OdinFS: Scaling PM Performance with Opportunistic Delegation

Diyu Zhou Yuchen Qian Vishal Gupta Zhifei Yang Changwoo Min[†] Sanidhya Kashyap

EPFL [†]*Virginia Tech*

1 Introduction

Persistent memory (PM), a storage-class memory, breaks the traditional dichotomy of storage and memory. It offers byte addressability, non-volatility, low latency, and high bandwidth. Recent characterization studies show that PM has many subtle performance characteristics [3, 6, 8], posing a significant challenge for storage stacks to utilize PM performance efficiently.

Such a challenge arises from two unique PM characteristics. The first factor is the tension between concurrent accesses and PM performance. In particular, a small number of threads underutilize PM bandwidth, while a high number of concurrent access threads lead to PM performance meltdown. The meltdown happens because a high number of concurrent access threads render the caching and prefetching in PM inefficient [1, 8]. The second factor is the pronounced NUMA impact on PM, as several prior works found that remote NUMA accesses on PM are much slower than DRAM, leading to at least 2× bandwidth reduction [1, 6]. Unfortunately, none of the existing PM file systems [2, 4, 7] utilizes these two unique PM characteristics effectively.

This paper presents OdinFS: (Opportunistic DelegatIoN File System), a NUMA-aware PM file system that maximizes PM performance[1]. We design OdinFS with three major design goals: (1) **Limit concurrent PM accesses (access arbitration)**: OdinFS controls the number of PM access threads to maintain the maximal PM performance within a NUMA node. (2) **Localized PM accesses (NUMA-awareness)**: OdinFS ensures threads always access the local PM within a NUMA node, thereby avoiding the PM NUMA impact. (3) **Automatic parallel PM accesses (automatic parallelization)**: OdinFS automatically parallelizes applications' PM access requests across all NUMA nodes without application modification. OdinFS thus efficiently utilizes aggregated PM bandwidth, thereby improving application performance. As detailed in the next section, OdinFS achieves these goals by proposing a new approach—***opportunistic delegation***—that decouples PM data accesses from application threads.

2 OdinFS Design

Figure 1 shows the key components of OdinFS and their typical workflow. We next present the key design of OdinFS and explain how they meet the design goals of OdinFS.

(1) NUMA-striped data layout for cumulative PM bandwidth utilization. Unlike other NUMA-aware PM file sys-

[1]This work has been accepted to OSDI '22 [9].

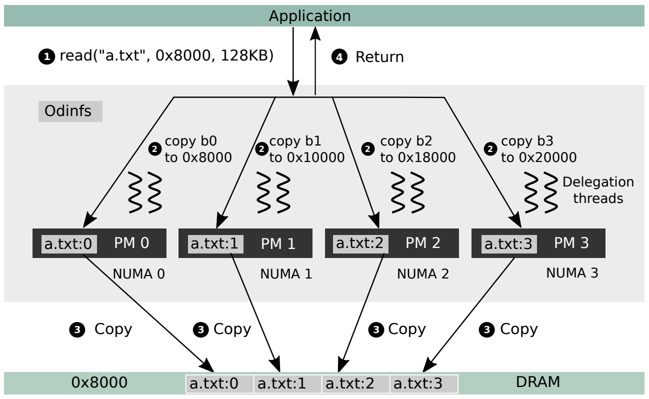

Figure 1: Overview of OdinFS. Each NUMA node has delegation threads that access local PM on behalf of application threads. OdinFS stripes the file data across all PM NUMA nodes. ❶ An application thread issues a read system call. ❷ OdinFS divides the system call into multiple access requests based on the stripe size and sends them to the delegation threads. ❸ The delegation threads read from PM in different NUMA nodes in parallel to service the access requests. ❹ The application thread returns.

tems that try to localize file accesses within a single PM NUMA node [5], OdinFS stripes the data of every file across PM on each NUMA node in a round-robin manner. OdinFS makes this design choice since it can minimize the PM NUMA impact *with delegation*, as detailed below. Furthermore, stripping file data across PM enables OdinFS to exploit all available PM bandwidth to handle application requests, which opens the door for automatic request parallelization.

(2) Delegation-based PM accesses to maximize PM performance. A key insight in OdinFS is that the access arbitration, NUMA-awareness, and automatic parallelization design goals can be simultaneously achieved by decoupling PM data accesses from application threads through delegation. In particular, for each NUMA node, OdinFS creates several background threads (delegation threads). Only the delegation threads can access PM. When the application thread needs to access PM, it first checks which NUMA node the PM address belongs to and then sends the PM access requests to one of the delegation threads on that NUMA node. The delegation thread performs the access on behalf of the application thread and informs the application thread when the access completes.

Since only the delegation threads can access PM, they effectively act as a central entity to *arbitrate* PM access. Regardless of the application thread count, delegation threads decide the level of concurrent accesses to PM. Thus, OdinFS accesses

PM with a thread count that avoids the PM performance collapse with many concurrent access threads. This effectively achieves the *access arbitration* design goal. Since the delegation threads are in the same NUMA node as PM, ODINFS always access PM locally. Thus, ODINFS minimizes the PM NUMA impact, achieving the *NUMA-aware* design goal.

(3) Automatic parallelization at the system call boundary. The data striping and the delegation threads allow ODINFS to serve I/O requests from applications in parallel across all the NUMA nodes. Moreover, the POSIX interface enables ODINFS to *automatically* parallelize the requests without modifying applications. Specifically, ODINFS divides all data system call (*e.g.*, `read`, `write`, `pread`, `writev`) requests into multiple independent sub-requests based on the stripe size, and sends them to the corresponding delegation threads. The delegation threads then execute these requests by accessing PM in different NUMA nodes in parallel. Figure 1 illustrates the case. In this way, ODINFS achieves the *automatic parallelization* design goals.

(4) High scalability with full PM performance. Delegating PM access allows ODINFS to maximize PM performance. ODINFS further maximizes concurrent accesses to ensure applications can benefit from the performance gains even under the high contention case. Specifically, ODINFS increases the disjoint data access parallelism with a readers-writer range lock for each inode. This enables concurrent writes to disjoint regions and concurrent reads from the same file region. The use of range lock poses a significant challenge for enforcing crash consistency. ODINFS overcomes this issue by preserving the whole inode lock and falling back to it for concurrency control if needed.

3 Evaluation

Evaluation environment. We conduct our evaluation on an eight-socket server. Each socket equips a 28-core Intel Xeon processor (224 cores in total) and six 128GB Intel Optane DIMMs interleaved at 4KB. We configure ODINFS to run on all eight NUMA nodes with twelve delegation threads on each NUMA node. We compare ODINFS with `ext4`, `PMFS` [2], `NOVA` [7], and `WineFS` [5]. We further include one setup (`ext4(RAID0)`) by creating a RAID0 across all eight PM NUMA nodes and mount `ext4` on top of it. We also emulate a non-existent setup: `NOVA(MN)` (`NOVA` with multiple nodes) to estimate the performance of a NUMA-aware `NOVA` by mounting a single instance of the `NOVA` file system on each NUMA node and evenly distribute the testing files among instances.

Throughput. We use `fio` to measure throughput by letting each thread access a private 1GB file. Figure 2 shows the throughput of all evaluated file systems. For `4K-read`, only ODINFS and `NOVA(MN)` scale beyond one NUMA node, outperforming other file systems by 9.4× with 224 threads. For `4K-write`, when the thread count is low, ODINFS suffers from the communication overhead of delegation and is up to 62% slower than other file systems. However, ODINFS can maintain its throughput thanks to limiting PM accesses, outperforming others by up to 8.1×. With the 2MB access size, ODINFS benefits from accessing all PM NUMA nodes in parallel to serve IO requests, outperforming other file systems by 1.1× to 24.7×, and up to 14.8× for `2M-read`, and `2M-write`, respectively.

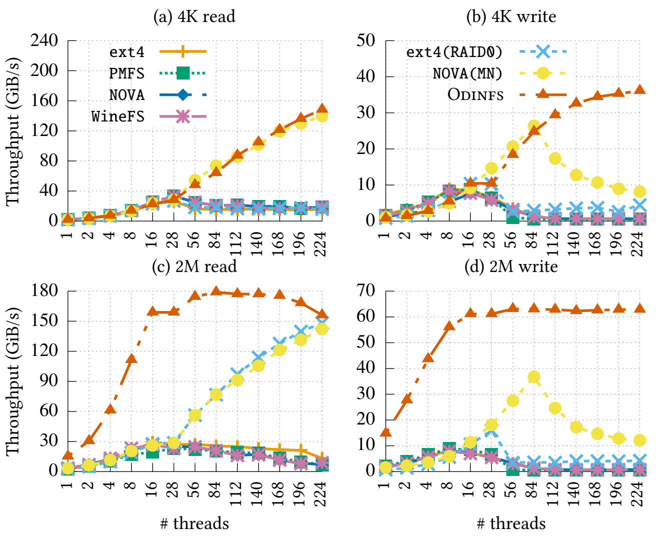

Figure 2: Throughput of evaluated file systems.

References

[1] B. Daase et al. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 ACM SIGMOD/PODS Conference*, Xi'an, Shaanxi, China, May 2021.

[2] S. R. Dulloor et al. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[3] J. Izraelevitz et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714*, 2019.

[4] R. Kadekodi et al. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[5] R. Kadekodi et al. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[6] W.-H. Kim et al. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[7] J. Xu and S. Swanson. 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)*, Santa Clara, CA, Feb. 2016.

[8] J. Yang et al. 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)*, Santa Clara, CA, Feb. 2020.

[9] D. Zhou et al. Odinfs: Scaling PM performance with Opportunistic Delegation. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022. URL `https://zhou-diyu.github.io/files/odinfs-osdi22.pdf`.