ODINFS: Scaling PM Performance with Opportunistic Delegation

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

EPFL [†]Virginia Tech

Abstract

Existing file systems for persistent memory (PM) exploit its byte-addressable non-volatile access with low latency and high bandwidth. However, they do not utilize two unique PM properties effectively. The first one is contention awareness, *i.e.*, a small number of threads cannot thoroughly saturate the PM bandwidth, while many concurrent accesses lead to significant PM performance degradation. The second one is NUMA awareness, *i.e.*, exploiting the remote PM efficiently, as accessing remote PM naively leads to significant performance degradation.

We present ODINFS, a NUMA-aware scalable datapath PM file system that addresses these two challenges using a novel ***opportunistic delegation*** scheme. Under this scheme, ODINFS decouples the PM accesses from application threads with the help of background threads that access PM on behalf of the application. Because of PM access decoupling, ODINFS automatically parallelizes the access to PM across NUMA nodes in a controlled and localized manner. Our evaluation shows that ODINFS outperforms existing PM file systems up to 32.7× on real-world workloads.

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 [8, 14, 23, 43]. Recent characterization studies show that PM has many subtle performance characteristics [18–20, 23, 27, 29, 37, 39, 40, 43], 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[1] lead to PM performance meltdown [39]. The meltdown happens because a high number of concurrent access threads render the caching and prefetching in PM inefficient [14, 43]. 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 [14, 27, 38]. Supporting multiple NUMA domains is currently the primary way to increase PM's capacity and aggregated bandwidth. Unfortunately, the pronounced NUMA impact defeats the purpose of PM NUMA architecture.

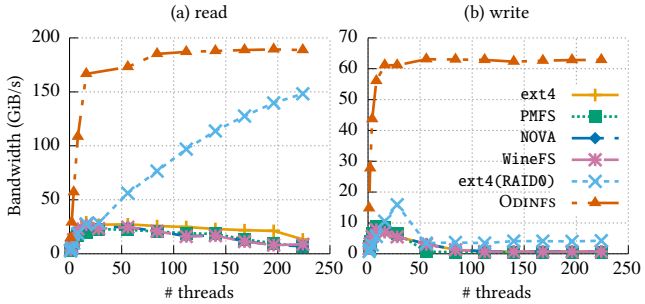

Figure 1: The maximal PM read and write bandwidth of four PM file systems and ODINFS. `ext4(RAID0)`: `ext4` mounted on a RAID0 built from PM across all NUMA nodes. Benchmark: `fio`, in which each thread accesses a private file at the granularity of 2MB sequentially, on an eight-socket machine.

Several proposed PM file systems exploit various characteristics of PM [10, 12, 16, 17, 24, 25, 28, 34, 41]. However, none of the existing PM file systems considers the tension between concurrent accesses and PM performance. Moreover, the conventional approach to mitigate the PM NUMA impact is to migrate data to CPUs or vice versa [25, 42], which incurs a high migration overhead, and cannot efficiently utilize the aggregated PM bandwidth. Figure 1 illustrates the issues for several PM file systems [2, 17, 25, 41]. The bandwidth of these file systems highly depends on the thread counts. Except for `ext4(RAID0)` performing the read workload, the read and write bandwidth of these file systems collapse after the thread count exceeds a certain threshold. In summary, existing PM file systems cannot efficiently utilize PM in a NUMA setup and incur performance collapse if multiple threads of

[1]In this paper, we use the term "access thread" to denote a thread that directly accesses PM. It could be an application thread or a kernel thread.

the application (*e.g.*, file server software and video streaming software) access PM concurrently.

This paper presents OdinFS: (Opportunistic DelegatIoN File System), a NUMA-aware PM file system that maximizes PM performance by controlling concurrent accesses, minimizing the NUMA impact, and parallelizing PM accesses to utilize the aggregated bandwidth. To design OdinFS, we first holistically analyze the behavior of existing PM file systems on an eight-socket NUMA machine. We analyze two issues specifically: maintaining maximum PM performance within a NUMA node and minimizing the PM NUMA impact. For the first issue, we find that both read and write performance of PM collapse with high thread counts, while prior work only reports the write performance collapse [14, 43]. For the second issue, we provide a detailed analysis and quantitatively confirm that placing the access threads in the same NUMA node as PM minimizes the NUMA impact.

Motivated by our performance analysis, 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.

OdinFS achieves these goals by proposing a new approach—***opportunistic delegation***—that decouples PM data accesses from application threads. Specifically, on each NUMA node, OdinFS creates multiple background kernel threads (delegation threads) that access PM on behalf of the application threads. Therefore, OdinFS limits the maximum concurrency within PM by controlling the number of delegation threads. Moreover, the delegation threads are local to each NUMA node, leading to NUMA-aware localized accesses. OdinFS thus departs from the conventional NUMA-mitigation approaches in PM file systems that mainly involve data or thread migration.

Furthermore, the delegation threads enable servicing bulk data requests by efficiently utilizing aggregated PM bandwidth across all NUMA nodes. Specifically, OdinFS first stripes the file data across PM in all NUMA nodes. Exploiting the well-designed system call interface, OdinFS services data system calls (*e.g.*, `read()` and `write()`) by transparently dividing them into multiple disjoint access requests based on the stripe size and sending such access requests to the corresponding delegation threads. The delegation threads thus access PM in different NUMA nodes in parallel to serve the system call. We further enhance OdinFS with fine-grained parallelism for data operations. Our evaluation shows that OdinFS outperforms other file systems by up to 32.7× for real-world workloads and has up to two orders of magnitude performance improvement for scalability microbenchmarks.

This paper makes the following contributions:

- **Analysis.** We thoroughly analyze the behavior of existing PM file systems on a large NUMA machine and reveal two new findings.
- **Opportunistic delegation.** We propose an opportunistic delegation scheme for PM file systems that decouples PM data accesses from application threads, thus efficiently utilizing both local and remote PM.
- **OdinFS** We design and implement OdinFS: a PM file system that builds on the opportunistic delegation scheme with state-of-the-art concurrency control mechanisms. OdinFS maximizes the performance and further scales data operations with increasing threads.

2 PM Performance Analysis

Prior study has shown that the underlying architecture of PM is quite complicated [39], and PM has limited bandwidth and higher latency compared to DRAM [14, 40]. Moreover, naively utilizing PM in NUMA machines often underutilizes PM or leads to performance collapse [14, 25, 27, 38]. To showcase the issues of concurrent NUMA accesses in PM, we first provide an overview of the current hardware (§2.1). We then analyze why existing PM file systems fail to handle many concurrent requests (§2.2) and the impact of different types of accesses in a NUMA machine (§2.3).

2.1 Intel Optane internals

The Intel Optane [8] PM is the only publicly available nonvolatile memory technology so far. A memory controller accesses PM at the granularity of a cache line (64 bytes). However, the access size of the internal 3D-Xpoint storage media is 256 bytes. Such an access size mismatch results in read or write amplification. The 3D-Xpoint media includes a buffer (*XPBuffer*) and an associated prefetcher (*XPPrefetcher*) to address this issue and mitigate its long latency. The XPBuffer combines adjacent accesses to PM, and the XPPrefetcher prefetches blocks in the 3D-Xpoint media to XPBuffer based on the access pattern. In addition, the 3D-Xpoint media also stores the inter-NUMA node coherence information [6, 27]. Hence, inter-NUMA accesses may involve writing to the 3D-Xpoint media to update the coherence information, which is the root cause of the slow inter-NUMA PM accesses (§2.3).

2.2 Concurrent accesses to PM

Prior work [14, 43] finds that PM performance depends on the access size and the number of concurrent access threads. The impact of access size is well understood. Applications accessing PM perform well as long as the access size is large enough to stress all the interleaved PM DIMMs. To understand the impact of concurrent access threads, we run `fio` [4], in which each pinned thread sequentially accesses a private 1GB file at a 2MB granularity. We evaluate on an eight-socket

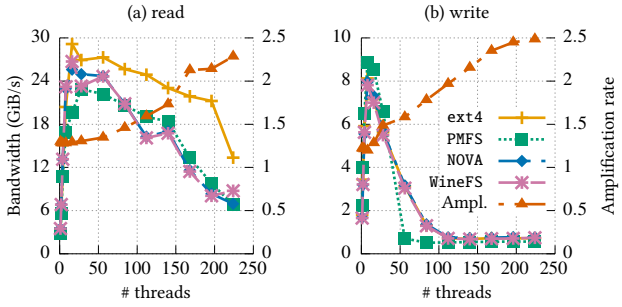

Figure 2: PM read and write bandwidth of PM file systems with increasing threads for sequential 2MB access size. Results show that both read and write performance collapse after exceeding a specific limit. We observe a dramatic increase in read/write amplification due to cache thrashing in the PM storage device.

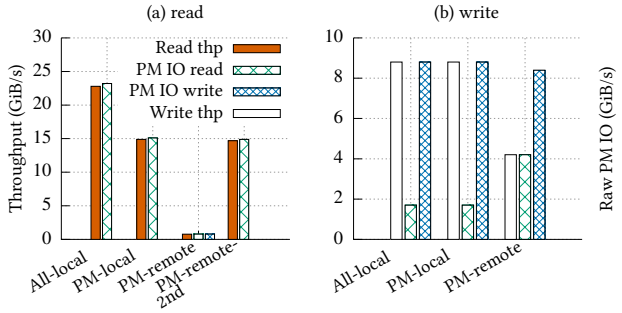

Figure 3: Application throughput and the raw PM I/O for reading data from PM (left) and writing data to PM (right) with the same workload in §3.6 and the following configurations. *All-local*: Access threads, PM, and DRAM (I/O buffer) are in NUMA node 0. *PM-local*: Access threads and PM are in NUMA node 0; DRAM is in NUMA node 1. *PM-remote*: PM is in NUMA node 0; access threads and DRAM are in NUMA node 1. *PM-remote-2nd*: A consecutive run with *PM-remote*.

NUMA machine, with each socket having six interleaved Optane DIMMs and a processor with 28 cores.

Figure 2 shows the read and write bandwidth of PM file systems with increasing threads. Both read and write reach their peak bandwidth with sixteen and eight threads, respectively. After that, increasing threads severely degrades the overall bandwidth. Specifically, with 224 threads, the read and write bandwidth degrades by 3.7× and 17.2× compared to the peak bandwidth, respectively.

Such performance collapse occurs because high concurrent accesses thrash the underlying cache of PM.[2] Specifically, a mismatch exists between the CPU access size (64 bytes) and the underlying PM storage access size (256 bytes). PM minimizes the read/write amplification overhead by batching writes (XPBuffer) and prefetching (XPPrefetcher) (§2.1). However, with high concurrent accesses, the sequential accesses from different threads convert into non-adjacent accesses. These accesses arrive at the PM simultaneously, which reduces the efficiency of both caching and prefetching. As a result, it increases read and write amplification, as XPBuffer cannot keep up with the requests and the underlying PM media latency starts to dominate for fetching or writing data, leading to performance collapse. The read collapse threshold is higher than write since reads perform better than writes with the 3D-Xpoint media. Thus, PM can sustain the read bandwidth despite reducing caching and prefetching efficiency up to two NUMA nodes.

We find that both read and write bandwidth crashes after a certain point. The results for writes are consistent with prior works [14, 43]. However, we find that read bandwidth also starts to collapse after two NUMA nodes. This is contrary to prior work, which reports that the read bandwidth scales with increasing threads.

Insight #1. A file system must control the number of threads that concurrently access PM for both reads and writes to preserve the maximal performance within a NUMA node.

[2]We verify that the performance collapse is not due to the scalability bottleneck in the file systems by confirming that most of the CPU cycles are spent in accessing PM.

2.3 NUMA impact on PM

We now analyze the NUMA effect on PM. `WineFS` [25] proposes to migrate a thread to a PM NUMA node to mitigate the NUMA impact [25]. Unfortunately, there is no in-depth analysis of the effectiveness of this mechanism. Specifically, suppose a thread copies data between remote PM and local DRAM. In this case, migrating the thread to the respective PM NUMA node still involves remote memory access, and thus it is not clear why or how the thread migration can improve performance.

We answer the aforementioned question by investigating the performance impact of NUMA placements of thread, DRAM, and PM. We configure `fio` with three setups. In the *All-local* setup, threads, PM, and DRAM (*i.e.*, I/O buffer) are in the same NUMA node (node 0), which serves as the best-case scenario. In the *PM-local* setup, threads and PM are in the same NUMA node (node 0), while DRAM is in a remote NUMA node (node 1). In the *PM-remote* setup, PM is in NUMA node 0, while threads and DRAM are in the same NUMA node (node 1). We evaluate this experiment using `PMFS` with 28 threads for read and 8 threads for write. Other file systems or thread counts produce similar results.

Figure 3 shows the PM read and write throughput with the three setups. We note that both *PM-local* and *PM-remote* perform the same task of copying data between PM on NUMA node 0 and DRAM on NUMA node 1. However, *PM-local* achieves nearly 19.1× and 2.1× higher throughput than *PM-remote*. Furthermore, *PM-local* achieves 60% and almost 100% of *All-local* read and write throughput, respectively. The above results are due to the implementation of the *directory coherence protocol* in Intel machines [6, 27]. Specifically, Intel maintains intra-NUMA and inter-NUMA directory information separately [5]. The processor cache and memory store the intra-NUMA directory and inter-NUMA directory information, respectively. With the *PM-local* setup, the PM cache blocks become NUMA-local: data is written to the processor

cache; while the DRAM cache block moves between NUMA nodes. Hence, the system updates the PM directory locally, while writing to DRAM to update its directory information. However, with the *PM-remote* setup, the processor updates DRAM directory information on the processor cache, while updating its directory information on PM. Hence, the performance difference between DRAM and PM leads to the performance difference between *PM-local* and *PM-remote*.

To verify the above claim, we used Intel PCM [7] to measure the PM device level read/write IO, as shown in Figure 3. For *All-local* and *PM-local*, the total bytes written and read from the PM device match the actual I/O bandwidth, indicating no directory information update. However, *PM-remote* incurs extra read and write traffic to the PM device. Furthermore, a consecutive read with *PM-remote* (*PM-remote-2nd*) can restore the performance, while a consecutive write in *PM-remote* still suffers from NUMA impact. The above evidence confirms that *PM-remote* involves directory coherence updates. Specifically, the extra read and write traffic is due to updating the coherence information. The *PM-remote-2nd* setup can only restore the read performance since, in this case, coherence information update is not needed for read but is still required for write. In summary, our performance analysis quantitatively confirms that placing the access threads and PM in the same NUMA node minimizes the NUMA impact on PM.

Insight #2. To minimize the pronounced PM NUMA impact, and efficiently utilize remote PM, a file system should place the access threads local to the PM.

3 OdinFS Design

Following our performance analysis on PM (§2), we present OdinFS, a NUMA-aware PM file system that maximizes PM performance within and across NUMA nodes through opportunistic delegation. This section first presents the design goals that enable OdinFS to maximize PM performance (§3.1), an overview of OdinFS (§3.2), followed by the design of each individual component.

3.1 OdinFS Design Goals

We design OdinFS to meet the following goals:

- **Limiting concurrent PM access (access arbitration).** To avoid the PM performance collapse with many concurrent PM accesses (§2.2), OdinFS should limit the concurrency to maximize PM performance within a single NUMA node.
- **Localized PM access (NUMA-awareness).** To avoid the performance collapse due to the PM NUMA impact, OdinFS only allows threads to access local PM (§2.3). This minimizes the PM NUMA impact and opens the opportunity for OdinFS to utilize remote PM efficiently.
- **Automatic parallel PM access (automatic parallelization).** The access arbitration and NUMA-aware design goals allow OdinFS to maximize the local and remote PM performance. To fully benefit from the aggregated PM bandwidth, OdinFS further parallelizes large PM accesses across all NUMA nodes automatically without application changes.
- **Scalability.** Scalability is the overarching goal of OdinFS. Modern machines have multiple NUMA nodes and hundreds of CPUs. The above three design goals allow OdinFS to scale PM performance with an increasing core count. Beyond that, OdinFS should maximize concurrent accesses within the same file.

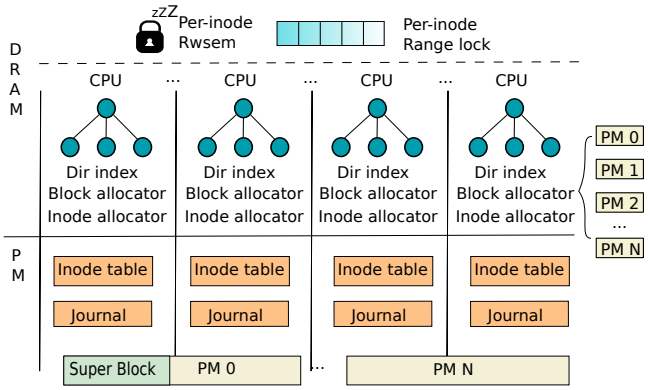

Figure 4: OdinFS architecture. OdinFS maintains per-CPU data structures to minimize the synchronization overhead. Furthermore, OdinFS maintains the directory index, block allocator, and inode allocator in DRAM to maximize performance. OdinFS enhances the per-inode readers-writer semaphore with our optimized range lock to increase concurrency.

3.2 OdinFS Architecture

Figure 5 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 (§3.1).

(1) NUMA-striped data layout for cumulative PM bandwidth utilization. Unlike other NUMA-aware PM file systems that try to localize file accesses within a single PM NUMA node [25, 42], 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

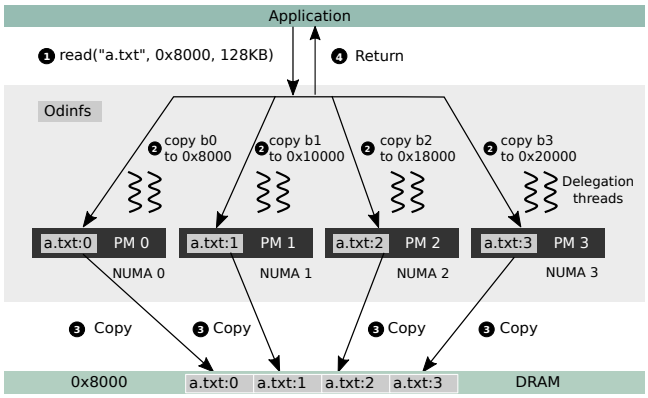

Figure 5: Overview of Odinfs. Each NUMA node has delegation threads that access local PM on behalf of application threads. Odinfs stripes the file data across all PM NUMA nodes. ❶ An application thread issues a read system call. ❷ Odinfs divides the system call into multiple access requests based on 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.

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 (§2.2). This effectively achieves the *arbitration* design goal. Since the delegation threads are in the same NUMA node as PM, Odinfs always access PM locally, in either *All-local* or *PM-local* setup (§2.3). 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 IO 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 5 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, most existing PM file systems globally protect the inode [2, 17, 25, 41]. Odinfs further 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. (§3.7).

Odinfs design novelty. To the best of our knowledge, Odinfs is the first PM file system that addresses the goal of access arbitration and automatic parallelization. While some NUMA-aware file systems mitigate PM NUMA impact [25], they either move computation to data [25], or move data to computation [42]. Odinfs proposes a fundamentally different approach by using the delegation to minimize the NUMA impact. Furthermore, Odinfs extends the scope of the NUMA-aware file systems. Instead of focusing only on minimizing the NUMA impact, Odinfs takes a radical approach of parallelizing and striping data across all PM NUMA nodes for the best performance. Moreover, our controlled PM access approach also minimizes I/O amplification (§2.2), leading to low write amplification. Lower write amplification further increases the life of the PM device and minimizes the long latency that happens due to wear leveling [39].

3.3 Handling system calls

Odinfs is a POSIX-compliant in-kernel file system. A key novelty in Odinfs is the PM access delegation. However, since delegation incurs communication overhead, Odinfs does not delegate small PM accesses (§3.5). As a result, the delegation threads only perform data operations, while the metadata operations (*e.g.*, `open`, `close`) are handled by application threads. Furthermore, application threads directly access PM for small data operations.

Handling bulk data operations with delegation. After issuing a data system call (*e.g.*, `write`) and entering into the kernel space, the application thread divides the requests into multiple sub-requests, each consisting of a data stripe (§3.4). For each access request, the application thread walks the indexing structure and obtains the corresponding address on PM and the NUMA domain. It then enqueues the request (*e.g.*, source and destination memory address, access length) on a corresponding ring buffer of the PM NUMA node. After enqueueing all requests, the application thread waits for delegation threads to complete and then returns to the user space. Meanwhile, a delegation thread receives the request via the ring buffer. The delegation thread dequeues the request and accesses PM on behalf of the application thread by copying the data between PM and the specified DRAM buffer. After completing the request, the delegation threads notify the application threads.

3.4 NUMA-aware PM allocation

NUMA-aware allocator. To operate on multiple PM NUMA nodes and stripe file data across them (§3.2), Odinfs designs a NUMA-aware PM allocator. Odinfs inherits the performant and scalable allocator design from `NOVA` and `WineFS` and ex-

tends the allocator design to handle multiple PM NUMA nodes. OdinFS uses a per-CPU allocator residing on DRAM, in which each CPU owns multiple private PM pools, each corresponding to one PM NUMA node (Figure 4). Block allocation works as follows: The allocator receives the allocation request with a block size and a NUMA node as arguments. It first tries to serve the request from its own per-CPU PM pool. If that fails, it tries to serve the requests from the pool in the specified NUMA node having the largest free space. If both attempts fail, the allocator returns an error.

Layout policy. OdinFS employs different layout policies for file data, indexing structure, and other metadata. OdinFS stripes the file data across all PM NUMA nodes to enable parallel access (§3.2). Since OdinFS does not delegate small PM accesses (§3.5), it optimizes for small files by placing the first stripe of each file in the local PM node as the creation thread, if possible. This assumes that the following accesses are likely from the same NUMA node thanks to temporal locality. OdinFS places the remaining stripes in a round-robin fashion across all PM NUMA nodes. With our system, the memory controller in each NUMA node interleaves six PM DIMMs at 4KB granularity. Therefore, we set the stripe size as 32KB to maximize PM performance. Since OdinFS does not delegate access to the indexing structure, OdinFS places it in PM local to the CPU that creates the file, leveraging temporal locality. Regarding other metadata, OdinFS places the superblock in the first PM NUMA node and per-CPU metadata (*e.g.*, journaling) in the local PM node (Figure 4).

3.5 Opportunistic delegation

Since delegating PM accesses involves communication overhead, it is not always beneficial, especially for small accesses. Thus, OdinFS performs opportunistic delegation only for PM accesses that might improve the performance. Based on our performance analysis (§2), OdinFS uses different delegation policies for PM reads and writes.

Write access. OdinFS always delegates write accesses with an access size larger than 256 bytes (XPBuffer size) to limit the performance collapse and minimize the NUMA impact.

Read access. Unlike writes, OdinFS chooses a more relaxed policy for delegating reads. Specifically, PM read performance starts to collapse with a high thread count (> 56). Furthermore, the PM read performance can be restored after repetitive remote reads (PM-remove vs. PM-remote-2nd in Figure 3). Thus, OdinFS checks the number of threads that read from each PM NUMA node for every fixed interval. If it finds that for one PM NUMA node, the number of threads is constantly higher than the collapse threshold (56), OdinFS arbitrates access to that PM device by using the same policy as write. Otherwise, OdinFS only delegates the read accesses that may benefit from the automatic parallelization (*i.e.*, those with access size larger than the stripe size: 32KB). We find this policy is enough to achieve good performance.

Saving CPU cycles. A delegation thread uses a variant of the spin-then-park strategy to 1) avoid wasting CPU cycles when there is no request and 2) minimize the long latency due to naively parking and waking up threads [26]. OdinFS uses the IO size of application threads as a heuristic to decide the length that a delegation thread should spin before parking. The spinning is inversely proportional to the IO request size. For example, for large IO requests, delegation threads spin for a shorter duration because they can amortize the parking/wake-up latency by handling long requests. On the other hand, delegation threads spin for a longer duration for small IO requests, since we assume that application threads are likely to issue sparse requests in this case. We find that this heuristic works well for every evaluated workload.

3.6 Concurrency control

Prior in-kernel PM file systems [17, 25, 41] rely on VFS's inode lock for concurrency control. Inspired by recent works [12, 36], OdinFS increases fine-grained access to a file with a per-inode readers-writer range lock. The lock allows parallel writes to disjoint regions, while concurrent reads on the same region in a file. The existing concurrency control mechanisms still protect other operations.

3.7 Crash consistency

Consistency mode. OdinFS currently supports two consistency modes: POSIX and sync [24]. The POSIX mode guarantees that all metadata operations are synchronous and atomic (*e.g.*, `ext4`). The sync mode is the default setup that in addition to POSIX mode, further ensures that all data operations are synchronous but not atomic. Specifically, when the system call returns, the data is guaranteed to persist on PM. However, a crash may cause data operations being partially completed. This provides the same guarantee as `PMFS` and the "relaxed mode" of `NOVA`. If required, we can extend OdinFS to provide other consistency modes [24, 25, 41].

Atomic updates and per-CPU journaling. OdinFS provides metadata crash consistency with atomic updates [17] and per-CPU journaling [41]. Intel architecture only supports 8 bytes as atomic updates and (aligned) 16 bytes with the double compare-and-exchange operation. OdinFS leverages this to update simple metadata whenever possible. For complex metadata updates, OdinFS leverages journaling for crash consistency. Note that OdinFS does not need to journal file data for its current consistency models. OdinFS inherits the per-CPU undo journal design from `WineFS` [25]. As detailed below, OdinFS ensures a file can only be in one per-CPU journal at any time. Hence, OdinFS can recover from the per-CPU journals by using a global transaction ID.

Ensuring crash consistency with range locks. Since the range lock allows multiple threads to access the same file (§3.2), ensuring crash consistency becomes a challenge. OdinFS addresses this issue by maintaining an invariant that a file can only be in one per-CPU journal. The key idea is that if a thread performs any operation that requires jour-

naling, it must acquire the writer lock of the whole inode lock for exclusiveness, even if this may reduce concurrency. Specifically, we classify data operations into three types: *read*, *overwrite* (writes to an existing file block), and *unallocated write* (writes to an unallocated file block, such as appending a file or writing to holes in a sparse file). Only the metadata stored in the inode, such as access or modification time, needs to be updated for read and overwrite. Following PMFS strategy, OdinFS stores these fields in a 16-byte PM block and updates them atomically without journaling. Hence, OdinFS can allow concurrent reads and overwrites to the same file. However, an unallocated write updates both inode and multiple blocks in the indexing structure and thus requires journaling. Hence, OdinFS only allows one thread to perform unallocated write at a time to maintain the invariance.

Thus, for reads, OdinFS acquires the reader lock of the whole inode lock and reader lock of the relevant range in the range lock. For writes, OdinFS distinguishes between overwrite and unallocated write. OdinFS first acquires the reader lock of the whole inode lock and walks the indexing structure to identify whether the write involves writing to unallocated blocks. No journaling is required if the write only updates allocated blocks (*i.e.*, overwrite). Hence, the thread can proceed by acquiring the writer lock of the relevant range in the range lock. Otherwise, it is an unallocated write and requires journaling. The thread then upgrades from the reader lock to the writer lock of the whole inode lock to ensure the inode is only in one per-CPU journal.

4 OdinFS implementation

File system implementation. We modify and extend PMFS [17] to design and implement OdinFS, while also referring to NOVA [41] and WineFS [25]. In summary, the inode table consists of multiple blocks forming a linked list. The directory data structure is similar to a linked list. The indexing structure of a regular file is a B-tree. Crash consistency is achieved with atomic updates and undo journaling (§3.7). OdinFS maintains the inode allocator, block allocator, and cached directory entries in DRAM with red-black trees. The state of the inode and block allocator needs to persist across power cycles. Thus, OdinFS writes their state to PM during unmount and reads from PM during mount. Upon crash, OdinFS recovers the state by scanning used inodes and their indexing structures. To minimize the synchronization overhead, inode allocator, block allocator, inode table, and journaling use per-CPU data structures. To handle complex metadata operations, such as `rename` or `mmap`, OdinFS follows PMFS by using the synchronization mechanisms in both VFS and the file system. We implemented OdinFS as a kernel module for the 5.13.13 Linux kernel and thus, its deployment challenges and manageability are similar to other in-kernel file systems. OdinFS is publicly available at https://github.com/rs3lab/Odinfs.

Efficient communication with delegation threads. Application and delegation threads communicate via a ring buffer (§3.3). To minimize communication overhead, we adopt the scalable ring buffer implementation from Solros [32]. Furthermore, each delegation thread has its private ring buffer to reduce the contention. The application threads send requests to a random delegation thread in the target NUMA node to load balancing delegation threads. We choose this algorithm since it incurs minimal runtime overhead without central coordination while achieving good performance. For PM access request notification, we use a pair of per-NUMA counters: issued counter and completed counter. The application thread increases the issued counter for each request, and sends the pointer of the completed counter. After issuing all the requests, the application thread waits until the number of issued request count on each NUMA node equals the per-NUMA completed count, which is atomically updated by delegation threads.

Accessing userspace memory via delegation. Delegation threads do not have access to the application address space, even though both of them are in the kernel space when handling a system call. We resolve this issue by first letting the application thread pre-faults and pins the user buffer pages in the kernel. It then passes the user buffer along with its root page table information (`mm->pgd`) to the delegation thread. Upon receiving the request, the delegation thread walks the page table for each user buffer page to figure out the physical page. Since the Linux kernel maps all physical pages into its address space linearly, the delegation threads can obtain the corresponding kernel virtual address by adding an offset. The delegationt threads can then access the user buffer with the kernel virtual address.

Minimizing synchronization overhead. To achieve the scalability design goal (§3.1), OdinFS further adopts state-of-the-art synchronization mechanisms to minimize the synchronization overhead. Specifically, OdinFS enhances the readers-writer range lock (§3.6) with BRAVO [15]. BRAVO optimizes the reader side performance of a readers-writer lock by leveraging a hash table, thus avoiding updating the shared reader counters. As discussed in §3.7, since a thread only acquires the writer lock of the whole inode lock for unallocated-write, we use the readers-writer semaphore in [30] for the whole inode lock. The per-CPU readers-writer semaphore optimizes the reader side performance of the semaphore with a per-CPU counter.

CPU usage fairness with delegation. To ensure fairness, OdinFS charges the request serving time of delegation threads to the application thread. Specifically, the application thread passes a pointer to its CPU usage time (`vruntime`) in each request, and the delegation threads thus atomically update it accordingly.

Implementation limitations. In the current implementation, we pin each delegation thread on a particular CPU.

Furthermore, with the current Linux scheduler, if application threads are also pinned to the same CPU, both the delegation and the application threads' performance will degrade. We plan to explore a lightweight and more efficient thread scheduling algorithm to address this issue.

5 Evaluation

We evaluate OdinFS by answering the following questions:

- How does opportunistic delegation affect OdinFS' performance? (§5.2)
- What is the I/O amplification factor of OdinFS? (§5.3)
- Does OdinFS scale with different delegation thread counts and PM NUMA nodes? (§5.4)
- Does OdinFS scale data operations? (§5.5)
- How does OdinFS perform with real-world applications? (§5.6)

5.1 Evaluation methodology

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 (768GB on each NUMA node). The machine has a total DRAM size of 768GB, with two A100 and two A5000 Nvidia GPUs. The server is running Linux kernel v5.13.13 and hyper-threading is disabled.

OdinFS configuration and target comparisons. Unless otherwise mentioned, we configure OdinFS to run on all eight NUMA nodes with twelve delegation threads on each NUMA node. We evaluate and compare OdinFS with four PM file systems: `ext4` [2], `PMFS` [17], `NOVA` [41], and `WineFS` [25]. We configure `ext4` with the DAX option and all the other file systems with the default setup. They provide weaker or the same level of consistency as OdinFS (§3.7). Since these four file systems operate on a single PM NUMA node, we further include one setup: `ext4(RAID0)`. Specifically, we create a RAID0 across all eight PM NUMA nodes using dm-stripe [1] and mount `ext4` on top of it. We cannot run RAID0 with the other file systems because unlike `ext4`, other PM file systems access the PM storage device by memory mapping it into the kernel address space and accessing it with load and store. The RAID0 device created by PM block devices does not support memory mapping to the kernel address space. Because of this, existing PM file systems crash at the time of mounting.

To the best of our knowledge, `ext4(RAID0)` is the only available setup that utilizes all the PM NUMA nodes.[3] However, we further emulate a non-existent setup: `NOVA(MN)` (`NOVA` with multiple nodes) to estimate the performance of a NUMA-aware `NOVA`. Specifically, we mount a single instance of the `NOVA` file system on each NUMA node and evenly distribute the testing files among instances.

Workload. Our workloads include a wide range of file system use cases, covering both data- and metadata-intensive ones. For microbenchmarks, we chose `fio` [4] and FxMark [31] to measure throughput, latency, and scalability, respectively. We configure `fio` to let each thread access a 1GB private file. We only show the results of sequential access due to space limitations and confirm that random access yields similar results. We evaluate `fio` with both small (4KB) and large (2MB) access sizes. For macrobenchmarks, we use Webserver, Fileserver, Videoserver, and Varmail in Filebench [3] and DNN checkpointing.

[3] `WineFS` crashed when mounting on multiple PM NUMA nodes on our server.

5.2 Throughput and latency

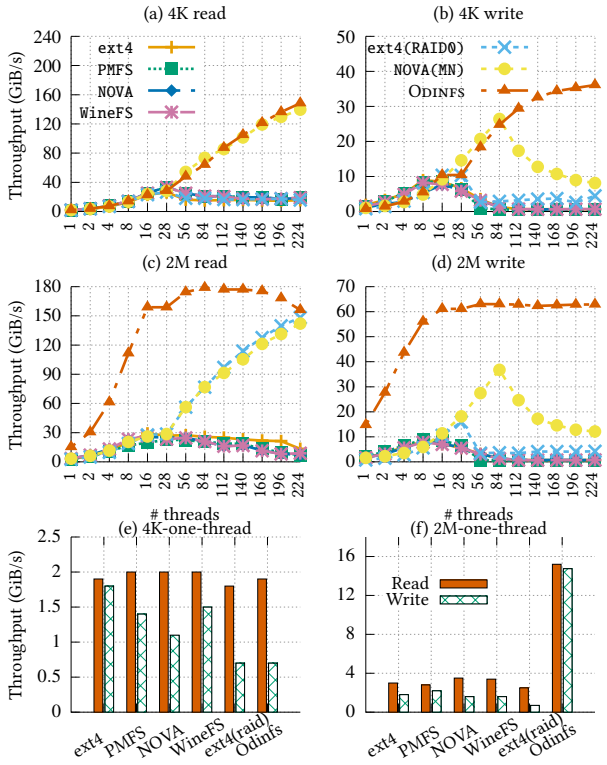

Figure 6: Throughput of evaluated file systems with up to 224 threads. OdinFS scales and outperforms others by up to 24.7×, as it utilizes all PM NUMA nodes and controls concurrent PM accesses.

Throughput. We use `fio` to evaluate OdinFS's throughput and latency. Figure 6 shows the throughput of all evaluated file systems. For `4K-read`, when the thread count is low ($\leq$ 28), all the evaluated file systems perform similarly. However, 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 less than eight, OdinFS suffers from the communication overhead due to delegation and is up to 62% slower than other file systems. However, when the thread count reaches a certain threshold, the throughput of `ext4`, `PMFS`, and `NOVA` starts to collapse due to the reducing efficiency of XPBuffer and XPPrefetcher (§2.2). Instead, 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. As a result, OdinFS outperforms other file systems even with a

low thread count. OdinFS allows applications to utilize most PM bandwidth with eight threads for write and 28 threads for read. OdinFS similarly scales both read and write throughput, outperforming other file systems by 1.1× to 24.7×, and up to 14.8× for `2M-read`, and `2M-write`, respectively. The throughput drop in OdinFS with `2M-read` is likely due to the increasing contention in the ring buffer or the shared counters §4. We plan to address this issue by investigating mechanisms to further increase the scalability of the communication mechanisms.

OdinFS scales because (1) OdinFS utilizes all the PM NUMA nodes in the system , and (2) it limits the number of PM access threads to avoid the performance collapse. `ext4(RAID0)` and `NOVA(MN)` similarly utilize all the PM NUMA nodes and thus performs closest to OdinFS. However, `ext4(RAID0)` and `NOVA(MN)` only scale read operations. `ext4(RAID0)` cannot scale `4K-read` due to a scalability bottleneck in small reads (§5.5). With `2M-read`, they only reach the throughput of OdinFS with a high thread count.

Summary: For small I/O requests, OdinFS incurs overhead with a small thread count but preserves PM performance with a large thread count. For large I/O requests, OdinFS benefits from handling them by paralleling accesses to PM NUMA nodes. This allows an application to utilize most of the PM bandwidth even if it has a small thread count. In summary, OdinFS scales both read and write operations for both small and large I/O sizes.

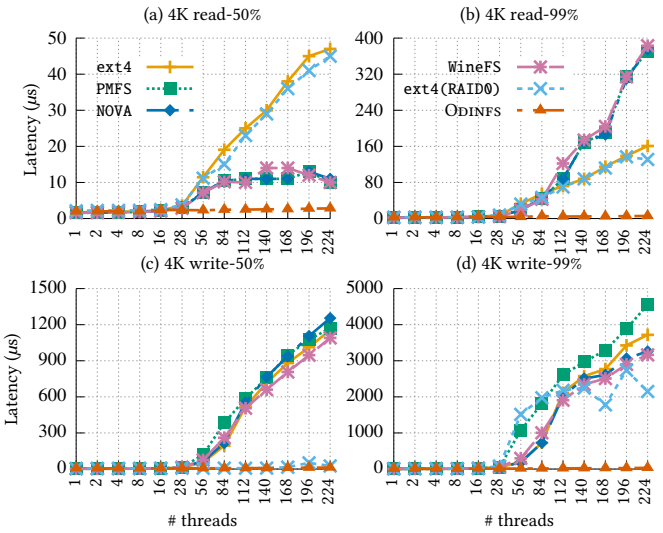

Figure 7: The median and 99 percentile latency of the evaluated file systems with a 4KB access size. OdinFS constantly maintains the low latency due to delegating PM accesses, avoiding the performance collapse within and across NUMA nodes.

Latency. Figure 7 presents the mean and the 99 percentile latency of all the evaluated file systems. For all the other file systems, increasing threads lead to either contention on PM or suffering from PM NUMA impact, resulting in skyrocketing latency. Thanks to delegation, OdinFS constantly maintains low latency. Its median and 99 percentile are 2.8µs and 5.7µs for `4K-read`, 5.4µs to 12.0µs and 6.3µs to 32.4µs for `4K-write`, respectively, outperforming the other file systems by up to 190×. OdinFS consistency has lower latency than `ext4(RAID0)`. The lowest median latency of other file systems is 1.6µs for `4K-read` and 2.6µs for `4K-write` with one thread. However, their latency quickly worsens after sixteen read threads and eight write threads. With a 2MB access size, the trend is similar. The performance advantage of OdinFS is even higher due to parallelization.

Summary : Delegating PM accesses enables OdinFS to maintain low latency.

5.3 IO amplification

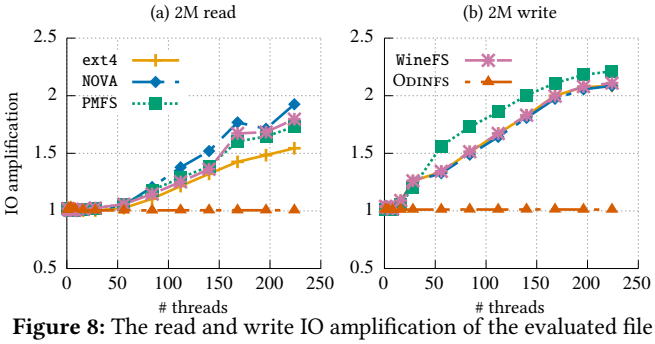

Figure 8: The read and write IO amplification of the evaluated file systems. OdinFS achieves low IO amplification since delegation maintains the caching/prefetching efficiency.

The conventional wisdom is that IO amplification is relevant for traditional storage devices (especially SSD) but not for PM since it is byte-addressable. However, due to the access size mismatch between the memory controller and the PM storage media, PM also suffers from IO amplification if the internal caching/prefetching becomes inefficient (§2.2). A high IO amplification reduces the PM lifetime and causes a latency spike triggered by the internal wear-leveling operation [43]. Thus, a PM file system must reduce it.

We report the I/O amplification as the number of bytes read from (or written to) the underlying PM media divided by the number of bytes requested (or issued) by the CPUs. We use Intel PMWatch [9] to obtain the relevant data. Figure 8 shows the I/O amplification for different file systems with the same setup in §5.2. OdinFS constantly achieves a low IO amplification (*i.e.*, less PM-level IO incurred for the same workloads) with increasing threads since delegation limits concurrent accesses and thus preserves the caching/prefetching efficiency. All other file systems suffer from a high IO amplification rate (*i.e.*, more PM-level IO incurred for the same workload), validating their low throughput (Figure 6) and high latency (Figure 7).

Summary : I/O amplification is still relevant for PM. OdinFS maintains a balance of amplification and high PM utilization. Our delegation scheme limits concurrent accesses, which maintains the caching/prefetching efficiency.

5.4 Sensitivity analysis

This section presents how delegation thread counts and PM NUMA nodes affect OdinFS's performance. We use the same experimental setup in §5.2.

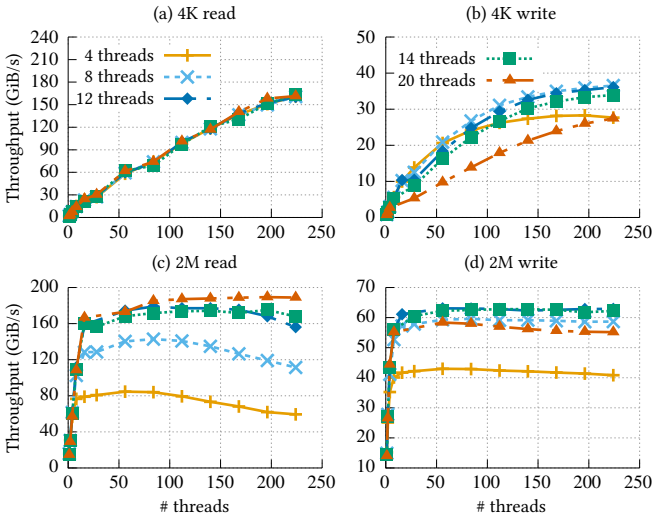

Figure 9: OdinFS's throughput with different delegation threads.

OdinFS with varying delegation threads. The optimal number of delegation threads for OdinFS depends on many factors, such as the relative speed between the processor and PM. Thus, we run experiments that vary the delegation thread counts to find the optimal one for our system.

Figure 9 shows the results. With `4K-read`, the delegation thread counts have no impact because OdinFS does not delegate read accesses (§3.5). With `2M-read`, the throughput is close to being saturated with twelve delegation threads but continues to increase until twenty delegation threads. With `4K-write` and `2M-write`, the throughput of OdinFS increases with up to eight or twelve delegation threads, respectively. Hence, we chose twelve delegation threads as the default setup for OdinFS since it performs well in all four setups.

Summary : The optimal delegation thread number in OdinFS depends on many factors and thus should be decided with experiments. Twelve delegation threads achieve a balanced performance in our system.

OdinFS with varying PM NUMA nodes. Figure 10 shows OdinFS's throughput with a different number of PM NUMA nodes. For `4K-read`, OdinFS enables delegation to prevent throughput collapse after 56 threads with one PM NUMA node and 112 threads with two PM NUMA nodes (§3.5). For the other three setups, OdinFS always delegates PM accesses. The results show that (1) OdinFS can maintain the throughput with a high thread count for different numbers of PM NUMA nodes, and (2) OdinFS scales PM performance with increasing PM NUMA nodes.

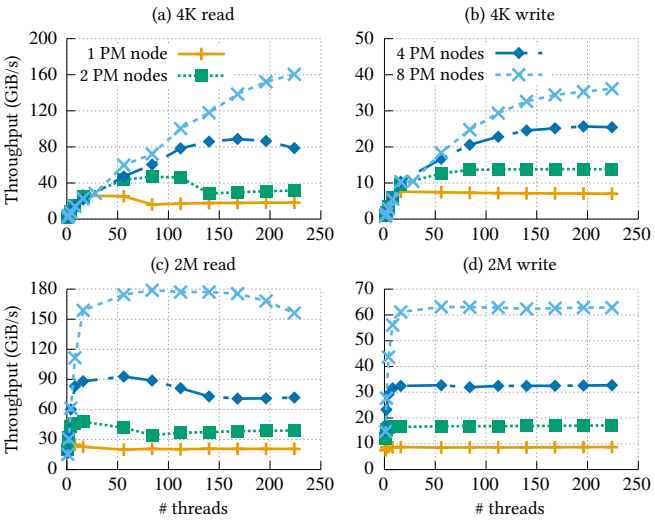

Figure 10: OdinFS with different numbers of PM NUMA nodes.

Name	Description
`DRBL`	Each thread reads a private block in a private file.
`DRBM`	Each thread reads a private block in a shared file.
`DRBH`	Each thread reads a shared block in a shared file.
`DWOL`	Each thread overwrites a private block in a private file
`DWSL`	Same as `DWOL` plus an `fsync()` after each writes
`DWAL`	Each thread appends to a private file
`DWOM`	Each thread writes to a private block in a shared file

Table 1: Summary of microbenchmarks in the FxMark suites [31]. Each thread repetitively performs the corresponding operations in each microbenchmark.

Summary : OdinFS scales PM performance with increasing PM NUMA nodes because of its efficient delegation scheme, showing the generality of its design.

5.5 Datapath scalability

To test whether OdinFS achieves the scalability design goal, we evaluate it with FxMark [31] microbenchmark suites. OdinFS mainly focuses on data operations and partially reuses the scalable data structures in `NOVA` and `WineFS` for metadata scalability. Hence, we focus on evaluating the scalability of data operations. Table 1 summarizes the FxMark microbenchmarks used in the evaluation. We use all the data operation microbenchmarks from FxMark except `DWTL`, where each thread concurrently truncates a private file; `DWTL` does not involve typical data operations (*i.e.*, read or write), and thus we view it as a metadata microbenchmark.

Figure 11 shows the scalability results of the evaluated file systems. Among the compared file systems, only `PMFS` and `NOVA` can scale one microbenchmark: `DRBL`. Instead, OdinFS scales all seven evaluated microbenchmarks. For read microbenchmarks, OdinFS is 12% slower than `PMFS` in `DRBL`. However, OdinFS outperforms other file systems by around 233× and 269× in `DRBM` and `DRBH`, respectively. For `DRBM` and `DRBH`, all other evaluated file systems suffer from the

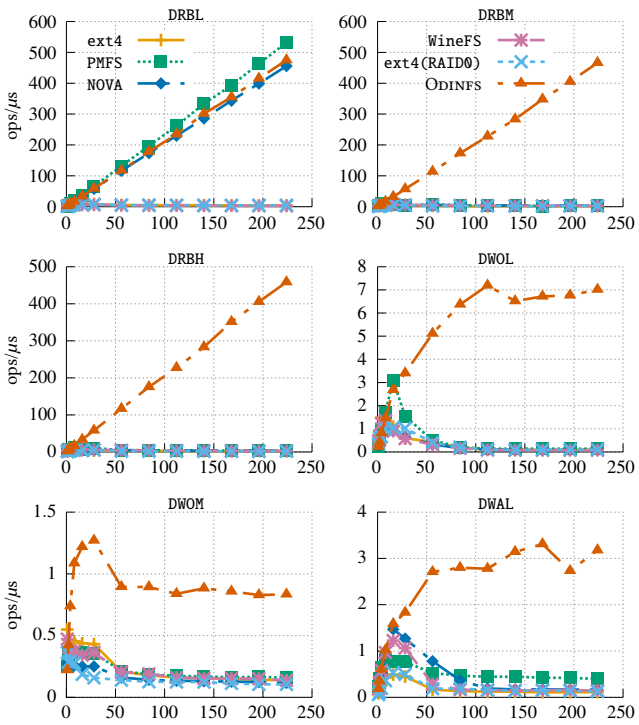

Figure 11: Scalability of data operations with the evaluated file systems. DWSL is not shown since its result is the same as DWOL. Other evaluated file systems only scale DRBL while OdinFS scales all microbenchmarks, thanks to controlling PM accesses, scalable metadata structures and concurrency control mechanisms, and the unique readers-writer range lock.

default readers-writer semaphore implementation in the Linux kernel, resulting in high locking overhead. The scalability of OdinFS comes from the unique scalable synchronization mechanisms (per-CPU readers-writer semaphore and BRAVO on top of the range lock) that minimize the synchronization overhead (§3.6, §4).

For write microbenchmarks, OdinFS outperforms other file systems by 53.8×, 8.2×, and 8.3× in DWOL (DWSL)[4], DWAL, and DWOM, respectively. OdinFS scales DWOL and DWSL due to arbitrating PM accesses and thus prevents performance collapse caused by concurrent writes. OdinFS scales DWOL due to arbitrating PM accesses and the scalable allocator design. In addition to arbitrating PM accesses and the scalable allocator, OdinFS scales DWOM with the readers-writer range lock.

Summary : While other evaluated file systems only scale DRBL, OdinFS scales all seven evaluated microbenchmarks with PM access control, scalable metadata structures, scalable concurrency control mechanisms, and the unique readers-writer range lock.

Name	# Files	Avg. file size	I/O size (r/w)	R/W
Fileserver	10K	2MB	1MB / 256KB	1:2
Webserver	20K	4MB	1MB / 256KB	10:1
Videoserver	226	512MB	1MB / 1MB	27:1
Varmail	100K	16KB	1MB / 16KB	1:1

Table 2: Configuration of the Filebench workloads. Fileserver, Webserver, and Videoserver is data-intensive with large I/Os. Varmail is metadata-intensive with small I/Os, representing the worst case for OdinFS. Webserver and Varmail are write-intensive while Fileserver and Videoserver are read-intensive.

5.6 Macrobenchmarks

We use a set of benchmarks from Filebench [3] as macrobenchmarks to evaluate OdinFS. We select four benchmarks: Fileserver, Webserver, Videoserver, and Varmail with configurations shown in Table 2. We configure Fileserver, Webserver, and Videoserver to work on relatively large files, reflecting the trend of growing sizes with these types of files. Varmail works on a large number of small files and performs small IO, representing the worst case for OdinFS. Webserver and Videoserver are read-intensive while Fileserver and Varmail are write-intensive. For Videoserver, since not all the threads are doing the same task, we measure the overall read and write throughput. For the other benchmarks, we measure the number of operations per second.

Figure 12 shows the result. For Fileserver, OdinFS outperforms other file systems by 4.8× to 25.3×. For Webserver, OdinFS outperforms all the single PM file systems and ext4(RAID0) by at least 3.8× and 1.6× to 3.1×, respectively. For Videoserver, OdinFS outperforms single PM file systems by around 6.6× and at least 5.4× for read and write throughput, respectively. OdinFS outperforms ext4(RAID0) by up to 2.3× for read throughput and around 7.3× for write. For these benchmarks, OdinFS's performance advantage comes from delegating PM accesses to preserve the maximum performance and utilizing the bandwidth of all the PM NUMA nodes. For read-intensive benchmarks: Webserver and Videoserver, ext4(RAID0)'s performance matches OdinFS with large thread counts, which is consistent with results in §5.2. However, OdinFS still outperforms ext4(RAID0) by 1.6× with 224 threads for Webserver. ext4(RAID0) achieves the same read throughput for Videoserver with high thread counts. However, this is because OdinFS still maintains around 4GiB/s write throughput while ext4(RAID0) completely starves the write threads.

Varmail is the worst case for OdinFS since delegating small I/Os incurs large communication overhead. However, the results show that OdinFS can maintain the similar performance as NOVA and WineFS. OdinFS outperforms PMFS, ext4, and ext4(RAID0) by 6.0× to 32.7×. The performance advantage of OdinFS, NOVA, and WineFS comes from the scalable

[4]The DWSL result is the same as DWOL since all evaluated file systems treat fsync() as a no-op.

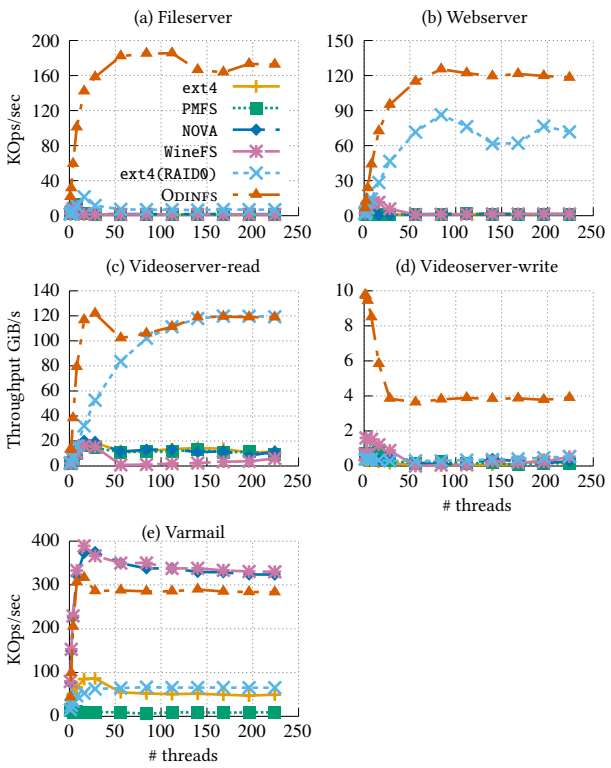

Figure 12: Results of the Filebench benchmarks. See Table 2 for configurations. Results show that OdinFS continue to scale PM performance with macrobenchmarks. OdinFS behaves the same as NOVA in the worst-case scenario: Varmail.

metadata structures (*e.g.*, in-memory directory indexing and per-CPU inode table).

Model	VGG16-ImageNet1K	BERT-SQuAD
Frequency (Steps/ckpt)	34	86
Checkpoint Size (MB)	1055	3828

Table 3: Machine learning checkpointing workloads setup. We use the same frequencies as [33]. A step refers to a training mini-batch.

Machine learning checkpointing. We also evaluate OdinFS with deep neural networks (DNN) checkpointing. DNN training is a time-consuming process and thus must checkpoint its state into persistent storage for fast failure recovery [33]. PM allows high frequency checkpointing and thus minimizes the window of losing work. Table 3 lists models and datasets we use. We measure the end-to-end execution time of training one epoch with checkpointing and the time spent in the file systems. Figure 13 shows the result. OdinFS results in end-to-end execution time reduction over the evaluated file systems by at least 2.6% on VGG16 and 12.3% on BERT. When looking into the time spent in the file systems, OdinFS outperforms evaluated file systems by at least 3.9× on VGG16 and 5.7× on BERT.

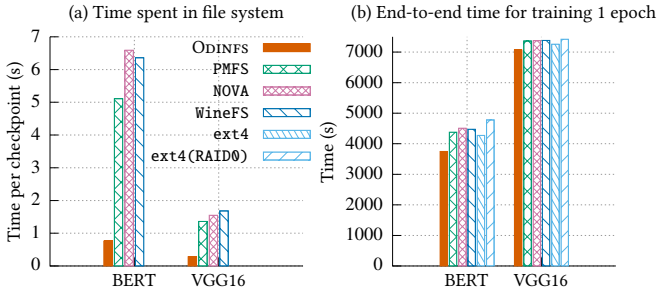

Figure 13: Machine learning checkpointing benchmark.

Summary : OdinFS continues to scale PM performance with macrobenchmarks. OdinFS achieves similar performance to state-of-the-art PM file systems in its worst-case scenario.

6 Discussion

6.1 PM I/O scheduling

The Videoserver result in §5.6 indicates that PM file systems similarly need to employ I/O scheduling as traditional file systems do to, for example, ensure fairness among applications [21, 44]. An I/O scheduling algorithm can be implemented in the system call, page cache, or block layer. Prior research has shown that an I/O scheduling algorithm is more effective if implemented across the various layers of the storage stack [44]. Existing PM file systems [2, 17, 25, 41] bypass the page cache and access PM directly with loads and stores. Hence, they can only implement the I/O scheduling algorithm in the system call layer, limiting the effectiveness of the scheduling algorithm.

Since OdinFS forces application threads to delegate bulk PM accesses to the delegation threads, the delegation threads thus become a central entity to access PM, acting similarly to the block layer in the traditional storage stack. Hence, the delegation enables OdinFS to perform I/O scheduling in both the system call layer and block layers. Exploiting this unique feature, we plan to extend OdinFS to support various I/O scheduling algorithms to ensure fairness or prevent head-of-line blocking to further improve its performance.

6.2 Comparison against RAID0

OdinFS's data layout policy is similar to RAID0 in that it stripes data across multiple PM NUMA nodes in a round-robin manner (§3.4). However, unlike RAID0, OdinFS employs PM-specific optimizations for small files (*i.e.*, put the first file stripe in the local PM of the creation threads). Furthermore, OdinFS stripes in the file level while RAID0 stripes in the disk level. OdinFS thus maximizes parallelization by ensuring adjacent file stripes are highly unlikely in the same PM NUMA node.

A fundamental difference between OdinFS and RAID0 is that OdinFS delegates PM access. As discussed across the paper, this enables OdinFS to avoid the bandwidth collapse with many access threads, minimize the PM NUMA impact,

and access PM in parallel, thus maximizing the PM performance. A typical RAID0 implementation achieves none of the above tasks. As a result, OdinFS consistently outperforms `ext4(RAID0)` with the evaluated benchmarks (§5).

6.3 Applicability to future PM hardware

Two of OdinFS's design goals: localized PM access and access arbitration, are based on the current implementation of PM hardware (§2). Although there is a possibility that future PM hardware may mitigate the above issues, we believe that many of OdinFS' designs will remain effective. Specifically, the PM NUMA impact is mostly due to the implementation of directory coherence information. Some of the recent Intel two-socket machines support snoop protocol for PM, and a prior work [27] has reported that the snoop protocol can significantly mitigate the PM NUMA impact. However, we find that many (large) multi-socket machines, only support directory coherence protocol. Hence, we believe that localized PM access is still a practically important design consideration.

Excessive concurrent access leads to performance collapse since it renders the on-DIMM caching and prefetching inefficient (§2.2). Furthermore, the access size mismatch between the CPU (64 bytes) and the underlying storage media (256 bytes) exacerbates the performance collapse. It also reduces the lifetime of the PM device due to the incurred I/O amplification. While it is difficult to predict the feasibility of changing the PM access sizes in future PM hardware, we expect that such changes would be non-trivial, especially making the PM access size as small as the CPU access size. This would require changes to other critical components in a PM DIMM, such as the address indirection table (AIT) and the DIMM-level prefetching logic. Furthermore, despite the reduced PM access size, limiting concurrent access might still be needed to avoid the performance degradation caused by DIMM-level cache thrashing.

In summary, we expect that both localized PM access and access arbitration will still be relevant for future PM hardware. Moreover, since OdinFS only incurs a small delegation overhead, it provides a "cost-effective" solution to the above problems without hardware changes. In addition, OdinFS' automatic parallelization design will remain useful to utilize the aggregated PM bandwidth across NUMA nodes without modifying the application code. We believe that the automatic parallelization design can be further generalized to other present or future storage systems (*e.g.*, CCIX-based storage systems [13]).

7 Limitations

Extra CPU usage. OdinFS's design incurs additional CPU usage due to parallelizing large PM accesses and the communication between the application thread and the delegation thread. OdinFS's current design reduces the CPU usage by pausing the delegation threads if there is no incoming request (§3.5). In addition, OdinFS can further trim down the CPU usage by (1) offloading PM accesses to I/OAT DMA and (2) disabling the delegation when there is no idle CPU on one NUMA node.

Stripping overhead. OdinFS stripes the data of a file across all NUMA nodes so that even a single-threaded application can benefit from the aggregated PM bandwidth through automatic parallelization (§3.2). However, since the stripping often involves remote access, while the delegation mechanism already significantly mitigates the NUMA impact, the stripping may reduce the best-case throughput and latency. Specifically, without stripping, a best-case scenario occurs when the application and the PM data are in the same NUMA node. However, benefiting from such a scenario requires extra code development to remember the NUMA node where the data resides and pin application threads to the NUMA node, which also limits scheduling flexibility.

If an application does not benefit from the automatic parallelization and wishes to enjoy the best-case performance, we expect OdinFS can work without stripping by placing the data of a file on a single NUMA node. In this case, OdinFS still achieves the other two design goals: (1) limiting concurrent accesses and (2) minimizing the PM NUMA impacts. Large I/O accesses can still be parallelized to one NUMA node but not all NUMA nodes as before.

Memory mapping. Due to stripping, OdinFS's memory mapping (mmap) performance is lower than other single-NUMA-node PM file systems in the best-case scenario as described above. To optimize this, OdinFS can use a copy-then-mmap model similar to `NOVA` [41]. Specifically, upon `mmap`, OdinFS allocates PM pages in the same NUMA node, copies the file content in remote NUMA nodes to these pages, and `mmap` the PM pages to the applications. Upon `msync` or `munmap`, OdinFS propagates the changes in the replicated PM pages back to the files.

8 Related work

PM file system. Unlike OdinFS, most existing PM file systems are designed to work on a single PM NUMA node and do not limit the number of access threads [12, 16, 17, 24, 28, 34, 41], leading to PM performance collapse. In terms of NUMA-aware PM file systems, Xu *et al.* proposed a new `ioctl` command that allows applications to specify the preferred NUMA node of a file [42]. This approach requires application changes and relies on the application to avoid the NUMA impact. `WineFS` [25] assigns a home NUMA node to each application thread and migrates the thread to the home NUMA node before writing to PM. As acknowledged by the authors, thread migration is expensive. Furthermore, it still suffers from the NUMA impact when two threads from different home nodes share the same file. Unlike OdinFS, none of these works focuses on utilizing both local and remote PM simultaneously as OdinFS does. `ext4(RAID0)` [1, 2] does not control the number of access threads nor resolve the PM NUMA impact, leading to lower performance than OdinFS in most cases.

Other NUMA-aware PM systems. PACTree uses the snoop protocol to minimize the PM NUMA impact [27]. However, the snoop protocol is unlikely to scale on large NUMA machines and may thus impact the performance of memory-intensive applications. Nap caches frequently accessed items in DRAM to avoid remote PM accesses [38]. However, Nap relies on a skewed access pattern to benefit from caching and still suffers from the PM NUMA impact upon cache misses. OdinFS proposes a fundamental different approach that uses delegation to address the PM NUMA impact.

Scalable file system. There are scalable file systems for both traditional storage devices [11, 30, 36] and PM [12, 41]. NOVA designs several scalable metadata structures, and OdinFS inherits them. Similar to OdinFS, there are file systems scaling the data operations with range locks [12, 36]. The closest work to OdinFS is KucoFS [12]. KucoFS is a PM file system that scales metadata operations through bypassing the VFS and scales data operations with range locks and versioned read. However, KucoFS shows that it cannot scale data operation benchmarks in FxMark beyond fifteen threads, while OdinFS scales all of them up to 224 threads. The difference is that (1) OdinFS uses delegation to prevent PM performance collapse while minimizing the NUMA impact. (2) OdinFS uses state-of-the-art concurrency mechanisms which minimize the synchronization overhead.

Localized I/O threads. Since PCIe devices also conform to NUMA topology, utilizing localized I/O threads (*i.e.*, placing I/O threads in the same NUMA code as the I/O devices) is a relatively common design in many non-PM systems [22, 35, 45]. To realize the old wisdom in PM systems, OdinFS has encountered and resolved many unexplored challenges (§3, §4), leading to a design significantly departs from the other PM systems (§3.2).

9 Conclusion

This paper presents OdinFS, a file system that maximizes PM performance in NUMA machines. A key novelty in OdinFS lies in decoupling the PM data accesses from the application threads by offloading them to a set of delegation threads in each NUMA node. Such decoupling simultaneously allows OdinFS to preserve the maximum PM performance with a single NUMA node, efficiently utilize PM in remote NUMA nodes, and service system calls by accessing PM in all NUMA nodes in parallel, thus maximizing the PM performance. OdinFS further includes fine-grained synchronization control mechanisms to scale all typical file system data operations. Extensive evaluation shows that OdinFS constantly outperforms existing PM file systems by several times to orders of magnitude.

Acknowledgments

We sincerely thank our shepherd Oana Balmau and the anonymous reviewers for their insightful feedback. Yunxin Sun contributed to the evaluation. This work was in part supported by Institute for Information and communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No. 2014-3-00035).

References

[1] Device Mapper. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/striped.html.

[2] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[3] Filebench - A Model Based File System Workload Generator . https://github.com/filebench/filebench.

[4] Flexible I/O Tester. https://github.com/axboe/fio.

[5] Directory Structure in Skylake Server CPUs, . https://community.intel.com/t5/Software-Tuning-Performance/Directory-Structure-in-Skylake-Server-CPUs/td-p/1185376.

[6] Intel® 64 and IA-32 Architectures Optimization Reference Manual, . https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[7] Intel Processor Counter Monitor (PCM), . https://github.com/opcm/pcm.

[8] Intel Optane Persistent Memory, . https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[9] Intel PMWatch. https://github.com/intel/intel-pmwatch.

[10] T. E. Anderson, M. Canini, J. Kim, D. Kostic, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, Nov. 2020.

[11] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[12] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, Feb. 2021.

[13] T. C. Consortium. Home Is Where the Memory Is, 2022. https://www.ccixconsortium.com/home-is-where-the-memory-is/.

[14] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 ACM SIGMOD/PODS Conference*, Xi'an, Shaanxi, China, May 2021.

[15] D. Dice and A. Kogan. BRAVO: Biased Locking for Reader-Writer Locks. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[16] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen. Performance and Protection in the ZoFS User-space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[17] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[18] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali. Single Machine Graph Analytics on Massive Datasets using Intel Optane DC Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, Aug. 2020.

[19] S. Gugnani, A. Kashyap, and X. Lu. Understanding the Idiosyncrasies of Real Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, Aug. 2020.

[20] T. Hirofuchi and R. Takano. The Preliminary Evaluation of a Hypervisor-based Virtualization Mechanism for Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1907.12014*, 2019.

[21] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[22] Intel. Storage Performance Development Kit, 2021. SPDK.io.

[23] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714*, 2019.

[24] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[25] R. Kadekodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. R. Ganger, A. Kolli, and V. Chidambaram. 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.

[26] S. Kashyap, C. Min, and T. Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[27] W.-H. Kim, R. M. Krishnan, X. F. S. Kashyap, and C. Min. 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.

[28] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[29] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, Los Angeles, CA, Aug. 2019.

[30] X. Liao, Y. Lu, E. Xu, and J. Shu. Max: A Multicore-Accelerated File System for Flash Storage. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, Virtual, July 2021.

[31] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[32] C. Min, W.-H. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. SOLROS: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, Porto, Portugal, Apr. 2018.

[33] J. Mohan, A. Phanishayee, and V. Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, Feb. 2021.

[34] I. Neal, G. Zuo, E. Shiple, T. A. Khan, Y. Kwon, S. Peter, and B. Kasikci. Rethinking File Mapping for Persistent Memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, Feb. 2021.

[35] T. L. F. Projects. DPDK, 2021. https://www.dpdk.org/.

[36] Y. Ren, C. Min, and S. Kannan. CrossFS: A Cross-layered Direct-Access File System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, Nov. 2020.

[37] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent Memory I/O Primitives. In *Proceedings of the International Workshop on Data Management on New Hardware*, Amsterdam, The Netherlands, July 2019.

[38] Q. Wang, Y. Lu, J. Li, and J. Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[39] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Virtual, Oct. 2020.

[40] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Denver, CO, Nov. 2019.

[41] 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.

[42] J. Xu, J. Kim, A. Memaripour, and S. Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the 23th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[43] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. 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.

[44] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Split-Level I/O Scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[45] D. Zheng, R. Burns, and A. S. Szalay. Toward Millions of File System IOPS on Low-Cost, Commodity Hardware. In *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Denver, CO, Nov. 2013.