Analyzing and Improving CPU and Energy Efficiency of PM File Systems

Lukas Werling
Karlsruhe Institute of Technology

Yussuf Khalil
Karlsruhe Institute of Technology

Peter Maucher
Karlsruhe Institute of Technology

Thorsten Gröninger
Karlsruhe Institute of Technology

Frank Bellosa
Karlsruhe Institute of Technology

Abstract

Persistent memory (PM) provides advantages such as direct access from the CPU at low latency, but it turned out that its performance is sensitive to certain access patterns. Particularly, the access latency increases when the PM is under parallel load. This leads not only to degraded PM performance, but slows down the entire system and wastes energy due to busy, but stalling CPUs. We propose an efficiency metric for measuring the CPU cost of a PM storage system. We evaluate our metric and the power consumption of existing file systems and explore PM-optimized memory copy routines to improve their efficiency. Finally, we propose an alternative system design where access to PM is mediated by an FPGA while allowing selective user space access to files.

ACM Reference Format:
Lukas Werling, Yussuf Khalil, Peter Maucher, Thorsten Gröninger, and Frank Bellosa. 2023. Analyzing and Improving CPU and Energy Efficiency of PM File Systems. In *1st Workshop on Disruptive Memory Systems (DIMES '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609308.3625265

1 Introduction

Persistent memory (PM) is a storage-class memory that offers advantages such as low latency and high bandwidth, but also introduces challenges for efficient storage stacks. In particular, recent studies have extensively analyzed the performance of Intel Optane PM [21, 22, 24]. The primary challenge is the management of concurrent access. Multiple parallel readers or writers are necessary to achieve maximal performance [24]. However, the total bandwidth—especially for writes—quickly deteriorates if concurrency increases beyond that. This issue is pronounced for write accesses, but can also be observed for read accesses at very high thread counts [27]. We limit our analysis to PM writes, but we expect our techniques to transfer to reads as well.

Parallel CPU access is problematic for PM as any delays lead to CPU stalls. Thus, accessing PM with high concurrency can do more than just reducing bandwidth: The overall system performance can be hurt by wasting CPU time, preventing useful work if available or low core power states.

The operating system commonly provides access to PM via file systems. For this work, we do not consider direct access (DAX) since it does not allow mediation by the operating system. A wide variety of new PM file systems have been proposed [8, 13, 14, 16, 23, 25–27]. A common goal is the reduction of static overhead such as from system calls [14]. However, we find that analysis of the overall CPU and energy efficiency is lacking. Since CPU utilization changes drastically with concurrency and storage performance, it is not obvious how to compare the CPU efficiency of PM file systems. We propose an easy-to-measure metric for assessing the CPU cost of a file system: CPU seconds per accessed GiB. We evaluate this metric as well as power measurements for several PM file systems, including ext4, NOVA [23], and OdinFS [27] and show a large potential for improvement.

At the core of every PM file system is a routine for copying memory between DRAM and PM. Aside from small transfers of file system metadata, this routine is performance-critical for every read and write since it copies data between user space buffers and file system-managed PM. We explore replacements for this copy routine that aim to follow PM best practices to reduce the CPU cost. By working at this level, we can integrate into existing PM file systems with low effort.

We finally propose an alternate system design where an FPGA acts as a mediator to PM. With that approach, the CPU merely submits copy tasks to the FPGA via ring buffers in system memory and is thereby freed from time-consuming copy operations. In contrast to traditional NVMe devices, our design can also support direct user space access to files similar to DAX. An FPGA-side memory protection unit ensures that processes can only access blocks over their command buffers that belong to their mapped files.

DIMES '23, October 23, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0300-3/23/10…$15.00
https://doi.org/10.1145/3609308.3625265

2 Background and Related Work

In this section, we introduce details about Intel Optane, which is the PM we target in our evaluation. We then examine PM file systems regarding their implementation of Optane best practices.

2.1 Intel Optane Performance

Intel Optane DC Persistent Memory is currently the only widely-available implementation of PM. Its performance turned out to be complicated and dependent on a lot of variables, including choice of instructions, random access size, and parallelism [12]. Further analysis suggested that a complex buffering scheme within the Optane DIMM is responsible for its behavior [21, 22]. We observe steadily decreasing bandwidth for writer threads accessing an Optane DIMM with increasing parallelism (see *devdax* in Figure 1). In accordance with previous works [24], we thus stress the importance of managing concurrency and NUMA scheduling when accessing PM. In the following, we propose mechanisms for file systems to deal with these issues. We focus on write accesses in this paper, but similar considerations also apply to read accesses at higher levels of concurrency.

First analysis of CXL memory devices suggests that these devices are at risk for similar parallelism issues [19].

2.2 PM File Systems

Concurrency control and NUMA awareness are uncommon features that are not present in most PM file systems, including PMFS [8], NOVA [23], and WineFS [13]. OdinFS [27] and SPMFS [25] are two more recent file systems that manage concurrent PM accesses by delegating writes to a thread pool. Both file systems keep the write bandwidth stable with increasing parallelism. OdinFS additionally delegates remote NUMA accesses to a local thread and can delegate reads at high thread counts. We compare OdinFS delegation to our copy routines in Section 7.2.

Energy efficiency is still a rare consideration for storage systems. Sundar et al. investigated energy efficiency of storage interfaces [20]. To our knowledge, there is no such analysis for PM file systems.

2.3 Hardware Offloading

Intel I/OAT [17] is a DMA device designed for offloading memory-to-memory copies. With Fastmove [18], Su et al. analyze and improve the use of Intel I/OAT in PM file systems. They focus on reducing latency and do not evaluate CPU or energy efficiency. We include measurements for a basic DMA offloading mechanism in our evaluation, but were unable to run Fastmove on our system due to software bugs.

3 PM File System Efficiency

As we have discussed above, PM is very sensitive regarding the access strategy. Consequently, PM file systems need to consider PM best practices to avoid costly CPU stalls. We find that there is a lack of comparison between file systems for CPU and energy efficiency. We argue that this is partly the case because there is no established metric so far for determining the efficiency of a PM file system.

We aim to close this gap by proposing a novel metric and measuring methodology for the CPU cost of a file system. It is independent of the bandwidth, easy to measure, and can be applied to arbitrary file systems without adjustments. We define the CPU costs for a write operation as follows:

$$\text{CPU cost} = \frac{\text{CPU time}}{\text{amount of written data}}$$

We measure it in seconds per Gibibyte (s/GiB). The CPU time is the sum of seconds each CPU was active, as reported by the scheduler (`/proc/stat` in Linux). We use fio [5] as benchmarking tool with a random write benchmark.

We use CPU time as the numerator, as compared to, e.g., CPU cycles, for two reasons: First, CPU time is independent of effects like dynamic frequency changes, and effectively measures actual progress as experienced by an actual application. Second, CPU time is more dependent on I/O hardware, since the I/O performance does not change with the CPU frequency. Every CPU that can sustain the bandwidth of the I/O device will experience similar scores in this metric.

By measuring CPU time, our metric gives a good indication of how a file system affects other CPU-bound processes. Although CPU activity often dominates the power budget, the contribution of memory and I/O devices should not be neglected. We thus measure the average power at the wall plug during our benchmarks and subtract the idle power usage of the system. From this value, we derive the *energy cost* of the file system, measured in Joules per GiB.

Our measurement methodology is as follows. (1) Choose fio parameters, including runtime t, block size, number of jobs. (2) Measure idle system load and power during runtime t. Verify that the load is low. (3) Set up the file system under test and allocate the fio test files. (4) While monitoring power, run the fio benchmark. Measure load by reading `/proc/stat` before and after running fio. We do not filter system load by process to avoid missing activity outside of the process context, for example in kernel threads.

3.1 File System Analysis

We measure power and CPU cost of four existing PM file systems with our methodology. We present the results here to motivate the following sections. We perform additional evaluation of the metric in Section 7.1.

Figure 1 shows the results. The graph also includes measurements for ext4 on an NVMe drive (Micron 7300 Pro 1 TB) and for direct user space access to PM (devdax) as a comparison. Our bandwidth measurements show that most of the file systems (ext4, NOVA [23], WineFS [13]) do not implement best practices for accessing PM. For these file systems, the total bandwidth decreases as more jobs access the PM, especially for accesses from a remote NUMA node. This has a

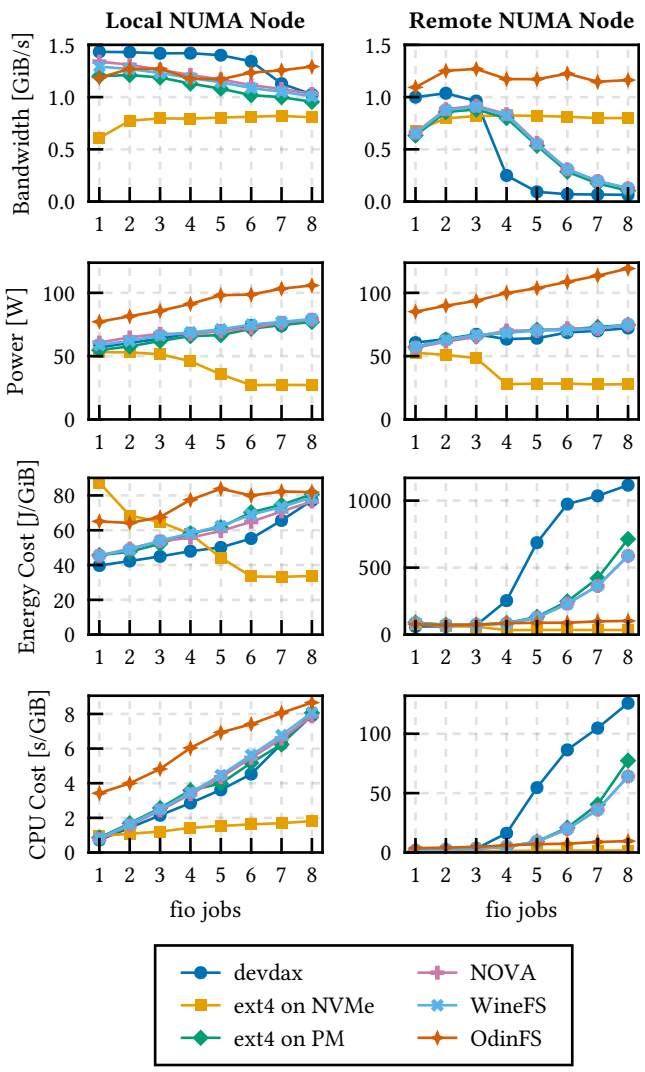

Figure 1. Evaluation of our metric for several file systems and for direct PM access (devdax). The write benchmark runs either from a CPU with the PM attached or from a different NUMA node.

dramatic effect on the CPU cost as well as energy efficiency: With each additional job, the CPU spends more time and energy writing to PM while hurting the overall bandwidth.

For OdinFS [27], we can see that its delegation to writer threads running on a local NUMA node successfully keeps the bandwidth at a steady level for both local and remote accesses. However, this mechanism is not for free. At two jobs, OdinFS spends more than double the CPU time and around 30 % more energy per GiB for the write operation than other PM file systems. Moreover, the CPU cost also rises with more jobs, making OdinFS only cheaper than the other file systems for parallel remote accesses.

We argue that ext4 on NVMe shows close to optimal behavior. The bandwidth stays roughly constant with higher

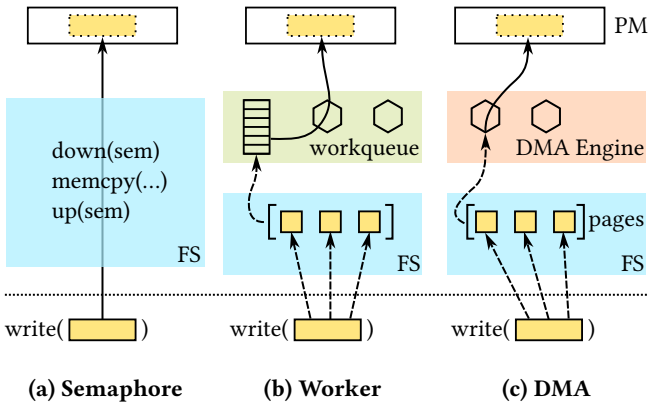

Figure 2. Variants for writing to PM efficiently.

numbers of jobs for both local and remote accesses. Even though the SSD performs the memory copy operation with DMA, the CPU costs from managing these transfers starts out slightly higher than for the PM file systems. However, the cost does not rise much with parallel jobs.

Our measurements for NVMe also show an important limitation of "wall plug" power measurements. Since we measure the whole system, we capture a complex interaction between devices with different power management strategies. We can observe a decrease in power with more fio jobs, suggesting that the NVMe device needs less energy with higher load. We assume that this anomaly results from CPU scheduling in our particular benchmarking scenario, but this is hard to prove without more fine granular power sensors. We see similar behavior for "Semaphore" in Figure 4.

In the following, we explore ways to decrease these CPU costs, bringing them closer to the level of NVMe.

4 Efficient PM Copy

As we have seen, most PM file systems do not handle parallel write accesses well. Although OdinFS's delegation [27] keeps the bandwidth stable, it has a non-trivial CPU and energy cost. Our goal is to provide a minimal API for efficient PM access that can be easily fitted onto existing PM file systems.

We analyzed CPU costs of PM file systems with perf's stack sampling. A key insight for the design of our API is that virtually all performance-critical writes in a file system are simple memory copies from user space buffers to PM. Our API is thus a single function `ep_write_pmem()` that copies memory from a source to a destination buffer—identical to `memcpy()`. The function can detect the location of the PM with the address of the destination buffer and can then take measures to prevent PM overload. Internally, we provide three variants for accessing the PM, as illustrated in Figure 2.

Semaphore. First, we protect the existing `memcpy()` routine with a semaphore, limiting parallel accesses to the PM. The semaphore thus prevents a decrease in bandwidth and allows other tasks to run. However, it cannot improve

the performance of PM accesses from remote NUMA nodes. Based on our analysis shown in Figure 1, two parallel PM writes to a single Optane DIMM is a reasonable compromise between bandwidth and CPU cost.

Workqueue. Second, we implement a mode similar to the delegation threads in OdinFS [27]. For each PM area, we introduce a workqueue with a limited amount of worker threads pinned to the respective NUMA node. A call to `ep_write_pmem()` appends the copy request to the workqueue and sleeps until the request is completed.

This approach can mitigate both a decrease in bandwidth as well as inefficient remote PM accesses: Parallel accesses are limited with the amount of worker threads and the worker threads always perform local PM accesses.

DMA. Finally, we implement a mode that offloads PM copies to an Intel I/OAT DMA device (see §2.3). We create DMA mappings for the pinned source and destination buffers and enqueue a DMA copy operation. We limit the amount of DMA channels to avoid parallel PM writes. As we will see in the evaluation (§7.2), Intel I/OAT cannot satisfy the full PM bandwidth on our system. Newer Intel CPU generations include improved offloading hardware [11, 19], which might make this approach viable in the future.

5 FPGA-Managed PM

As an alternative design for PM-equipped systems, we propose using periphery hardware to manage PM accesses. Maintaining low access latency is a key goal in such a scenario, albeit difficult to achieve with off-the-shelf hardware. With a GPU, for example, all data in DRAM↔PM copy operations would need to travel the system's PCIe bus twice: once for reading and again for writing. We use an FPGA instead that has the PM directly attached via a DDR-T interface. The CPU only communicates indirectly with the PM by submitting copy commands to the FPGA via ring buffers in DRAM, which the FPGA then executes asynchronously. The buffer size may be configured arbitrarily, allowing tuning to the application's burst size to avoid running full. This approach entirely prevents PM-related stalls and minimizes the CPU effort for PM operations. Our implementation currently polls for completion, but could support interrupts in the future.

NVMe generally provides similar functionality, but is incapable of providing DAX-style file access to user applications as it entirely relies on the operating system to enable access protection. We solve this challenge by leveraging SR-IOV to multiplex our hardware and provide a device-side memory protection unit (MPU). Each SR-IOV virtual function (VF) provides its own pair of read/write command buffers. The MPU isolates VFs from each other via range-based access checks. More precisely, for each VF a number of accessible address ranges with 2 MiB granularity may be defined. We consider this enough for typical PM file system use cases with few active processes that work on large files.

The operating system kernel is merely involved for a) handing out VFs to user space processes which they may then configure to their needs (e.g., regarding command buffer size), and b) communicating block range permissions to the FPGA. We currently require the kernel to establish DMA address mappings for user space pages in the IOMMU. We plan to leverage the Shared Virtual Memory (SVM) capability [2] in recent Intel processors in the future to allow the FPGA to access a process's memory without further kernel involvement. We achieve lock-free parallelism by employing one VF per thread that makes use of PM.

To support file system accesses via standard OS interfaces, we further implemented a Linux kernel block device based on blk_mq [6]. Using the block device, off-the-shelf file systems such as ext4 may be used atop our design.

Discussion. Moneta-D [7] is a previous approach that implemented hardware-based access checks for storage to allow direct user space access. However, their implementation fails to minimize latency as they use complex tree lookups for access checks with a stated average latency of 96 ns and a worst-case latency of 180 ns. Our implementation, in contrast, allows storing all ranges in on-die SRAMs and is able to deterministically perform access checks within one single clock cycle, i.e., 4 ns in our case (250 MHz). Further, our SR-IOV-based design can easily be extended for multi-tenant virtualization by adding a secondary access protection layer in the hardware. Then, shares of FPGA-managed PM may be handed out to virtual machines independently and without requiring further mediation by the hypervisor.

6 Implementation

We implemented two Linux kernel modules for the file system interface and the FPGA driver. Our source code is available at https://github.com/KIT-OSGroup/efficient-pm.

For the implementation of the file system interface, we make use of existing kernel interfaces as much as possible. The Semaphore variant uses a standard Linux `struct semaphore`. We implement the Workqueue variant with an unbound Linux workqueue [10] that is pinned to specific CPU cores. Finally, the DMA variant is based on the DMAEngine framework [1]. For each transfer, it selects a DMA channel and enqueues one DMA operation per page. The thread then blocks until all transfers are completed. We adapt NOVA to make use of our file system interface. For that, we had to modify a single line of code within NOVA.

We implemented the FPGA-managed PM approach using an Intel Stratix 10 DX FPGA that is connected to the base system via a PCIe 3.0 x16 interface. The kernel driver and the corresponding user space library encompass a total of about 1500 lines of C code. Our implementation supports a total of 512 VFs.

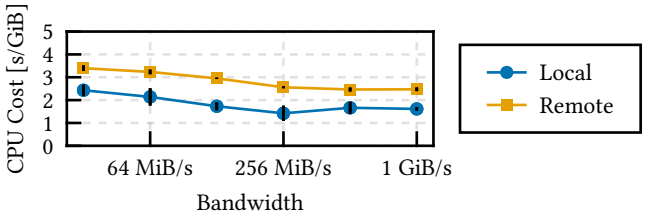

Figure 3. CPU cost evaluated for ext4 on PM running fio with two jobs at varying target bandwidths. Mean out of three runs, error bars show standard deviation.

7 Evaluation

We have introduced the design and implementation of a metric for assessing the CPU efficiency of PM file systems, three variants for improving the CPU efficiency, and FPGA-mediated PM. In our evaluation, we attempt to answer the following questions:

- Is our proposed metric for CPU cost bandwidth-independent? (§7.1)
- Do our variants reduce the CPU cost of PM file systems? Where does the CPU cost come from? (§7.2)
- What is the performance and CPU cost of our FPGA-mediated PM? (§7.3)

We perform our evaluation on a dual-socket system with eight-core Intel Xeon Silver 4215 CPUs and 128 GB of DRAM. We evaluate performance to a single first-generation Intel Optane DIMM with a capacity of 128 GB. We keep SMT enabled and the CPU power governor (intel_pstate) set to "powersave" so that our setup is close to a real system.

We use fio [5] as our main benchmarking tool. If not noted otherwise, we configure fio to do random sync writes to 100 MiB large files per job with a block size of 128 KiB. We let fio run for 30 seconds and observe steady bandwidth and power measurements with low standard deviation.

7.1 PM Efficiency Metric

A major goal of our CPU cost metric is independence from the device bandwidth. Plain CPU time is not a good metric for evaluating I/O-bound software since the CPU utilization naturally rises with faster I/O. To evaluate our metric's bandwidth independence, we set fio to write to an ext4 file system on PM with target bandwidths ranging from 32 MiB/s to 1 GiB/s. Based on our results in Figure 1, we configure two fio jobs so that we can evaluate the full range of bandwidths.

Figure 3 shows the results. We can see that the CPU cost decreases slightly from 32 MiB/s up to 256 MiB/s and stays relatively constant from there. We assume that static overhead from fio leads to this result, since such overhead has more weight on the metric for small bandwidths and consequently a lower amount of written bytes.

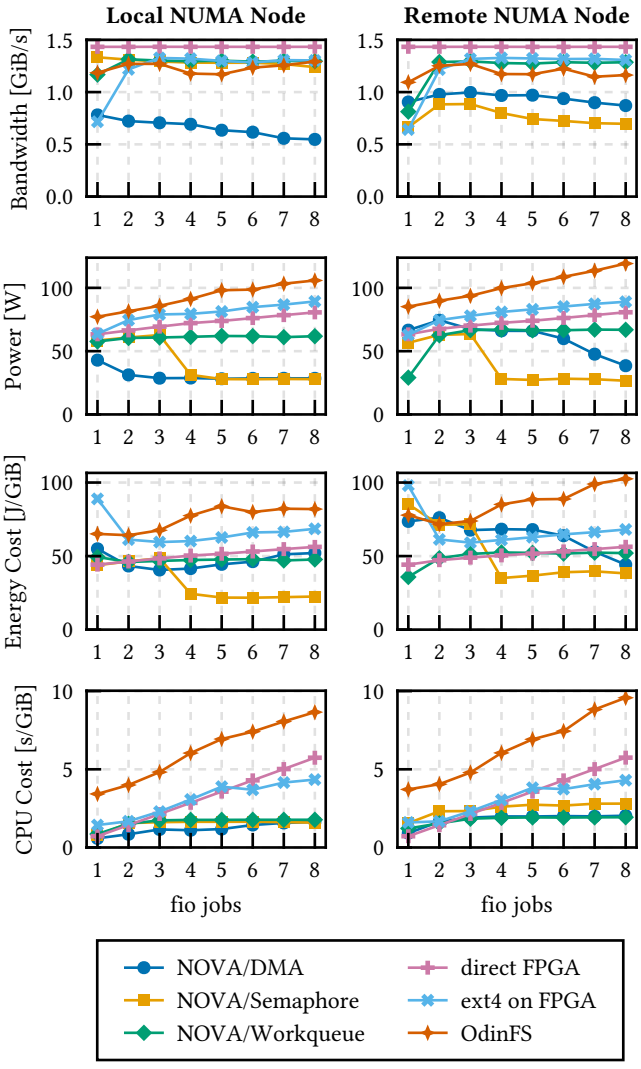

Figure 4. Evaluation of NOVA with our copy mechanisms as well as our FPGA-managed PM.

7.2 Efficient PM Copy

To evaluate our three copy mechanisms, we have modified NOVA to use them. The benchmark setup is the same as in Figure 1. We show the results in Figure 4. We include OdinFS as comparison to Workqueue, since both delegate copies to another kernel thread. We can see that all three variants successfully keep bandwidth, power, and the CPU cost stable with an increasing amount of jobs.

Semaphore provides good results for accesses from the local NUMA node. It does not suffer from reduced bandwidth with a single job as the two delegation-based mechanisms. However, it cannot prevent low bandwidth that results from remote PM writes. Additionally, we have observed reduced performance when there is high contention at the semaphore, which becomes problematic with smaller block sizes. At

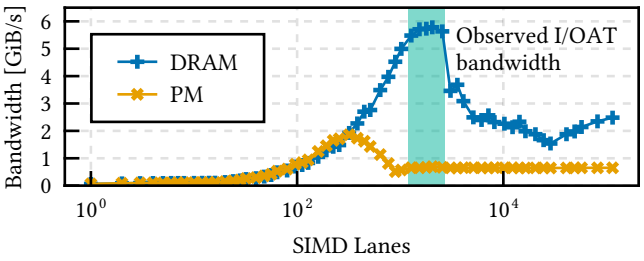

Figure 5. DMA bandwidth from DRAM to DRAM and to PM, using a GPU with varying numbers of active SIMD lanes.

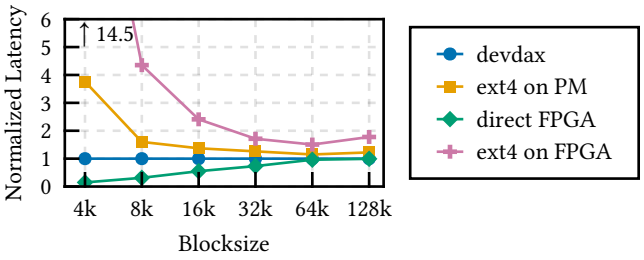

Figure 6. Write latency of single job local random writes at varying block sizes, normalized to devdax latency.

four or more jobs, we measure lower overall power with Semaphore. We discuss this anomaly in Section 3.1.

Bandwidth-wise, Workqueue and OdinFS provide similar performance. However, Workqueue has only a fraction of the CPU cost of OdinFS and is thus more energy-efficient. We analyze the CPU usage of these two benchmarks at eight jobs with perf's stack sampling. Even though the memory copy only requires around 10 % of the CPU in both cases, OdinFS spends 31 % waiting with a spinlock for the delegation thread to complete, compared to only 0.1 % with Workqueue.

The DMA variant provides a relatively stable bandwidth of only around 500 to 750 MiB/s to the local NUMA node in our benchmark, less than half of what the PM can provide. We confirm that there is no bottleneck in our code by running the same benchmark on an emulated PM device backed by DRAM, where we observe a bandwidth that exceeds the PM bandwidth. To get a better understanding of the bottleneck with Intel I/OAT, we measure the DMA bandwidth to DRAM and PM using a GPU. The GPU's parallelism can be configured by controlling the number of SIMD lanes, i.e., the number of ALUs executing the program. We plot the results in Figure 5. With a rising number of SIMD lanes, we observe increasing bandwidth for both PM and DRAM. However, the PM bandwidth peaks at a lower level of parallelism than the DRAM bandwidth. We assume that the internal parallelism in I/OAT is tuned to the DRAM peak, at which point the PM bandwidth matches our observations in Figure 4.

7.3 FPGA-Managed PM

We evaluate the performance of our prototype of FPGA-managed PM with a custom fio ioengine communicating directly with the FPGA from user space, as well as with an ext4 file system on top of our block device driver. The results of the fio benchmark are shown in Figure 4. We additionally measure write latency at varying access sizes in Figure 6. For user space access, we can see a very stable write bandwidth of 1.43 GiB/s from both NUMA nodes. We achieve low latency at small block sizes due to efficient buffering on the FPGA. As the prototype uses polling to wait for request completion, we see an increase in power and CPU cost with more writers. Access over our block device driver shows high latency especially at small access sizes. We attribute the high latency to bad support for polled block devices in the Linux kernel. A future implementation of hybrid polling and interrupts as proposed by Harris et al. [9] could improve latency and energy efficiency.

8 Future Work

CXL. With Compute Express Link [3], we can implement direct access to our FPGA-mediated PM. Given that early measurements of CXL devices show write contention similar to Optane [19], we propose keeping asynchronous access as default and only offering DAX to applications that need it. For devices that offer access only via CXL, we expect that our techniques for limiting parallelism are generally applicable. For DMA offloading, Sun et al. show that Intel Data Streaming Accelerator (DSA) [11], the successor of I/OAT, has better performance accessing CXL memory [19].

High-level interfaces. With io_uring [4], ring buffer based communication is starting to replace traditional system call interfaces to I/O hardware. However, in the current implementation, the kernel interprets io_uring commands, which adds latency and risks security issues [15]. We plan to adopt io_uring as command language for our FPGA-mediated PM. After creating mappings for opened files, this approach allows full kernel bypass for unmodified applications, improving performance and reducing security risks.

9 Conclusion

Managing parallelism is critical for good PM performance. In this paper, we have introduced a metric for assessing the CPU efficiency of PM file systems. Evaluating this metric and measuring energy efficiency of several existing PM file systems, we could observe that these file systems are wasteful with CPU time and energy. We explored three PM copy mechanisms and have shown that they can successfully limit parallel write accesses to PM, reducing CPU usage. We finally introduced a system design with an FPGA as mediator for PM. The FPGA frees the CPU completely from expensive copy operations and communicates with software over ring buffers, offering energy-efficient access at low latency.

References

[1] 2021. *DMAEngine documentation.* https://www.kernel.org/doc/html/v5.15/driver-api/dmaengine/index.html.

[2] 2022. *Intel® Virtualization Technology for Directed I/O Architecture Specification.* https://www.intel.com/content/www/us/en/content-details/774206/intel-virtualization-technology-for-directed-i-o-architecture-specification.html

[3] 2023. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org.

[4] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf

[5] Jens Axboe. 2023. Flexible I/O Tester. https://github.com/axboe/fio.

[6] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '13)*. Association for Computing Machinery, New York, NY, USA, Article 22, 10 pages. https://doi.org/10.1145/2485732.2485740

[7] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/2150976.2151017

[8] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[9] Bryan Harris and Nihat Altiparmak. 2022. When Poll is More Energy Efficient than Interrupt. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (Virtual Event) *(HotStorage '22)*. Association for Computing Machinery, New York, NY, USA, 59–64. https://doi.org/10.1145/3538643.3539747

[10] Tejun Heo and Florian Mickler. 2010. *Concurrency Managed Workqueue (cmwq).* https://www.kernel.org/doc/html/v5.15/core-api/workqueue.html.

[11] Intel. 2023. Intel® Data Streaming Accelerator User Guide. https://www.intel.com/content/www/us/en/content-details/759709/intel-data-streaming-accelerator-user-guide.html

[12] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. https://doi.org/10.48550/ARXIV.1903.05714

[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* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 804–818. https://doi.org/10.1145/3477132.3483567

[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* (Huntsville Ontario Canada, 2019-10-27). ACM, 494–508. https://doi.org/10.1145/3341301.3359631

[15] Tamás Koczka. 2023. Learnings from kCTF VRP's 42 Linux kernel exploits submissions. https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html

[16] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[17] Thai Le, Steven Briscoe, and Jonatha Stern. 2017. Fast memcpy with SPDK and Intel® I/OAT DMA Engine. https://www.intel.com/content/www/us/en/developer/articles/technical/fast-memcpy-using-spdk-and-ioat-dma-engine.html

[18] Jingbo Su, Jiahao Li, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, and Yinlong Xu. 2023. Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/fast23/presentation/su

[19] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. arXiv:cs.PF/2303.15375 https://doi.org/10.48550/arXiv.2303.15375

[20] Sidharth Sundar, William Simpson, Jacob Higdon, Caeden Whitaker, Bryan Harris, and Nihat Altiparmak. 2023. Energy Implications of IO Interface Design Choices. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '23)*. Association for Computing Machinery, New York, NY, USA, 58–64. https://doi.org/10.1145/3599691.3603411

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

[22] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems* (New York, NY, USA, 2022-03-28) *(EuroSys '22)*. Association for Computing Machinery, 488–505. https://doi.org/10.1145/3492321.3519556

[23] Jian Xu and Steven Swanson. 2016. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*. 323––338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[24] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[25] Yang Yang, Qiang Cao, Jie Yao, Yuanyuan Dong, and Weikang Kong. 2021. SPMFS: A Scalable Persistent Memory File System on Optane Persistent Memory. In *Proceedings of the 50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP '21)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/3472456.3472503

[26] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 207–219. https://www.usenix.org/conference/fast19/presentation/zheng

[27] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 179–193. https://www.usenix.org/conference/osdi22/presentation/zhou-diyu