Persistent Memory I/O-Aware Task Placement for Mitigating Resource Contention

Hyunwoo Ahn
Sungkyunkwan University
Suwon, Republic of Korea
ahw9925@skku.edu

Jongseok Kim
Sungkyunkwan University
Suwon, Republic of Korea
ks7sj@skku.edu

Euiseong Seo
Sungkyunkwan University
Suwon, Republic of Korea
euiseong@skku.edu

ABSTRACT

Direct access (DAX) file systems for persistent memory (PM) perform reads and writes through `load` and `store` instructions, respectively, bypassing the I/O path inside the operating system (OS) kernel. However, because of this, the OS is unable to differentiate PM I/O tasks from CPU-bound tasks, resulting in them being treated equally in task placement. PM I/O-oblivious task placement significantly impacts the DRAM access performance of co-located tasks due to severe resource contention on the memory controller, particularly with remote PM access. Moreover, such task placement fails to utilize on-chip idle resources from the stall cycles for PM I/O, impacting the efficacy of simultaneous multi-threading (SMT). We propose a PM I/O-aware task placement scheme that detects PM I/O activities and dynamically places tasks to mitigate the memory controller contention and to efficiently utilize the idle on-chip resources. In our evaluation, PM I/O-oblivious task placement caused FIO to run over five times slower and reduced SPEC CPU performance by more than three times compared to optimal placements. However, our proposed approach limited the average performance loss to just 3.3% across both workloads, with a maximum loss of only 4.9%.

CCS CONCEPTS

• **Software and its engineering** → **Scheduling**.

KEYWORDS

Operating Systems, Scheduling, Load Balancing, Persistent Memory, Direct Access File Systems

APSys '24, September 4–5, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-1105-3/24/09
https://doi.org/10.1145/3678015.3680482

ACM Reference Format:
Hyunwoo Ahn, Jongseok Kim, and Euiseong Seo. 2024. Persistent Memory I/O-Aware Task Placement for Mitigating Resource Contention. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '24), September 4–5, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678015.3680482

1 INTRODUCTION

Direct access (DAX) file systems [13, 14, 19] for persistent memory (PM) execute read and write operations using `load` and `store` instructions leveraging the direct byte-addressable characteristics of PM. This minimizes mode switching for I/O operations between user and kernel levels, bypassing the operating system (OS) kernel's I/O stack and eliminating cache management overhead. As a result, DAX file systems offer lower latency and higher bandwidth.

However, DAX file systems introduce unique challenges in terms of task distribution across cores. For conventional storage devices, I/O-bound tasks spend the majority of their time in a sleep state, waiting for I/O operations to complete. In contrast, CPU-bound tasks continuously occupy the CPU until the end of their allocated time slice. The OS kernel's load balancer often blends these tasks arbitrarily, due to the low load of I/O-bound tasks. However, this blending does not typically impact the performance of either task type.

In DAX file systems, tasks perform I/O operations without invoking system calls. This makes it challenging for the OS to distinguish them from CPU-bound tasks. Although PM I/O tasks spend most of their time in stalls for loads and stores, OS kernels consider them as CPU-bound tasks because they still occupy the processor during those stalls. The inability to distinguish between task types leads to PM I/O oblivious load balancing and can result in pathological cases.

In simultaneous multi-threading (SMT) processors, a single physical core functions as multiple logical cores, PM I/O tasks may be assigned to the same physical core. Similarly, CPU-bound tasks might be concentrated on a single physical core. These cases prevent the effective utilization of idle processor resources, which arise from stalls induced by PM I/O tasks. Furthermore, the concentration of PM I/O tasks on cores sharing a memory controller can lead to severe

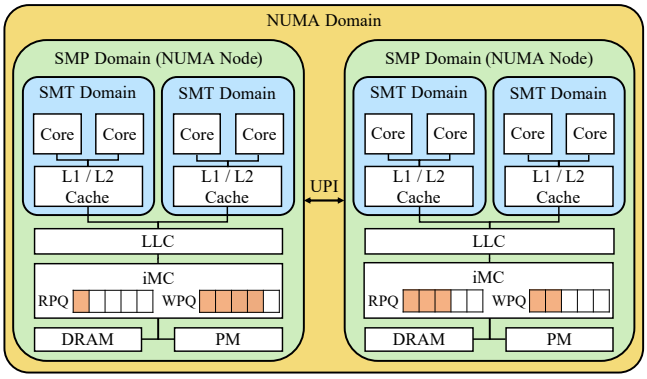


Figure 1: Scheduling domains in the Linux kernel.

memory controller contention, significantly degrading the DRAM access performance of concurrent CPU-bound tasks.

This paper presents an in-depth analysis of the resource contention caused by PM I/O-oblivious load balancing in the Linux kernel. Based on that, we propose a PM I/O-aware dynamic task placement approach. The proposed scheme includes a method for estimating a task's PM I/O load and an algorithm for distributing tasks that prevents the aforementioned issues and maximizes the overall performance while obeying the existing load balancing mechanism.

Although a few studies have proposed PM I/O throttling or scheduling schemes to mitigate the PM I/O-induced resource contention [10, 15, 17, 18], they did not tackle its fundamental cause. Moreover, to our knowledge, this is the first attempt to efficiently utilize stall cycles caused by PM I/O.

For evaluation, we implemented the proposed scheme in the Linux kernel. SPEC CPU2017 benchmark [8] was used as CPU-bound tasks while FIO [6] was used as PM I/O tasks.

2 BACKGROUND AND MOTIVATION

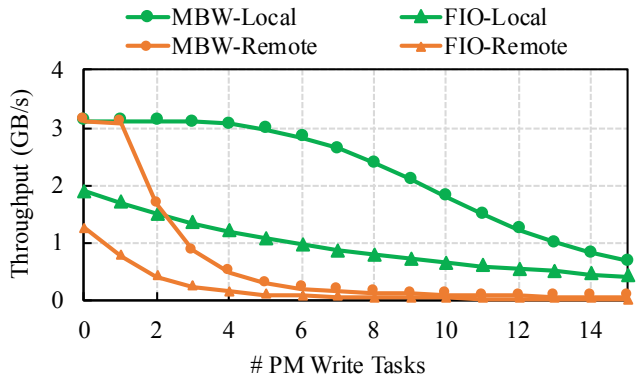
As shown in Figure 1, PM and DRAM are integrated into the system through an integrated memory controller (iMC) within the CPU chip. The iMC manages data flow between the CPU and memory bus, using its own read pending queue (RPQ) and write pending queue (WPQ) to handle requests. However, since the iMC manages both DRAM and PM, performance interference can occur between their requests. PM's slower access latency than DRAM can cause head-of-line (HOL) blockage in the iMC's RPQ and WPQ, delaying subsequent DRAM requests [17]. This blockage is more pronounced due to PM's asymmetric performance between read and write [20]. Therefore, this paper focuses on PM write operations, which are significantly slower than read.

In the Linux kernel, the scheduling domain is a group of logical processors where task migration is allowed, crucial for load balancing in multi-processor systems [9]. Figure 1 shows three Linux kernel scheduling domains: the simultaneous multi-threading (SMT) domain, which is a set of logical processors sharing a physical CPU core; the symmetric multiprocessing (SMP) domain, a set of processors sharing the last-level cache (LLC); and the non-uniform memory access (NUMA) domain, which consists of multiple SMP domains that have the same memory access latency. Each sub-domain within a scheduling domain is termed a *scheduling group*.

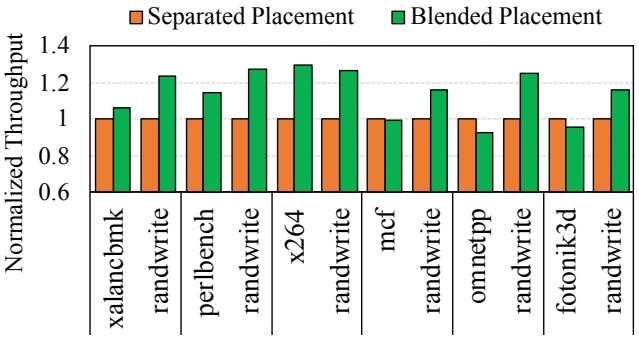
The Linux kernel's load balancer routinely migrates tasks from heavily loaded scheduling groups to less loaded ones within each scheduling domain, ensuring even distribution of CPU load across all scheduling groups. Load balancing begins at the lowest scheduling domain, distributing tasks among logical cores, then moves up to the SMP domain for physical core distribution, and finally performed at the highest-level NUMA domain to balance load between processor sockets.

We conducted experiments to examine the impact of the HOL blockage from PM write operations in an SMP domain and a NUMA domain on the DRAM access performance. For this, PM storage was constructed in a server described in Table 1 by interleaving six PM devices located in a single SMP domain. We used the FIO random write workload, which repetitively issues 4 KB write requests, as the PM write workload. The I/O engine of the FIO benchmark was set to *libpmem*, provided through the Persistent Memory Development Kit (PMDK)[5]. We also used MBW[3], a memory performance benchmark that copies a 10 GB array using `memcpy`, as a representative CPU-bound application that performs frequent memory accesses. To eliminate the effects of scheduling delays, the MBW task runs on a physical CPU core isolated from the FIO tasks. The MBW task is always located in the same SMP domain as the PM devices, while FIO writers may be located in another domain.

As shown in Figure 2a, when the FIO tasks and the MBW task were located on the same node as the PM devices, with results colored green, the PM write operations were processed as local writes within the same SMP domain. This configuration, as the number of write tasks increased, led to natural bandwidth contention and a steady, gradual decrease in task throughput. Conversely, when the FIO tasks were placed on a different NUMA node—results shown in orange—even a minimal remote PM write load caused a significant reduction in MBW throughput. Remote PM writes cause more severe DRAM interference than local writes because they take longer to issue, increasing HOL blockage in the WPQ. A CPU core must wait for credit allocation from the iMC to insert a write request into the WPQ where the target PM device is located. If no credits are available, the core waits for an existing request to be processed and the credit returned. Remote PM writes also hold up credits longer due to the additional time needed to traverse the interconnect network between NUMA nodes [7, 11, 16].



(a) MBW and FIO throughput according to varying number of concurrent FIO tasks.



(b) Throughput of SPEC CPU and FIO tasks according to their placement.

Figure 2: Impact of PM writes in an SMP domain and a NUMA domain (a), and in an SMT domain (b).

To evaluate the impact of PM writes on other tasks in the SMT domain, we conducted experiments using six workloads from the SPEC CPU2017 benchmark suite alongside FIO tasks. Each experiment ran two instances of a SPEC CPU2017 workload along with two FIO tasks repeatedly issuing 4KB random write requests to the PM. The four tasks were executed under different conditions across two physical cores, each housing two logical cores. The baseline configuration had two SPEC CPU tasks on one physical core and two FIO tasks on another. This setup was compared to one where a SPEC CPU task and an FIO task shared the same physical core, distributing the four tasks between two physical cores.

In Figure 2b, co-scheduling a CPU-bound task with a PM random write task in the same SMT domain improved performance by an average of 14.3% compared to scheduling similar tasks together. This enhancement stems from reduced resource contention; CPU-bound tasks often compete for shared resources like the ALU and FPU, lowering SMT efficiency. By pairing a CPU-bound task with a PM write task, the former can better utilize these resources during the latter's long stall cycles.

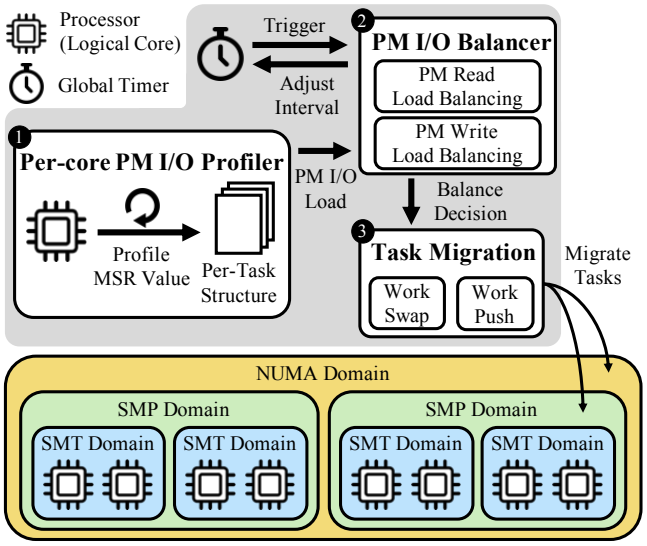


Figure 3: Design overview of our approach.

3 OUR APPROACH

3.1 Design Overview

As previously analyzed, the impact of PM I/O tasks on the performance of other tasks varies across scheduling domains. In the NUMA and SMP domains, it is crucial to evenly distribute PM I/O loads among scheduling groups to reduce HOL blocking in the iMC's WPQ. In the SMT domain, strategically pairing PM I/O with CPU-bound tasks within the same group can significantly enhance performance. The proposed PM I/O-aware task placement approach, illustrated in Figure 3, strategically manages the placement of PM I/O tasks to optimize performance across these domains.

The flow of our approach is as follows: ❶ A per-core PM I/O profiler periodically analyzes the PM read and write load of each task on each core. ❷ The PM I/O load balancer then redistributes these loads among scheduling groups in the top two scheduling domain layers—NUMA and SMP—using ❸ work swap and work push task migration mechanisms. This strategy ensures an even distribution of PM I/O loads across the NUMA and SMP domains, and collocates PM I/O with CPU-bound tasks on sibling cores within the SMT domain.

The higher the scheduling domain, the greater the task migration overhead between scheduling groups becomes. Therefore, the Linux kernel's load balancer performs load balancing from the lowest to the highest domains. However, the higher the scheduling domain, the more severe the performance degradation from resource contention due to PM I/O tasks. As a result, our approach prioritizes PM I/O

task placement at higher scheduling domains, employing a top-down placement approach that is the opposite of the Linux kernel's load balancer. During this, our approach also considers the CPU loads of scheduling groups to prevent undermining the effectiveness of the CPU load balancer.

3.2 Per-Core PM I/O Profiling

Our approach places a PM I/O profiler on each CPU core to track each task's PM I/O activity. The per-core PM I/O profiler counts PM reads and writes using off-core performance monitoring event counters [12] every million clock cycles.

The `ocr.all_data_rd.pmm_hit_local_pmm.any_ snoop` event counter is useful for identifying the number of read operations each task performs on the local PM of its NUMA node [4]. However, current commodity CPUs lack monitoring counters for the number of local PM write operations or cross-NUMA domain remote I/O [21]. To estimate the number of local PM writes a task performs, we utilize the `ocr.all_rfo.pmm_hit_local_pmm.any_snoop` event counter, which tracks reads-for-ownership (RFOs) needed to read data from the local PM for modification. When PM devices across all NUMA nodes are used in an interleaved fashion, a nearly equal distribution of read and write operations across all PMs is expected. Thus, the count of local I/O operations interfacing with the PM should closely mirror that of remote I/O operations.

Directly using the PM I/O amount profiled through performance counters for task placement can lead to frequent task migrations between scheduling groups, especially for tasks performing a lot of PM I/O in a short period. To prevent this, we apply the exponential moving average (EMA) to calculate the task's PM I/O load as in Eq. 1.

$$M_{i+1} = k\alpha(P_i/cycle) + (1-k)M_i \quad (0 \le k \le 1) \qquad (1)$$

In Eq.1, M_i represents the PM I/O load for the i-th profiling period, $P_i/cycle$ denotes the amount of PM I/O work per cycle measured, α is a scaling factor used to avoid excessively small M_i values when P_i is small compared to $cycle$. k signifies the weighting multiplier in the EMA. In this study, based on a profiling period set at one million cycles, we used values of 100,000 for α and 0.5 for k.

Taking into account the asymmetric read and write performance of PM [20], we calculate the PM I/O read load and write load of a task, separately. The derived PM read load and PM write load values are stored in the OS's task structure.

3.3 PM I/O-Aware Dynamic Task Placement

As stated, remote PM access can significantly impact the performance of tasks running on the NUMA node with the target PM device. In an environment where all PMs of every NUMA node are interleaved, a PM I/O task accesses each NUMA node equally. Hence, evenly distributing PM I/O tasks across all NUMA nodes helps prevent the concentration of *remote* PM accesses on a specific iMC. Also, to maximize the SMT effect, it is important to evenly distribute PM I/O tasks between scheduling groups within the SMP domain, enabling co-scheduling of CPU-bound tasks and PM I/O tasks on sibling logical cores sharing a physical core.

Due to PM's asymmetric read and write performance, the PM write load should be regarded as more important than the PM read load. Hence, the proposed task placement scheme first decides task placement considering the PM read load, then finalizes the task placement based on the PM write load to override the initial decisions made from the PM read load.

Task placement considering the PM read load is carried out through the following steps. First, starting from the top-level scheduling domain, it calculates the sum of the PM read loads of the tasks belonging to each scheduling group that constitutes the scheduling domain. Then, it computes Δ, which is the difference in PM read load between SG_{pmax}, the scheduling group with the largest sum of PM read load, and SG_{pmin}, the scheduling group with the smallest sum.

Next, it performs work swaps iteratively for all task pairs within SG_{pmax} and SG_{pmin} to reduce Δ. A *work swap* maintains PM read load balance between the two scheduling groups by exchanging a high PM read load task from SG_{pmax} with a low PM read load task from SG_{pmin}. Work swaps occur only if they reduce Δ by over 10% and if the CPU load difference between the tasks is within `imbalance_pct`, the criterion used by the current Linux load balancer to judge load imbalance among scheduling groups. The default value of `imbalance_pct` is 10% for the SMT domain and 17% for the other two top-level domains. If no tasks meet these conditions, no work swap occurs. The process repeats until no further swaps are possible in an iteration.

Subsequently, if it is impossible to perform a work swap due to a large CPU load difference between tasks, the occurrence of idle cores, the processor affinity of the task, etc., the proposed scheme attempts to move additional PM read load from SG_{pmax} to SG_{pmin} through a work push. A *work push* involves moving a PM read task unidirectionally from SG_{pmax} to SG_{pmin}. Therefore, to prevent potential CPU load imbalance, a work push moves a PM read task only if the CPU load difference between the two scheduling groups after the task movement is within `imbalance_pct`, based on the scheduling group with the larger CPU load.

The proposed scheme repeats this process until there are no more tasks that can be moved by a work push, at which point the task migration considering read load is terminated. Then, the same procedure will be initiated for the write load within the same domain before moving to the lower domain.

Table 1: Server configurations used for evaluation.

Server	Dell PowerEdge R740
CPU	Intel Xeon Gold 5218 × 2 Sockets 32 Physical Cores (64 Logical Cores) @ 2.30 GHz
RAM	16GB × 12
PM	Intel Optane DC PM 128GB × 12
OS	Linux 6.04 (using *ext4 DAX* [1] file system)

4 EVALUATION

We implemented the proposed approach in the Linux kernel 6.04 and evaluated it using the system described in Table 1. For CPU-bound workloads, we used six SPEC CPU2017 benchmarks with diverse memory bandwidth requirements. For PM I/O workload, we used FIO random write.

4.1 Analysis in SMT Domain

To understand the impact of our approach on the SMT domain, we conducted a performance analysis under the same conditions as the experiment depicted in Figure 2b. In this setup, high CPU loads of both SPEC CPU and FIO tasks make the Linux kernel randomly assign tasks to cores. Thus, a SPEC CPU task and a FIO task may share a physical core's two sibling logical cores, termed *Linux-Blended*, or may be allocated to separate physical cores, termed *Linux-Separated*.

Figure 4 presents the throughput of each workload in an environment where each of the six types of SPEC CPU workloads is executed with FIO random write tasks. The throughput of SPEC CPU workloads was measured as the reciprocal of the completion time. The results were normalized to the ones obtained under Linux-Blended.

Linux-Blended showed an average throughput improvement of 6.4% for SPEC CPU workloads and 22.3% for FIO random writes compared to Linux-Separated. This improvement was achieved by minimizing resource competition,

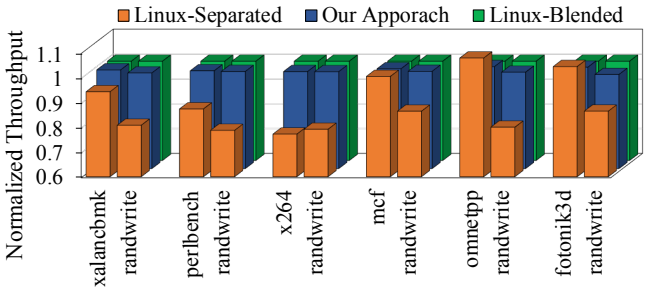


Figure 4: Normalized throughput when executing SPEC CPU tasks with FIO random write tasks in two physical cores each having two logical cores.

Table 2: Normalized throughput of SPEC CPU with FIO random write tasks placed in multiple NUMA nodes.

	Placement	2 Tasks Each		4 Tasks Each		8 Tasks Each	
		SPEC	FIO	SPEC	FIO	SPEC	FIO
xalancbmk	Separated	0.987	0.843	0.900	0.379	0.753	0.195
	Ours	0.995	0.999	0.986	0.982	0.962	0.961
perlbench	Separated	0.997	0.856	0.866	0.404	0.483	0.163
	Ours	0.999	0.993	0.987	0.992	0.968	0.984
x264	Separated	0.967	0.850	0.764	0.381	0.370	0.171
	Ours	0.996	0.997	0.980	0.984	0.975	0.944
mcf	Separated	0.875	0.824	0.386	0.397	0.065	0.194
	Ours	0.997	0.994	0.996	0.993	0.990	0.974
omnetpp	Separated	0.677	0.809	0.262	0.397	0.058	0.199
	Ours	0.973	0.998	0.990	0.992	0.958	0.974
fotonik3d	Separated	0.665	0.744	0.201	0.453	0.027	0.277
	Ours	0.993	0.979	0.994	0.994	0.948	0.964

such as ALU and FPU, within the same SMT domain, and by alleviating L1 and L2 cache pressure, respectively.

However, in Linux-Blended, mcf, omnetpp, and fotonik3d showed similar or even worse throughput than Linux-Separated. These workloads consumed about 78 times more memory bandwidth on average compared to the other three workloads. As a result, the performance loss from high L1 and L2 cache corruption by FIO outweighed the gains from reduced resource competition, resulting in an overall throughput decline. Despite this, the FIO tasks saw significant improvement in throughput, so it can be concluded that Linux-Blended is superior to Linux-Separated.

The Linux kernel randomly decides configuration between Linux-Blended and Linux-Separated, leading to poor and uncertain outcomes. Our approach chose Linux-Blended placement, yielding nearly identical results in all cases with only a 0.6% average difference. This minor difference comes from PM I/O profiling and task migration overhead.

4.2 Analysis in SMP and NUMA Domains

To assess our approach's effectiveness in alleviating WPQ contention in SMP and NUMA domains, we conducted experiments using both NUMA nodes of the server. The PM devices were installed in both NUMA nodes and configured to the interleaved mode. Only one task was run on each physical core to avoid SMT domain resource contention. As mentioned in Section 4.1, the Linux kernel randomly places SPEC CPU and FIO tasks, resulting in tasks of the same type being clustered in the same NUMA node (*Linux-Separated*), or evenly distributed across all NUMA nodes (*Linux-Blended*).

Table 2 shows the normalized throughput of each of the six SPEC CPU workloads when they were executed with FIO random write tasks. The more the PM write tasks, the more severe WPQ HOL blockage occurs. To analyze the impact of the varying degrees of WPQ blockage, each number of SPEC CPU tasks and FIO tasks varied from two to four to eight.

In the experiments with two SPEC CPU tasks and two FIO tasks in Table 2, our approach achieved an average 18.5% higher throughput for SPEC CPU tasks and 21.2% higher throughput for FIO tasks compared to Linux-Separated. As discussed in Section 2, Linux-Blended's task placement has a performance advantage by preventing FIO tasks from clustering in one SMP domain, thereby evenly distributing remote PM writes across all NUMA nodes to minimize HOL blockage. Our approach achieves this performance improvement by dynamically placing tasks like Linux-Blended.

Our approach demonstrated improvement increase in SPEC CPU throughput compared to Linux-Separated as memory bandwidth requirements increased. For fotonik3d, which had the highest memory bandwidth usage, our approach achieved a 49.4% higher throughput compared to Linux-Separated. This trend occurred because workloads requiring high memory bandwidth are more affected by the WPQ HOL blockage caused by PM write requests. FIO random write tasks also showed the same trend. Our approach showed a 31.5% higher throughput for fotonik3d over Linux-Separated.

As the number of tasks grew, WPQ contention between workloads increased, leading to more severe WPQ blockage. Accordingly, the throughput improvement of our approach compared to Linux-Separated also grew larger with more concurrent tasks. As shown in Table 2, in the 4-tasks experiment, our approach showed about 5 times higher throughput for fotonik3d and about 2.2 times higher throughput for the FIO tasks compared to Linux-Separated. In the experiment with eight tasks, our approach improved the throughput of fotonik3d by 35.2 times and FIO by 3.5 times.

5 CONCLUSION

I/O to DAX file systems cannot be monitored by the OS; therefore, the OS cannot distinguish PM I/O tasks from CPU-bound tasks during the scheduling process. As a result, the OS blindly places PM I/O tasks with other tasks, potentially leading to significant performance degradation.

Our research presents a dynamic task placement scheme for PM I/O tasks, which optimizes task placement across CPU cores based on PM I/O activities. Our evaluation showed that the proposed approach consistently maintains near-optimal performance, with an average performance degradation of only 3.3% due to PM I/O and CPU task interference, and a maximum of 4.9% compared to the optimal placements. These results confirm that our approach effectively addresses issues with PM I/O-oblivious task placement, actively correcting existing OSs' suboptimal placements.

Additionally, we believe that our approach is applicable to systems equipped with upcoming Compute Express Link (CXL) devices [2], which allow tasks to directly access storage devices, bypassing the OS kernel, similar to PM modules.

6 ACKNOWLEDGMENTS

This research was supported by the Ministry of Science and ICT (MSIT), Korea, under the Information Technology Research Center (ITRC) support program (IITP-2024-RS-2023-00258649) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP), and by the National Research Foundation of Korea (NRF) grant (RS-2023-00321688).

REFERENCES

[1] 2019. *Direct Access for files.* https://www.kernel.org/doc/Documentation/filesystems/dax.txt
[2] 2020. *Compute Express Link Specification.* https://computeexpresslink.org/cxl-specification
[3] 2020. MBW: memory bandwidth benchmark. https://manpages.ubuntu.com/manpages/kinetic/en/man1/mbw.1.html.
[4] 2023. 2nd Generation Intel Xeon Processor Scalable Family based on Cascade Lake product. https://perfmon-events.intel.com/cascadelake_server.html.
[5] 2023. PMDK: Persistent memory development kit. https://pmem.io/pmdk/.
[6] Jens Axboe. 2014. Fio-flexible IO tester. http://freecode.com/projects/fio.
[7] Kuljit Singh Bains, Raj Ramanujan, Wesley Queen, and Liyong Wang. 2021. Write credits management for non-volatile memory. US Patent 10,996,888.
[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)*. 41–42.
[9] Jonathan Corbet. 2014. Scheduling domains. https://lwn.net/Articles/80911/.
[10] Satoshi Imamura and Eiji Yoshida. 2020. FairHym: Improving interprocess fairness on hybrid memory systems. In *Proceedings of the 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.
[11] Intel. 2017. Intel Xeon Processor Scalable Memory Family Uncore performance monitoring reference manual. (2017), 90–111.
[12] Intel. 2019. Intel 64 and IA-32 architectures software developer's manual. Volume 3: System Programming Guide. (2019).
[13] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. 804–818.
[14] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 494–508.
[15] Jinyoung Oh and Youngjin Kwon. 2021. Persistent memory aware performance isolation with dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. 97–105.
[16] Saravanan Sethuraman, Sumantra Sarkar, Karthikeyan Natarajan, Tathagato Bose, and Adam J McPadden. 2021. Dynamic write credit buffer management of non-volatile dual inline memory module. US Patent 10,901,657.
[17] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, and Yuanyuan Dong. 2022. PATS: taming bandwidth contention between persistent and

dynamic memories. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 885–890.

[18] Yongfeng Wang, Yinjin Fu, Yubo Liu, Zhiguang Chen, and Nong Xiao. 2022. Characterizing and optimizing hybrid DRAM-PM main memory system with application awareness. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 879–884.

[19] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST)*. 323–338.

[20] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 169–182.

[21] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. 2022. MT^2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*. 199–216.