extent. In addition, the CPU load of PATS is less than 1% of the total CPU resources.

C. Sensitivity Study

1) Different numbers of PMs: We explore the effectiveness of PATS with different numbers of PM devices (i.e., parameter N in the contention model). We configure two PMs (`PM-2I`) in the interleaved mode and compare it with a single PM (`PM-NI`). Results under the PATS scheduling are normalized to that in Baseline and that without using PM (`w/o PM`), respectively.

Fig.13(a) shows that, in both `PM-NI` and `PM-2I`, the MEMCPY throughput in PATS is reduced slightly, by 11% from the peak performance without using PM, which satisfies the threshold T_{mc}. It shows that PATS adapts to different hardware environments with varying PMs. Fig.13(b) demonstrates that PATS improves the MEMCPY throughput by 5.4x over the Baseline with 6 PA-threads in `PM-2I`, while by 8.7x in `PM-NI`. We also observe that the difference between `PM-2I` and `PM-NI` narrows with more than 10 PA-threads. Previous experiment in Fig.8(a) shows that, in the uncontrolled case (i.e., Baseline), the MEMCPY throughput with a varying number of PM devices exhibit a larger gap in the waterfall region, while the difference in bottom region is relatively small.

2) Threshold T_{mc}: To evaluate the impact of the T_{mc} value on the performance, we compare the pre-defined T_{mc} value (`Def`), with $T_{mc}+0.1$ (`Inc`) and $T_{mc}-0.1$ (`Dec`), respectively. We set the write size for PM at 512B. Note that the threshold T_{mc} is designed to tradeoff between maintaining the DRAM performance and fully exploiting PM throughput. In general, a larger T_{mc} value always leads to more concurrent requests accessing PM (i,e., the Serving-Thread-count Ts) in PATS.



(a) MEMCPY throughput (b) PM throughput

Fig. 14. PM and DRAM throughput with varying T_S.

Fig.14(a) shows that `Dec` is able to improve the MEMCPY throughput by up to 8% over `Def` with a varying number of PA-threads. This is because `Dec` strictly limits the number of runnable PA-threads in PATS. However, the other side of the coin is that too few PA-threads may decrease the performance of PMs, especially for workloads with small request sizes (e.g., 512B in this experiment). Fig.14(b) shows that the write throughput of PM in `Dec` drops by up to 18% compared to `Def`. In this case, even more PA-threads from application cannot promote the PM overall throughput.

In contrast, `Inc` achieves a relatively small improvement in the PM throughput at the cost of notably decreased DRAM accessing performance. With 14 PA-threads, the throughput for MEMCPY drops by up to 19% in `Inc` compared to `Def`, while the PM write throughput increases by less than 3%. These results are consistent with the earlier observation that more PA-threads cause higher bandwidth contention, significantly reducing DRAM performance.

VI. Related Works

To the best of our knowledge, our work is the first to address and proposes solutions for the PM-access-induced bandwidth contention between DRAM and PM, with the following relevant issues studied in the literature.

Contention in multi-threaded execution environment. Prior to the advent of PM, existing works have explored performance issues stemming from resource contention in the critical paths when multiple threads access DRAM memory, including CPU cores [16], memory controllers [17], and memory channel bandwidth [18]. Appropriate analytical models were also constructed to quantify and predict contention behaviors [19]. However, none of them considers the DRAM-PM architecture where DRAM and PM are used simultaneously in the memory subsystem. Our findings reveal that in such an architecture accessing PM with multiple threads can heavily hurt the performance of the DRAM memory.

Previous works mitigate the memory contention among multiple application-threads by optimizing the scheduling policy within the memory controller, i.e., by prioritizing memory requests with static scheduling according to request-types [9], or dynamic scheduling to prevent starvation [20]. However, these request-based schemes operate on the execution critical path and introduce extra hardware cost and processing delay. In contrast, PATS is implemented in the PM-processing software layer without extra hardware modification. And importantly, PATS is not on the critical path of accessing DRAM.

PM characteristics. There have been some efforts on measuring the actual characteristics of the PM device, including testing its basic performance, analyzing available patterns for different applications and exploring its internal scheduling mechanisms [1]. However, none of them addresses the bandwidth contention problem arising from the joint use of PM and DRAM. Our study complements these prior studies by exploring these missing device characteristics of PM.

VII. Conclusion

We experimentally analyze the performance impact of severe bandwidth contention due to multiple threads accessing PM and DRAM simultaneously. We design a PM-Accessing Thread Scheduling mechanism with PMDK library that is guided by a contention model to mitigate the problem.

Acknowledgments

This work is supported by National key research and development program of China under Grant 2018YFA0701800, NSFC No.62172175, No.61821003, NSF CNS-2008835, and Alibaba Innovative Research (AIR) Program.

References

[1] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *FAST 2020*.
[2] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: reducing software overhead in file systems for persistent memory," in *SOSP 2019*.
[3] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for DRAM-NVM memory systems," in *USENIX ATC 2017*.
[4] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. Choi, "SLM-DB: single-level key-value store with persistent memory," in *FAST 2019*.
[5] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. G. Dreslinski, "Improving performance of flash based key-value stores using storage class memory as a volatile memory extension," in *USENIX ATC 2021*.
[6] J. Breitbart, S. Pickartz, S. Lankes, J. Weidendorfer, and A. Monti, "Dynamic co-scheduling driven by main memory bandwidth utilization," in *CLUSTER 2017*.
[7] A. Horvath and J. Slocum, "Mbw," 2020, https://github.com/raas/mbw.
[8] "memcached," 2020, http://memcached.org/.
[9] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. T. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *MICRO 2011*.
[10] I. Neal, G. Zuo, E. Shiple, T. A. Khan, Y. Kwon, S. Peter, and B. Kasikci, "Rethinking file mapping for persistent memory," in *FAST 21*.
[11] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *FAST 2016*.
[12] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, "Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules," in *MEMSYS 2019*.
[13] AXBOE, "Fio: Flexible i/o tester," 2020, https://github.com/axboe/fio.
[14] Intel, "Intel persistent memory programming," https://pmem.io/pmdk/.
[15] INTEL, "Lmbench," 2020, https://github.com/intel/lmbench.
[16] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled direct memory access: Isolating CPU and IO traffic by leveraging a dual-data-port DRAM," in *PACT 2015*.
[17] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *HPCA 2013*.
[18] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa, "Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead," in *HPCA 2014*.
[19] W. Wang, J. W. Davidson, and M. L. Soffa, "Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines," in *HPCA 2016*.
[20] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: providing performance predictability and improving fairness in shared main memory systems," in *HPCA 2013*.