Scalable Persistent Memory File System with Kernel-Userspace Collaboration

Youmin Chen, Youyou Lu, Bohong Zhu,
Andrea C. Arpaci-Dusseau[†], Remzi H. Arpaci-Dusseau[†], Jiwu Shu[*]
Tsinghua University [†] *University of Wisconsin – Madison*

Abstract

We introduce KUCO, a novel direct-access file system architecture whose main goal is scalability. KUCO utilizes three key techniques – collaborative indexing, two-level locking, and versioned reads – to offload time-consuming tasks, such as pathname resolution and concurrency control, from the kernel to userspace, thus avoiding kernel processing bottlenecks. Upon KUCO, we present the design and implementation of KUCOFS, and then experimentally show that KUCOFS has excellent performance in a wide range of experiments; importantly, KUCOFS scales better than existing file systems by up to an order of magnitude for metadata operations, and fully exploits device bandwidth for data operations.

1 Introduction

Emerging byte-addressable persistent memories (PMs), such as PCM [22, 34, 51], ReRAM [3], and the recently released Intel Optane DCPMM [27], provide performance close to DRAM and data persistence similar to disks. Such high-performance hardware increases the importance of redesigning *efficient* file systems. In the past decade, the systems community has proposed a number of file systems, such as BPFS [11], PMFS [14], and NOVA [43], to minimize the software overhead caused by a traditional file system architecture. However, these PM-aware file systems are part of the operating system and applications need to trap into the kernel to access them, where system calls (syscalls) and the virtual file system (VFS) still incur non-negligible overhead. In this regard, recent work [13, 21, 28, 39] proposes to deploy file systems in userspace to access file data directly (i.e., direct access), thus exploiting the high performance of PM.

Despite these efforts, we find that another important performance metric – *scalability* – still has not been well addressed, especially when multicore processors meet fast PMs. NOVA [43] improves multicore scalability by partitioning internal data structures and avoiding using global locks. However, our evaluation shows that it still fails to scale well due to the existence of the VFS layer. Even worse, some userspace file system designs further exasperate the scalability problem by introducing a centralized component. For example, Aerie [39] ensures the integrity of file system metadata by sending expensive inter-process communications (IPCs) to a trusted process (TFS) that has the authority to update metadata. Strata [21], as another example, avoids the involvement of a centralized process in normal operations by directly recording updates in PM logs, but requires a KernFS to apply them (including both data and metadata) to the file system, which causes one more time of data copying. The trusted process (e.g., TFS or KernFS) in both file systems is also responsible for concurrency control, which inevitably becomes the bottleneck under high concurrency.

In this paper, we revisit the file system design by introducing a *kernel-userspace collaboration* architecture, or KUCO, to achieve both direct access performance and high scalability. KUCO follows a classic *client/server model* with two components, including a userspace library (named Ulib) to provide basic file system interfaces, and a trusted thread (named Kfs) placed in the kernel to process requests sent by Ulib and perform critical updates (e.g., metadata).

Inspired by distributed file system designs, e.g., AFS [17], that improve scalability by minimizing server loads and reducing client/server interactions, KUCO presents a novel task division and collaboration between Ulib and Kfs, which offloads most tasks to Ulib to avoid a possible Kfs bottleneck. For metadata scalability, we introduce a *collaborative indexing* technique to allow Ulib to perform pathname resolution before sending requests to Kfs. In this way, Kfs can update metadata items directly with the pre-located addresses provided by Ulib. For data scalability, we first propose a *two-level locking* mechanism to coordinate concurrent writes to shared files. Specifically, Kfs manages a write lease for each file and assigns it to the process that intends to open the file. Instead, threads within this process lock the file with a range-lock completely in userspace. Second, we introduce a *versioned read protocol* to achieve direct reads even without interacting with Kfs, despite the presence of concurrent writers.

KUCO also includes techniques to enforce data protection and improve baseline performance. KUCO maps the PM space into userspace in readonly mode to prevent buggy programs from corrupting file data. Userspace direct writes are achieved with a *three-phase write protocol*. Before Ulib writes a file, Kfs switches the related PM pages from readonly to writeable by toggling the permission bits in the page table. A *pre-allocation* technique is also used to reduce the number of interactions between Ulib and Kfs when writing a file.

With the KUCO architecture, we build a PM file system named KUCOFS, which gains userspace direct-access performance and delivers high scalability simultaneously. We evaluate KUCOFS with file system benchmarks and real-world

[*]Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

applications. The evaluation results show that KucoFS scales better than existing file systems by an order of magnitude under high contention workloads (e.g., creating files in the same directory or writing data in a shared file), and delivers slightly higher throughput under low contention. It also hits the bandwidth ceiling of PM devices for normal data operations. In summary, we make the following contributions:

- We conduct an in-depth analysis of state-of-the-art PM-aware file systems and summarize their limitations on solving the software overhead and scalability problems.
- We introduce Kuco, a *userspace-kernel collaboration* architecture with three key techniques, including *collaborative indexing*, *two-level locking*, and *versioned read* to achieve high scalability.
- We implement a PM file system named KucoFS based on the Kuco architecture, and experimentally show that KucoFS achieves up to one order of magnitude higher scalability for metadata operations, and fully exploits the PM bandwidth for data operations.

2 Motivation

In the past decade, researchers have developed a number of PM file systems, such as BPFS [11], SCMFS [41], PMFS [14], HiNFS [29], NOVA [43], Aerie [39], Strata [21], SplitFS [28], and ZoFS [13]. They are broadly categorized into three types. First, *kernel-level file systems*. Applications access them by trapping into the kernel for both data and metadata operations. Second, *userspace file systems* (e.g., Aerie [39], Strata [21], and ZoFS [13]). Among them, Aerie [39] relies on a trusted process (TFS) to manage metadata and ensure the integrity of it. The TFS also coordinates concurrent reads and writes to shared files with a distributed lock service. Strata [21], in contrast, enables applications to append their updates directly to a per-process log, but requires background threads (KernFS) to asynchronously digest logged data to storage devices. ZoFS avoids using a centralized component and allows userspace applications to update metadata directly with the help of a new hardware feature named Intel Memory Protection Key (MPK). Note that Aerie, Strata, and ZoFS still rely on the kernel to enforce coarse-grained allocation and protection. Third, *hybrid file systems* (e.g., SplitFS [28] and our proposed Kuco). SplitFS [28] presents a coarse-grained split between a user-space library and an existing kernel file system. It handles data operations entirely in userspace, and processes metadata operations through the Ext4 file system. Table 1 provides a summary of existing PM-aware file systems and how well they behave in various aspects.

❶ ***Multicore scalability.*** NOVA [43], a state-of-the-art kernel file system for PMs, is carefully designed to improve scalability by introducing the per-core allocator and per-inode log. Nevertheless, VFS still limits its scalability for certain operations. We experimentally show this by deploying NOVA on Intel Optane DCPMMs (detailed experimental setup is described in § 5.1), and use multiple threads to create, delete, or rename files in the same directory. As shown in Figure 1a, their throughput is almost unchanged as we increase the number of threads, since VFS needs to acquire the lock of the parent directory. Aerie [14] relies on a centralized TFS to handle metadata operations and enforce concurrency control. Although Aerie batches metadata changes to reduce communication with the TFS, our evaluation in §5 shows that the TFS still inevitably becomes the bottleneck under high concurrency. In Strata [21], the KernFS needs to digest logged data and metadata in the background. If an application completely uses up its log, it has to wait for an in-progress digest to complete before it can reclaim log space. As a result, the number of digestion threads limits Strata's overall scalability. Both Aerie and Strata interact with the trusted process (TFS/KernFS) via expensive IPCs, which introduces extra syscall overhead. ZoFS does not require a centralized component, so it achieves much higher scalability. However, ZoFS still fails to scale well when processing operations that require allocating new spaces from the kernel (e.g., `creat` and `append`, see Figures 7d, 7f, and 7g in their paper). Our evaluation shows that SplitFS scales poorly for both data and metadata operations because it 1) does not support sharing between different processes, and 2) relies on Ext4 to update metadata (see Figures 7 and 9).

		NOVA	Aerie/Strata	ZoFS	SplitFS	KucoFS
Category		Kernel	Userspace		Hybrid	
❶ Scalability	Metadata	Medium (§5.2.1)	Low (§5.2.1)	Medium (Fig. 7g in [13])	Low (§5.2.1)	High (§5.2.1)
	Read	Medium (§5.2.2)	Low (§5.2.2)	High	Low (journaling in Ext4)	High (§5.2.2)
	Write	Medium (§5.2.3)	Low (§5.2.3)	Medium (Fig.7f in [13])		High (§5.2.3)
❷	Softeware overhead	High	Low	Medium (`sigsetjump`)	Medium (metadata)	Low
❸ Other issues	Avoid stray writes	✓	✗	✓	✗	✓
	Read protection	POSIX	Partition	Coffer	POSIX	Partition
	Visibility of updates	Immed-iately	After batch/ After digest	Immed-iately	`append`: After sync	Immed-iately
	Hardware required	None	None	MPK	None	None

Table 1: Comparison of different NVM-aware file systems.

❷ ***Software overhead.*** Placing a file system in the kernel faces two types of software overhead, i.e., the syscall and VFS overhead. We investigate such overhead by still analyzing NOVA, where we collect the latency breakdown of common file system operations. Each operation is performed on 1 million files or directories with a single thread. We make two observations from Figure 1b. First, syscalls take up to 21% of the total execution time (e.g., `stat` and `open`). Also, after a process traps into the kernel, the OS may schedule other tasks before returning control to the original one. Hence, syscalls bring extra uncertainty for latency-sensitive applications [12, 33]. Second, Linux kernel file systems are implemented by overriding VFS functions, and VFS causes non-negligible overhead. Although recent PM file

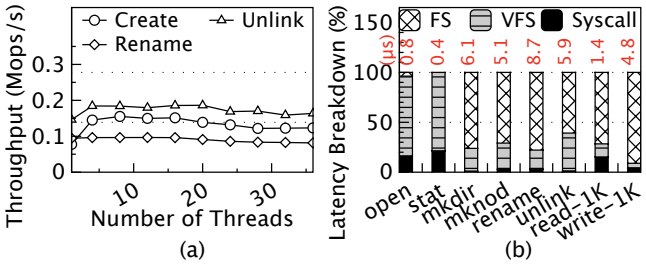

Figure 1: Software overhead and scalability of NOVA.

systems [9, 11, 14, 29, 40, 43, 50] use direct access (DAX) to bypass the page cache in VFS, we find that an average of 34% of the time is still spent in the VFS layer for NOVA. ZoFS [13] deploys a file system in userspace to avoid trapping into the kernel; however, it still incurs extra software overhead. ZoFS allows userspace applications to update metadata directly, which may cause a normal program to be terminated when accessing metadata that is corrupted by malicious attackers. To achieve graceful error return, ZoFS invokes a `sigsetjump` instruction at the beginning of each syscall, which causes extra delays (~200 ns). SplitFS requires a kernel file system to handle metadata operations, so it still introduces kernel overhead.

❸ ***Other issues.*** First, misused pointers can lead to writes to incorrect locations and corrupt the data, which is known as *stray writes* [14]. Strata [21] exposes the per-process operation log and the DRAM cache (including both metadata and data) to userspace applications. Aerie [39] and SplitFS [28] map a subset of the file system image to userspace. Hence, stray writes can easily corrupt the data in these areas, and such corruptions are permanent in NVM even after reboots. Second, Aerie, Strata, and SplitFS improve performance by delaying the visibility of the newly written data to other processes until issuing a `fsync`, forcing applications to make corresponding adjustments. Third, ZoFS heavily relies on the MPK mechanism, if an application also needs to use MPK, they may compete for the limited MPK resources.

To summarize, it is hard to achieve high scalability and low software overhead with existing file system designs, and this motivates us to introduce the Kuco architecture.

3 The Kuco Architecture

In this paper, we introduce the Kuco architecture to show that a *client/server model* can be adopted to realize the two goals simultaneously. The central idea underlying Kuco is a fine-grained task division and collaboration between the *client* and *server*, where most loads are offloaded to the client part to avoid the server from becoming the bottleneck.

3.1 Overview

Figure 2 shows the Kuco architecture. It follows a *client/server model* with two parts, including a userspace library and a global kernel thread, which are called Ulib and Kfs, respectively. An application accesses Kuco by linking with

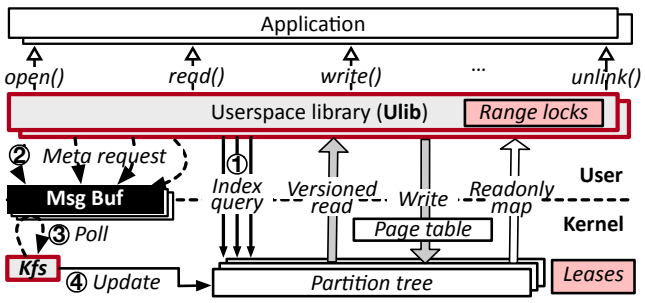

Figure 2: The Kuco architecture. metadata updates (①-④): Ulib interacts with Kfs via *collaborative indexing*; read: direct access via *versioned read*; write: direct access based on a *three-phase write protocol* and *two-level locking* for concurrency control.

Ulib first, and different Ulib instances (i.e., applications) interact with Kfs via separate memory message buffers. Like existing userspace file systems [21, 39], Kuco maps the PM space to userspace to support direct read and write accesses. To protect file system metadata from being corrupted, Kuco does not allow applications to update metadata directly; instead, such requests are posted to Kfs, and Kfs then updates metadata on behalf of them.

Kuco delivers high scalability with a fine-grained task division and collaboration between Ulib and Kfs. For metadata scalability, Kuco incorporates the *collaborative indexing* mechanism to offload the pathname traversal job from Kfs to userspace (§3.2). Instead of sending metadata operations (e.g., `creat` or `unlink`) to Kfs directly, Ulib first finds all the related metadata items in userspace, and then encapsulates such information in the request before sending it out. Therefore, Kfs can perform metadata modifications directly with the given addresses. For data scalability, a *two-level locking* mechanism is used to handle concurrent writes to shared files (§3.3). Specifically, Kfs uses a lease-based distributed lock to resolve write conflicts between different applications (or processes). Concurrent writes from the same process are serialized using a pure userspace range lock, which can be acquired without the involvement of Kfs. Kuco further introduces the *versioned read* technique to perform file reading in userspace (§3.5). By adding extra version bits in data block mappings (which map logical file data to physical PM addresses), Kuco can read a consistent version of data blocks without interacting with Kfs to acquire the lock, despite that there are other concurrent writers.

To further prevent buggy programs from corrupting file data, PM space is mapped to userspace in readonly mode. Kuco enables userspace direct writes on readonly addresses by placing Kfs in the kernel with a *three-phase write protocol* (§3.4). Before Ulib writes a file, Kfs modifies the permission bits in the page table first to switch the involved data pages from readonly to writable. To further reduce the number of interactions between Ulib and Kfs when writing a file, Kuco adopts *pre-allocation*, where Ulib can allocate more free pages from Kfs than desired. Except for the write protection

mechanism that prevents stray writes, the PM space in KUCO is then divided into different partition trees, which act as the minimum unit for read protection. By applying KUCO in a file system named KUCOFS and putting all techniques together, KUCOFS gains direct-access performance, delivers high scalability, and ensures the kernel-level data protection.

3.2 Collaborative Indexing

In a typical *client-server model*, whenever Kfs receives a metadata request, it needs to find the related metadata (e.g., *inodes* that describe file attributes, or *dentries* that map file names to inode numbers) by performing iterative pathname resolution from the root inode to the directory containing this file. Such pathname traversal overhead is a heavy burden for Kfs, especially when a directory contains a large number of sub-files or with deep directory hierarchies.

To address this issue, we propose to offload the pathname resolution task from Kfs to Ulib. By mapping partition trees to userspace, Ulib can find the related metadata items directly in userspace, and then sends a metadata update request to Kfs by encapsulating the metadata addresses in the request as well. In this way, Kfs can update metadata directly with the given addresses, and the pathname resolution overhead is offloaded from Kfs to userspace.

Figure 3 shows how KUCO creates a file with a pathname of "`/Bob/a`". Ulib first finds the predecessor dentry of file "`a`" in the dentry list of "`Bob`" (①). It then sends a `creat` request to Kfs, and the address of the predecessor is put in the message too (②). Kfs then creates the file after receiving the request (③④), which includes creating an inode of this file, and then inserting a new dentry in the parent directory's dentry list with the given predecessor. To delete a file, both the inode of this file and dentry in the parent directory should be deleted, so both of their addresses are kept in the `unlink` request before Ulib sends it. Note that `atime` is disabled by default, enabling readonly operations (e.g., `stat`, `readdir`) to be performed in userspace without posting extra requests to Kfs.

In KUCO, Ulibs produce pointers and Kfs consumes them. This "one-way" pointer sharing paradigm simplifies ensuring the correctness and safety of KUCO. On the one hand, metadata items are placed in a metadata area with separate address space and Ulib can only pass the addresses of two types of metadata items (i.e., dentry and inode). Hence, we add an identifier field at the beginning of each metadata item, which helps Kfs to check the metadata type – any addresses not in the metadata area or not pointing to a dentry/inode is considered invalid. On the other hand, Kfs also performs consistency checking based on the file system internal logic:

First, Ulib might read an inconsistent directory tree. For example, when Kfs is creating new files in a directory, concurrent Ulibs may read an inconsistent dentry list of this directory. To address this issue, we organize the dentry list of each directory with a skip list [32] and each dentry is indexed by the hash value of the file name. Skip list has multiple

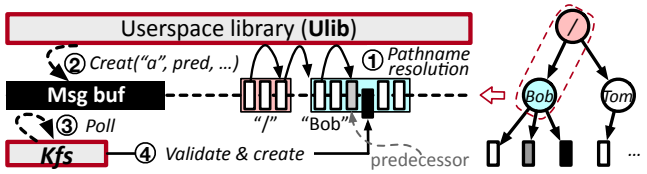

Figure 3: Creating a file (①-④) with *collaborative indexing*.

layers of linked list-like data structure. Each higher layer acts as an "express lane" for the lower list layer. The list-based structure enables lock-free atomic updates by performing pointer manipulations. Besides, there are only insert and delete operations to the dentry list performed by a single Kfs, including rename operations which are performed by first inserting a new node and then deleting the old one. Therefore, a read to a dentry is always performed to a consistent one even without acquiring the lock.

Second, with such a lock-free design, userspace applications may read metadata items that are being deleted by Kfs, causing the "read-after-delete" anomaly. To safely reclaim the deleted items, we need to ensure that no threads access it anymore. We address this issue by using an epoch-based reclamation mechanism (EBR) [15]. EBR maintains a global epoch and three reclaim queues, where the execution is divided into epochs and reclaim queues are maintained for the last three epochs. Each thread also owns a private epoch. Items deleted in epoch *e* are placed into the queue for epoch *e*. Each time Ulib starts an operation, it reads the global epoch and updates its own epoch to be equal to the global one. It then checks the private epochs of others. If all Ulibs are active in the current epoch *e*, then a new epoch begins. At this time, all threads are active either in *e* or in *e+1*, and items in the queue related to *e-1* can be reclaimed safely. We also add a *dirty* flag in each inode/dentry. Kfs deletes a metadata item by setting its *dirty* flag to an invalid state, preventing applications from reading the already deleted items.

Third, Kfs needs to handle conflicting metadata operations properly. For example, when multiple Ulibs are performing metadata operations concurrently, the pre-located metadata item of one Ulib might be deleted or renamed by another concurrent Ulib before Kfs accesses it. Hence, this item is no longer valid and its address cannot be used by Kfs anymore. It is also possible that a malicious process attacks Kfs by providing arbitrary addresses. Luckily, only the Kfs can update metadata, and it can validate the pre-located metadata before processing the operation. Specifically, Kfs checks if the pre-located item still exists or is still the predecessor, and avoids creating files with the same name. When the validation fails, Kfs then resolves the pathname itself and returns an error code to the Ulib if the operation fails anyway.

Discussion. First, KUCO ensures that all metadata operations are processed atomically. For `creat`, Kfs atomically inserts a new dentry in the skip list only after an inode has been created, to make the created file visible; For `unlink`, it

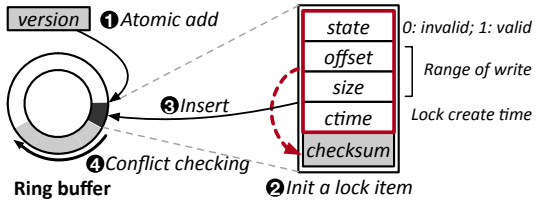

Figure 4: Direct access range-lock. Each opened file owns a range-lock. ❶-❹ show the steps to acquire a lock.

atomically deletes the dentry before deleting other fields. `Rename` involves updating two dentries (create a new entry in the destination path, and then delete the old one), so a program can see two same files on both places at some point in time. We leverage the *dirty* flag in each dentry to prevent such an inconsistent state. Specifically, the old entry on the source path is set to *dirty* before creating the new entry, and is then set to *invalid* after the new entry is created. As a whole, we can observe that metadata operations always change the directory tree atomically, and Ulib is guaranteed to have a consistent view of the directory tree even without acquiring the lock. Second, KUCO's scalability is further improved by avoiding using locks — concurrent metadata updates are all delegated to the global Kfs, so they can be processed without any locking overhead (only Kfs can update metadata) [16, 35]. KUCO ensures the crash consistency of metadata via an operation log, which will be discussed in §4.2.

3.3 Two-Level Locking

KUCO introduces a *two-level locking service* to coordinate concurrent writes to shared files, which prevents Kfs from being frequently involved in concurrency control. First, Kfs assigns *write leases* (in the kernel, see Figure 2) on files to enforce coarse-grained coordination between different processes, as in Aerie and Strata [21, 39]. Only the process that holds a valid write lease (not yet expired) can write the file. We assume that Ulib applies for leases infrequently, and this is based on the fact that it is not the common case for multiple processes to frequently and concurrently write the same file. More fine-grained sharing between processes can be achieved via shared memory or pipes [21]. *Read leases* are not needed in KUCO (see Section 3.5).

Second, we introduce a *direct access range-lock* to serialize concurrent writes between threads within the same process. Once a Ulib acquires the write lease of a file, it creates a range lock for this file in userspace, which is actually a DRAM ring buffer (as shown in Figures 4). A thread writes a file by acquiring the range-lock first, and it is blocked if a lock conflict occurs. Each slot in the ring buffer has five fields, which are state, offset, size, ctime, and a checksum. The checksum is the hash value of the first four fields. We also place a version at the head of each ring buffer to describe the order of each write operation. To acquire the lock of a file, Ulib firstly increments the version with an atomic `fetch_and_add` (i.e., ❶). It then inserts a lock item into a specific slot in the ring buffer (❷ and ❸, the location is determined by the fetched version modulo the ring buffer size). The insertion is blocked when this slot overlaps with the head of the ring buffer. After this, Ulib traverses the ring buffer backward to find the first conflicting lock item (i.e., their written data overlaps). If such a conflict exists, Ulib verifies its checksum, and then polls on its state until it is released. Ulib also checks its ctime field repeatedly to avoid the deadlock if a thread aborts before it releases the lock (❹). With this design, multiple threads can write different data pages in the same file concurrently.

3.4 Three-Phase Write

Once the lock has been required, Ulib can actually write file data. Since PM spaces are mapped to userspace in readonly mode, Ulib cannot write file data directly. Instead, we propose a *three-phase write protocol* to perform direct writes. To ensure the crash consistency, KUCO follows a copy-on-write (CoW) approach to write file data, where the newly written data is always redirected to new PM pages. Similar to NOVA [43] and PMFS [14], we use 4 KB as the default data page size. The write protocol in KUCO consists of three steps. First, Ulib locks the file via *two-level locking* and sends a request to Kfs to allocate new PM pages. Note that, by using a CoW way, space allocation is necessary for both `overwrite` and `append` operations. Kfs also needs to modify the related page table entries to make these allocated PM pages writable before sending the response message back. Second, Ulib copies both the unmodified data from the old place and new data from the user buffer to the allocated PM pages, and persists them via flush instructions. Third, Ulib sends another request to Kfs to update the metadata of this file (i.e., inode, block mapping), switch the newly written pages to readonly, and finally releases the lock.

Furthermore, we introduce the *pre-allocation* mechanism to avoid allocating new PM pages from Kfs for every write operations. Specifically, we allow Ulib to allocate more free pages from Kfs than desired (4 MB at a time in our implementation). In this way, Ulib can use local free PM pages without interacting with Kfs for most write operations. When an application exits, the unused pages are given back to the Kfs. For an abnormal exit, these free pages are temporarily non-reusable by other applications, but still can be reclaimed during the recovery phase (see §4.2). *Pre-allocation* also helps with reducing the overhead of updating page table entries. When the Kfs updates page table entries after each allocation, it needs to flush the related TLB entries explicitly to make the modifications visible. *Pre-allocation* allows allocating multiple data pages at a time, so the TLB entries can be flushed in batch.

3.5 Versioned Read

In the write protocol, both old and new versions of data pages are temporarily kept due to the CoW way, providing us the

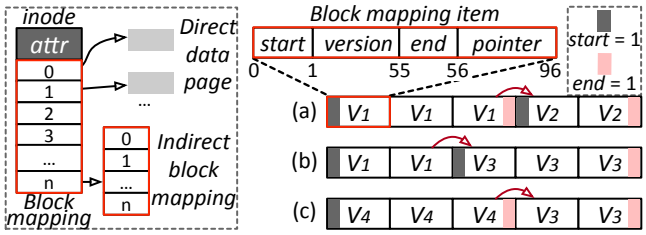

Figure 5: Block mapping format and the versioned read protocol. Mapping items with the same version correspond to the same write operation. The above three consistent cases describe how the *start* and *end* bits can be formatted when the version changes.

opportunity to read file data even without blocking writes. However, block mappings that map a logical file to physical pages are still updated in place by Kfs. This dirves us to design the *versioned read* mechanism to achieve user-level direct reads without any involvement of the Kfs, regardless of concurrent writers.

Versioned Read is designed to allow userspace reads without locking the file, while ensuring that readers never read data from incomplete writes. To achieve this, KUCO uses an Ext2-like [6] block mapping to index data pages and embeds a version field in each pointer of the block mapping. As shown in Figure 5, each 96-bit block mapping item contains four fields, which are start, version, end and pointer. For a `write` operation, say, writing three data pages, Kfs updates the related block mapping items with the following format: 1|V_1|0|P_1| 0|V_1|0|P_2| 0|V_1|1|P_3|. In particular, all three items share the same version (i.e., V_1), which is provided by Ulib when it acquires the range lock (in Section 3.3). The start bit of the first item and the end bit of the last item are set to 1. We only reserve 40-bit for the pointer field since it points to a 4 KB-aligned page and the lower 12 bits can be discarded.

With this format, readers can read a consistent snapshot of data pages when one of the three cases is met in Figure 5:

a) No overlapping. When two updates to a file are performed on non-overlapping pages, items with the same version should be enclosed with both a start bit and an end bit (V_1 and V_2 in case a).

b) Overlaps the end part. When a thread overwrites the end part of a former write, a reader should always see a start bit when the version increases ($V_1 \rightarrow V_3$ in case b).

c) Overlaps the front part. When a thread overwrites the first half of a former write, a reader should always see an end bit before the version decreases ($V_4 \rightarrow V_3$ in case c).

If Ulib meets any case other than the above three cases, it indicates that Kfs is updating the block mapping for some other incomplete writes. In this case, Ulib needs to validate again by re-scanning the sequence of the related versions. After Ulib succeeds in the version checking, it then reads the associated data pages. As a whole, KUCO utilizes the embedded versions to detect incomplete writes and retries until reading a consistent snapshot of data.

Read Semantics. In a multi-thread/process execution, versioned read is slightly different from legacy locked read in that it allows concurrent writes. For example, a write starts and has not yet been completed, but in-between, there is a read, which reads an old snapshot of data. In this case, the execution still equals to a serializable order (e.g., "read→write", "→" indicates *happens-before*). Versioned read has the same semantic as locked read within each thread, because a read or write has to complete before issuing the next one.

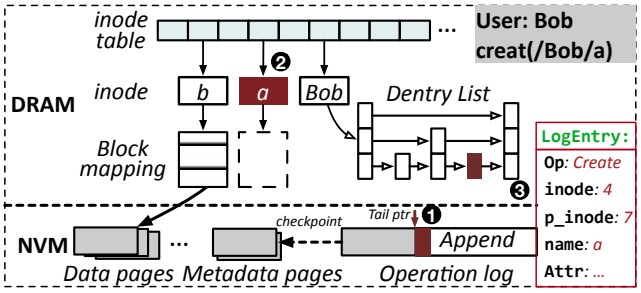

Figure 6: Data layout of a partition tree in KUCOFS. `creat` operation with three steps is also shown.

4 KUCOFS Implementation

In this section, we describe how the KUCO architecture is applied in a persistent memory file system named KUCOFS.

4.1 Data Layout

KUCOFS organizes partition trees of KUCO in a hybrid way using both DRAM and PM (Figure 6). In DRAM, an array of pointers (inode table) is placed at a predefined location to point to the actual inodes. The first element in the inode table points to the root inode of the current partition tree. With this, Ulib can find any files from the root inode in userspace. As discussed before, the dentry list of a directory is organized into a skip list, which is also placed in DRAM.

For efficiency, KUCOFS only operates on the DRAM metadata for normal requests. To ensure the durability and crash consistency of metadata, KUCOFS places an append-only persistent operation log in PM for each partition tree. When the Kfs updates the metadata, it first atomically appends a log entry, and then actually updates the DRAM metadata (see §4.2). When system failures occur, the DRAM metadata can always be recovered by replaying the log entries in the operation log. In addition to the operation log, the extra PM space is cut into 4 KB data pages and metadata pages. Free PM pages are managed with both a bitmap in PM and a free list in DRAM (for fast allocation), and the bitmap is lazily persisted by the Kfs during the checkpoint phase.

4.2 Crash Consistency and Recovery

Metadata consistency. KUCOFS ensures the metadata consistency by ordering updates to DRAM and PM. Figure 6 shows the steps of how Kfs creates a file when it receives a `creat` request from Ulib. In ❶, Kfs reserves an unused inode number from the inode table and appends a log entry to the

operation log. This log entry records the inode number, file name, parent directory inode number, and other attributes. In ❷, it allocates an inode with each field filled, and updates the inode table to point to this inode. In ❸, it then inserts a dentry into the dentry list with the given address of the predecessor, to make the created file visible. A creation fails if the same dentry already exists (avoid creating the same files). To delete a file, Kfs appends a log entry first, deletes the dentry in the parent directory with the given addresses, and finally frees the related spaces (e.g., inode, data pages and block mapping). If a crash happens before the operation is finished, the DRAM metadata updates will be lost, but Kfs can reconstruct them to the newest state by replaying the log after recovery. For `rename` oprations, except for system failures, the kernel thread may crash and cause the dirty flag to be in an inconsistent state. However, we consider the whole file system crashes if the kernel thread crashes, which requires the file system to be rebooted, and the above logging technique ensures that `rename` operation is also crash-consistent.

Data consistency. KucoFS handles file write operations by first updating data pages in a CoW way, and then appending a log entry in the operation log to record the metadata modifications. At this point, the write is considered *durable*. Then, KucoFS can safely update DRAM metadata to make this operation visible. when a system failure occurs before the log entry is persisted, KucoFS can roll back to its last consistent state since old data and metadata are untouched. Otherwise, this write operation is made visible by replaying the operation log after recovery.

Log cleaning and recovery. We introduce a checkpoint mechanism to avoid the operation log from growing arbitrarily. When the Kfs is not busy, or the size of the log exceeds a threshold (1MB on our implementation), we use a background kernel thread to trigger a checkpoint, which applies metadata modifications in the operation log to PM metadata pages. The bitmap that is used to manage the PM free pages is updated and persisted as well. After that, the operation log is truncated. Background digestion never blocks front-end operations, and the only impact is that log cleaning consumes extra PM bandwidth. However, metadata are typically small-sized and bandwidth consumption is not high.

Each time KucoFS is rebooted from a crash, Kfs first replays the un-checkpointed log entries in the operation log, so as to make PM metadata pages up-to-date. It then copies PM metadata pages to DRAM. The free list of PM data pages is also reconstructed according to the bitmap stored in PM. Crashing again during the recovery is not a concern since the log has not yet been truncated and can be replayed again. Keeping redundant copies of metadata between DRAM and PM introduces higher consumption of PM/DRAM space, but we believe it is worth the efforts. With structured metadata in DRAM, we can perform fast indexing directly in DRAM; appending log entries in the log saves the number of updates to PMs, which reduces the persistence overhead. In the future, we plan to reduce the DRAM footprint by only keeping active metadata in DRAM.

4.3 Write Protection

KucoFS strictly controls updates to the file system image. Both in-memory metadata and the persistent operation log are critical, so the Kfs in the kernel is the only one that is allowed to update them. File pages are mapped to userspace in readonly mode. Applications can only write data to newly allocated PM pages and existing data pages cannot be modified. KucoFS also provides process-level isolation for userspace data structures. The message buffer and range locks are privately owned by each process, so an attacker cannot access them in other processes, except that it performs a privilege escalation attack. Such security issues are out of the scope of this work. As such, we conclude that KucoFS achieves the same write protection as kernel file systems.

Preventing stray writes. Unlike many existing userspace file systems that are vulnerable to stray writes [21, 28, 39], KucoFS prevents this issue by mapping the PM space in readonly mode. Note that there is still a temporary, writable window (less than 1 μs) for the newly-written pages after a write operation is finished but before the permission bits are changed. This is unavoidable, as same as in existing kernel file systems like PMFS. Fortunately, this rarely happens. Besides, range locks and message buffers in userspace might also be corrupted by stray writes. For this threat, we add checksum and lease fields at each slot, which can be used to check whether the inserted element has been corrupted or not.

4.4 Read Protection

KucoFS organizes its directory tree with partition trees, which act as the minimal unit for access control. Each partition tree is self-contained, consisting of metadata and data in PM, and the related metadata copy in DRAM. KucoFS does not allow file/directory structures to span across different partitions. When a program accesses KucoFS, only the partition trees it has access to are mapped to its address space, but other partition trees are invisible to it.

In KucoFS, read access control is strengthened with the following compromises. First, similar to existing userspace file system [13, 39], KucoFS cannot support "write-only" or complex permission semantics such as POSIX access control lists (ACLs), since existing page table only has a single bit to indicate a page is readonly or read-write. Second, KucoFS does not support flexible data sharing between users, because it is hard to change the permission of a specific file (e.g., via `chmod`) with the partition tree design [13,21,31]. Yet there are several practical approaches: ❶ creating a standalone partition that applications with different permissions have access to it; ❷ posting user-level RPCs between different applications to acquire the data. We believe such a tradeoff is not likely to be an obstacle, since KucoFS still supports efficient data sharing between applications within the same user, which is the more

common case in real-world scenarios [13].

4.5 Memory-Mapped I/O

Supporting DAX feature in a copy-on-write file system needs extra efforts, since files are out-of-place updated in normal `write` operations [43]. Besides, DAX leaves great challenges for programmers to correctly use PM space with atomicity and crash consistency. Taking these factors into consideration, we borrow the idea from NOVA to provide `atomic-mmap`, which has higher consistency guarantees. When an application maps a file into userspace, Ulib copies file data to its privately managed data pages, and then sends a request to Kfs to map these pages into contiguous address space. When the application issues a `msync` system call, Ulib then handles it as a write operation, so as to atomically make the updates in these data pages visible to other applications.

4.6 KucoFS's APIs

KucoFS provides a POSIX-like interface, so existing applications are able to access it without any modifications to the source code. We achieves this by setting the `LD_PRELOAD` environment variable. Ulib intercepts all APIs in standard C library that are related to file system operations. Ulib processes syscalls directly if the prefix of an accessed file matches with a predefined string (e.g., `/kuco/usr1`). Otherwise, the syscalls is processed in legacy mode. Note that `read` or `write` operations only pass file descriptors in the parameter list. Ulib distinguishes them from legacy syscalls via a mapping table [23], which tracks files of KucoFS.

4.7 Examples: Putting It All Together

Finally, we summarize the design of the Kuco architecture and KucoFS by walking through an example of writing 4 KB of data to a new file and then reading it out.

`Open.` Before sending an `open` request, Ulib pre-locates the related metadata first. Since this is a new file, Ulib cannot find it directly. Instead, it finds the predecessor in its parent directory's dentry list for latter creation. The address, as well as other information (e.g., file name, `O_CREAT` flags, etc.), are encapsulated in the `open` request. When the Kfs receives the request, it creates this file based on the given address. It also needs to assign a write lease to this process. Then, the Kfs sends a response message. After this, Ulib creates a file descriptor and a range lock for this opened file, and returns to the application.

`Write.` The application then uses a `write` call via Ulib to write 4 KB of data to this newly created file. First, Ulib tries to lock the file via the *two-stage locking service*. Since the write lease is still valid, it acquires the lock directly through the range-lock. Ulib blocks the program when there are write conflicts and wait until other concurrent threads have released the lock. After that, Ulib can acquire the lock successfully. It then allocates a 4 KB-page from the pre-allocated pages, copies the data into it, and flushes them out of the CPU cache. Ulib also needs to post an extra request to the Kfs to allocate more free data pages once the pre-allocated space is used up. Finally, Ulib sends the `write` request to the Kfs to finish the rest steps, including changing the permission bits of the written data pages to readonly, appending a log entry to describe this write operation, and updating the DRAM metadata. Ulib finally unlocks the file in the range lock.

`Read.` KucoFS enables reading file data without interacting with the Kfs. To read the first 4 KB from this file, Ulib finds the inode in userspace and reads the first block mapping item. The version checking is performed to ensure its state satisfies one of the three conditions described in Section 3.5. After this, Ulib can safely read the data page pointed by the pointer in the mapping item.

`Close.` Ulib also needs to send a `close` request to the Kfs upon closing this file. Kfs then reclaims the write lease since it will not access this file anymore.

5 Evaluation

In our evaluation, we try to answer the following questions:

- Does KucoFS achieve the goal of delivering direct access performance and high scalability?
- How does each individual technique in KucoFS help with achieving the above goals?
- How does KucoFS perform under macro-benchmark and real-world applications?

5.1 Experimental Setup

Testbed. Our experimental testbed is equipped with 2× Intel Xeon Gold 6240M CPUs (36 physical cores), 384 GB DDR4 DRAM, and 12 Optane DCPMMs (256GB per module, 3 TB in total). We perform all experiments on the Optane DCPMMs residing on NUMA 0 (1.5 TB), whose read bandwidth peaks at 37.6 GB/s and the write bandwidth is 13.2 GB/s. The server is installed with Ubuntu 19.04 and Linux 5.1, the newest kernel version supported by NOVA.

Compared systems. We evaluate KucoFS against NVM-aware file systems including PMFS [14], NOVA [43], SplitFS [28], Aerie [39], and Strata [21], as well as traditional file systems with DAX support including Ext4-DAX [2] and XFS-DAX [38]. Strata only supports a few applications and has trouble running multi-threaded workloads. Similar to previous papers [13,49], we only show part of its performance results in § 5.3 and § 5.4. We only evaluate SplitFS in §5.2 and § 5.4 since it only supports a subset of APIs. For a fair comparison, we deploy SplitFS with *strict* mode, which ensures both durability and atomicity. ZoFS is not open-sourced so we did not evaluate it. Aerie is based on Linux 3.2.2, which cannot support Optane DCPMMs. Hence, we compare with Aerie [39] by emulating persistent memory with DRAM, which injects extra delays. Due to limited space, we only describe Aerie's experimental data verbally without adding extra figures.

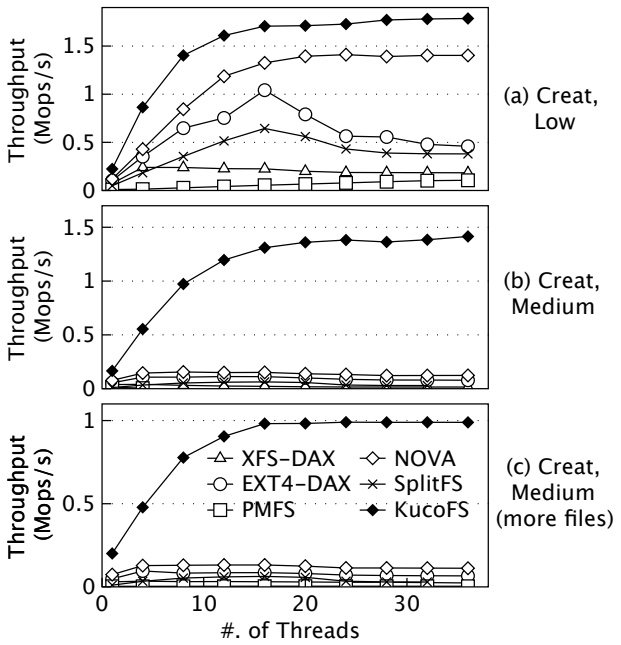

Figure 7: `creat` performance with FxMark. Low: in different folders; medium: in the same folder; more files: each thread creates one million files.

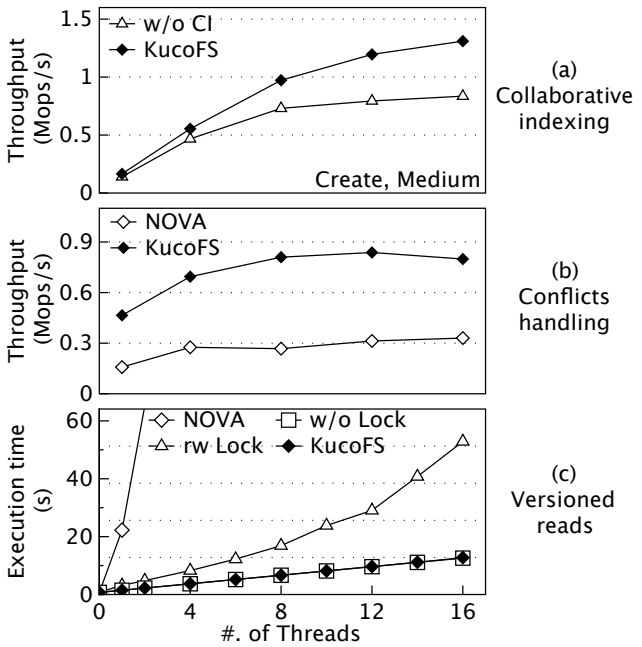

Figure 8: Benefits of *collaborative indexing* and *versioned read*. w/o CI: KucoFS without collaborative indexing.

5.2 Effects of Individual Techniques

We use FxMark [25] to analyze the effects of individual techniques, which explores the scalability of basic file system operations. FxMark provides 19 micro-benchmarks, which is categorized based on four criteria: data types (i.e., data or metadata), modes (i.e., read or write), operations, and sharing levels. We only evaluate the commonly used operations (e.g., read, write, mknod, etc.) due to the limited space.

5.2.1 Effects of Collaborative Indexing

Basic performance. In KucoFS, `creat` operation requires posting requests to Kfs, so we choose this operation to show the effects of *collaborative indexing*. FxMark evaluates `creat` operations by letting each client thread create 10 K files in private directories (i.e., low sharing level) or a shared directory (i.e., medium). As shown in Figures 7a and 7b, KucoFS exhibits the highest performance among the compared file systems and its throughput never collapses, regardless of the sharing level. XFS-DAX, Ext4-DAX and PMFS use a global lock to perform metadata journaling in a shared log, which leads to their poor scalability. NOVA shows excellent scalability under low sharing level by avoiding global locks (e.g., it uses per-inode log and partitions its free spaces). However, all kernel file systems fail to scale under the medium sharing level since VFS needs to lock the parent directory before creating files. SplitFS relies on Ext4 to create files, which accounts for its low scalability. From the ZoFS paper we also find that ZoFS even shows lower throughput than NOVA under low sharing level, since it needs to trap into the kernel frequently to allocate new spaces. Aerie supports synchronizing metadata updates of the created files to TFS with batching (by compromising the visibility), so it achieves comparable performance to that of KucoFS. Aerie fails to work properly when the number of threads increases. The throughput of KucoFS, however, is only decreased slightly with the medium sharing level, which is one order of magnitude higher than other file systems, and 3× higher than that of ZoFS. We explain the high scalability of KucoFS from the following aspects. First, in KucoFS, all metadata updates are delegated to Kfs, so it can update them without any locking overhead. Second, by offloading indexing tasks to userspace, Kfs only needs to do lightweight work.

Larger workload. Furthermore, we measure the scalability of KucoFS in terms of data capacity by extending the workload size. Specifically, we let each thread create 1 million files, 100× larger than the default size in FxMark, and the results are shown in Figure 7c. Compared to the results with a smaller workload size, the throughput of KucoFS drops by 28.5%. This is mainly because a file system needs more time to find a proper slot for insertion in the parent directory when the number of files increases. Even so, KucoFS still outperforms other file systems by an order of magnitude.

Conflict handling. KucoFS requires Kfs to fall back and retry when a conflict occurs, which may impact overall performance. In this regard, we also test how KucoFS behaves when handling operations that conflict with each other. Specifically, we use multiple threads to create the same file concurrently if it does not exist, or delete it instead when it has already been created. We collect the throughput of these successful creations and deletions and the results are shown in Figure 8b. As a comparison, the results of NOVA

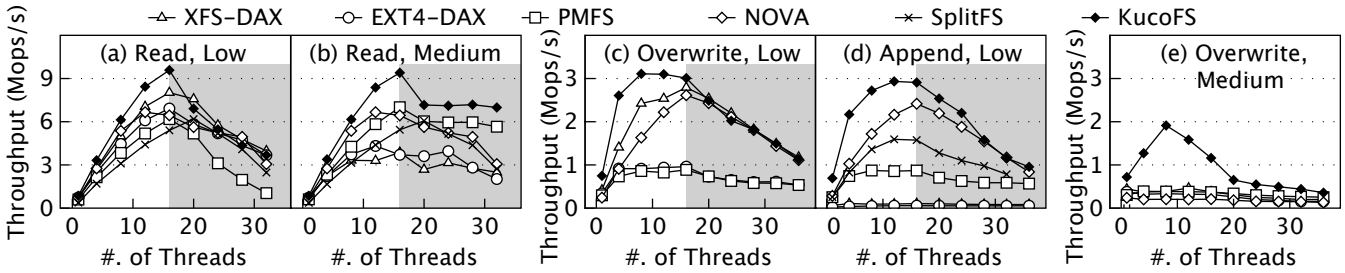

Figure 9: `Read` and `Write` throughput with FxMark. *Low: threads read(write) data from(to) separate files; medium: in the same file but different data blocks; default I/O size: 4 KB; gray area: Optane DCPMMs do not scale on NUMA platform.*

is also shown in the figure. We can observe that KucoFS achieves 2.4× higher throughput than NOVA. In NOVA, a thread needs to acquire the lock before creating or deleting files. Worse, if this creation or deletion fails, other concurrent threads will be blocked unnecessarily since the lock does not protect a valid operation. Instead, in KucoFS, threads can send creation or deletion requests to Kfs without been blocked and Kfs is responsible for determining whether this operation can be processed successfully. Furthermore, since Ulib has already provided related addresses in the request, Kfs can use these addresses to validate metadata items directly, which introduces insignificant overhead.

Breakdown. We also measure the benefit of *collaborative indexing* by comparing with a variant of KucoFS that disables this optimization (i.e., move the metadata indexing tasks back to Kfs, denoted as "*w/o CI*"). Figure 8a shows the results by measuring the throughput of `creat` with a varying number of clients. We make the following observations. First, in the single thread evaluation, *collaborative indexing* does not contribute to improving performance, since moving the metadata indexing task from Ulib back to the Kfs does not reduce the overall latency of each operation. Second, when the number of client threads increases, we find that *collaborative indexing* improves throughput by up to 55%. Since KucoFS only allows the Kfs to update metadata on behalf of multiple Ulib instances, the theoretical throughput limit is $T_{max} = 1/L$ (Ops/s, where L is the latency for Kfs to process one request). Therefore, the offloading mechanism improves performance by shortening the execution time of each request (i.e., L).

5.2.2 Effects of Versioned Read

Figures 9a and 9b show the file read performance of each file system with a varying number of threads under different sharing levels (i.e., low/medium). We make the following observations. First, KucoFS exhibits the highest throughput among the compared file systems, which peaks at 9.4 Mops/s (hardware bandwidth has been fully utilized). The performance improvement stems primarily from the design of *versioned read*, which empowers userspace direct access without the involvement of Kfs. These kernel file systems (e.g., XFS, Ext4, NOVA and PMFS) have to perform context switches and walk through the VFS layer, which impact the read performance. SplitFS only achieves comparable performance to that of NOVA despite its direct-access feature. We find that SplitFS needs to map more PM space to userspace whenever it reads a page that has not been mapped yet, which causes extra overhead. The performance improvement of KucoFS is more obvious for medium sharing level because all the compared systems need to lock the file before actually reading file data. The locking overhead impacts their performance significantly, despite they use shared locks [23]. Second, the read performance of all evaluated file systems drops dramatically when the number of threads keeps increasing (gray area). To get stable results, we first bind threads to NUMA 0 (local access), and the cores at NUMA 1 are used only if the total number of threads is greater than 18. Both we and past work [47] observe that cross-socket accessing to Optane impacts performance greatly. To confirm that our software design is scalable, we deploy NOVA and KucoFS in DRAM, and both of them show scalable read throughput again. Therefore, many recent papers [13, 19] only use the cores from the local NUMA node in their evaluation. With our emulated persistent memory, Aerie shows almost the same performance as that of KucoFS with the low sharing level, but its throughput falls far behind others at a medium sharing level because Aerie needs to interact with the TFS frequently.

We further demonstrate the efficacy of *versioned read* by concurrently reading/writing data from/to the same file. In our evaluation, one read thread is selected to sequentially read a file with an I/O size of 16 KB, and an increasing number of threads are launched to overwrite the same file concurrently (4 KB writes to a random offset). We let the read thread issue read operations for 1 million times and measure its execution time with a varying number of writers. For comparison, we also implement KucoFS *r/w lock* that reads file data by acquiring read-write locks in the range-lock ring buffer, and KucoFS *w/o lock* that reads file data directly without a correctness guarantee. We make the following observations from Figure 8b. First, the proposed *versioned read* achieves almost the same performance as that of KucoFS *w/o lock*. This proves that the overhead of version checking is extremely low. We also observe that KucoFS *r/w lock* needs much more time to finish reading (7% to 3.2× more time than KucoFS for different I/O sizes). This is because it needs to use atomic operations to acquire the range lock, which severely impact read performance when conflicts become frequent.

Second, the execution time of NOVA is orders of magnitudes higher than that of KucoFS. NOVA directly uses `mutexes` to synchronize the reader and concurrent writers. As a result, the reader is always blocked by writers.

5.2.3 Effects of Three-Phase Writes

We evaluate both `append` and `overwrite` operations to analyze the write protocol (see Figures 9 c-d). For overwrite operations with low sharing level, some of them exhibit a performance curve that increases first and then decreases. In the rising part, KucoFS shows the highest throughput among the compared systems because it is enabled to write data in userspace directly. XFS and NOVA also show good scalability. Among them, NOVA partitions free spaces to avoid the locking overhead when allocating new data pages, while XFS directly writes data in-place without allocating new pages. Both PMFS and Ext4 fail to scale since they rely on a centralized transaction manager to write data, introducing extra locking overhead. In the decreasing part, their throughput is mainly affected by two factors: the cross-NUMA overhead, which has been explained before, and the poor scalability of Optane's write performance [19]. SplitFS fails to run properly under this setting. For append operations, XFS-DAX, Ext4-DAX and PMFS exhibit bad scalability as the number of threads increases. This is because they use a global lock to manage the free data pages and metadata journal, so the lock contention contributes to the major overhead. Both NOVA and KucoFS show better scalability, and KucoFS outperforms NOVA by from 10% to 2× with an increasing number of threads. The throughput of SplitFS lies between NOVA and Ext4-DAX. This is because, SplitFS first appends data in a staging file, and then re-links it to the original file by trapping into the kernel. On our emulated persistent memory, Aerie shows the worst performance because the trusted service is the bottleneck, where clients need to frequently interact with the TFS to acquire the lock and allocate spaces.

Two-level locking. To analyze the effects of the lock design, we also evaluate overwrite operations with the medium sharing level, where threads write data to the same file at different offsets. As shown in Figure 9e, the throughput of KucoFS is one order of magnitude higher than the other four file systems when the number of threads is small (SplitFS fails to run properly in this setting). The range-lock design in KucoFS enables parallel updates to different data blocks in the same file. The performance of KucoFS drops again when the number of threads grows to more than 8, which is mainly restricted by the ring buffer size in the range-lock (we reserve 8 lock slots in the ring buffer). We also find that ZoFS shows 2× - 3× higher throughput than that of NOVA (Fig.7f in their paper), but it still underperforms KucoFS.

Memory-mapped I/O. Memory-mapped I/O is the most efficient way to access the file system. Kfs in KucoFS constructs all page tables in advance when processing `mmap` requests. For a fair comparison, we add the MAP_POPULATE flag when using `mmap` to access kernel file systems, which builds the page table during the syscall. The experimental results are as expected (not shown in the figure): when we concurrently issue 4KB read/write requests, all the evaluated file systems saturate the hardware bandwidth.

Workload	**Fileserver**		**Webserver**		**Webproxy**		**Varmail**	
R/W Size	16 KB/16 KB		1 MB/8 KB		1 MB/16 KB		1 MB/16 KB	
R/W Ratio	1:2		10:1		5:1		1:1	
Total number of files in each workload is 100K.								
Threads	1	16	1	16	1	16	1	16
XFS-DAX	39K	127K	121K	1.35M	192K	863K	99K	319K
Ext4-DAX	52K	362K	123K	1.33M	316K	2.50M	57K	135K
PMFS	72K	317K	110K	1.25M	218K	1.54M	169K	1.06M
NOVA	71K	537K	133K	1.43M	337K	3.02M	220K	2.04M
Strata	75K	-	105K	-	420K	-	283K	-
KucoFS	99K	683K	141K	1.48M	463K	3.22M	320K	2.55M
➶	32%	27%	6%	3%	10%	7%	13%	24%

"➶" indicates the performance improvement over the 2nd-best system.

Table 2: Filebench throughput with 1 and 16 threads (Ops/s).

5.3 Filebench: Macro-Benchmarks

We then use Filebench [1] as a macro-benchmark to evaluate the performance of KucoFS. Table 2 shows both workload settings (similar to that in the NOVA paper) and experimental results with 1 and 16 threads (adding more threads does not contribute to higher throughput with Filebench [13]). We can observe that, first, KucoFS shows the highest performance among all the evaluated workloads. In single-threaded evaluation with Fileserver workload, its throughput is 2.5×, 1.9×, 1.38×, 1.39× and 1.32× as much as that of XFS, Ext4, PMFS, NOVA, and Strata respectively, and is 3.2×, 5.6×, 1.9×, 1.45× and 1.13× higher with Varmail workload. For read-dominated workloads (e.g., webserver/webproxy), KucoFS also shows slightly higher throughput. The performance improvement mainly comes from the direct access feature of KucoFS. Strata also benefits from direct access and performs the second-best in most workloads. We also observe that the design of KucoFS is a good fit for the Varmail workload. This is expected: Varmail frequently creates and deletes files, so it generates more metadata operations and issues system calls more frequently. As described before, KucoFS eliminates the OS-part overhead and is better at handling metadata operations. Besides, Strata shows much higher throughput than NOVA since the file I/Os in Varmail is small-sized. Strata only needs to append these small-sized updates to the operation log, reducing the write amplification dramatically.

Second, KucoFS is better at handling concurrent workloads. With 16 client threads under the Fileserver workload, KucoFS outperforms XFS-DAX by 4.4×, PMFS by 1.2×, and NOVA by 27%. The performance improvement is more obvious for Varmail workload: it achieves 10× higher performance than XFS-DAX and Ext4-DAX on average. Two reasons contribute to its good performance: first, KucoFS incorporates techniques like *collaborative indexing* to enable Kfs to provide scalable metadata accessing performance; second,

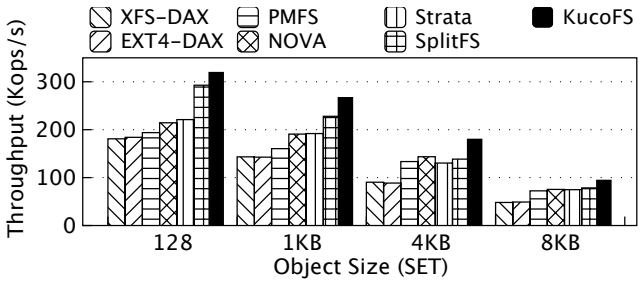

Figure 10: Redis performance with different file systems.

KucoFS avoids using a global lock by letting each client manage private free data pages. NOVA also exhibits good scalability since it uses per-inode log-structure and partitions the free spaces to avoid global locks.

5.4 Redis: Real-World Application

Redis exports a set of APIs allowing applications to process and query structured data, and uses the file system for persistent data storage. Redis has two approaches to persistently record its data: one is to log operations to an append-only-file (AOF), and the other is to use an asynchronous snapshot mechanism. We only evaluate Redis with AOF mode in this paper. Figure 10 shows the throughput of SET operations using 12-byte keys with various value sizes. For small values, the throughput of Redis is 53% higher on average on KucoFS, compared to PMFS, NOVA, and Strata, and 76% higher compared to XFS-DAX and Ext4-DAX. This is consistent with the results of `append` in Section 5.2. With larger object sizes, KucoFS achieves slightly higher throughput than other file systems since most of the time is spent on writing data. Note that Redis is a single-threaded application, so it is reasonable for KucoFS to achieve a throughput of 100 Kops/s with 8KB objects (around 800MB/s). SplitFS is good at handling `append` operations since it processes data-plane operations in userspace. However, it still underperforms KucoFS, because Redis posts `fsync` to flush the AOF file each time it appends new data. Hence, SplitFS needs to trap into the kernel to update metadata, which again causes VFS and syscall overhead.

6 Related Work

Kernel-userspace collaboration. The idea of moving I/O operations from the kernel to userspace has been well studied. Belay et al. [4] abstract the Dune process leveraging the virtualization hardware in modern processors. It enables direct access to the privileged CPU instructions in userspace and executes syscalls with reduced overhead. Based on Dune, IX [5] steps further to improve the performance of data-center applications by separating management and scheduling functions of the kernel (control-plane) from network processing (data plane). Arrakis [31] is a new network server operating system, where applications have direct access to I/O devices and the kernel only enforces coarse-grained protection. FLEX [42] avoids kernel overhead by replacing conventional file operations with similar DAX-based operations, which shares some similarities to SplitFS. While these systems share the same idea of splitting tasks between the kernel and userspace, KucoFS is different in that it exhibits a fine-grained split of responsibilities while enforcing close collaboration.

Persistent memory storage systems. Except for persistent memory file systems mentioned before, we summarize more PM systems here. First, *general PM optimizations.* Yang et al. [46] explore the performance properties and characteristics of Optane DCPMM at the micro and macro levels, and provide a number of guidelines to maximize the performance. Libnvmmio [10] extends userspace memory-mapped I/O with failure atomicity. Many recent papers also designed various data structures that work correctly and efficiently on persistent memory [7, 18, 26, 30, 48, 52]. Second, *PM-aware file systems.* BPFS [11] adopts short-circuit shadow paging to guarantee the metadata and data consistency. SCMFS [41] simplifies the file management by mapping files to contiguous virtual address regions with the virtual memory management (VMM) in existing OS. NOVA-Fortis [44] steps further to be fault-tolerant by providing a snapshot mechanism. Ziggurat [49] is a tiered file system which estimates the temperature of file data and migrates cold data from PM to disks. DevFS [20] pushes the file system implementation into the storage device that has compute capability and device-level RAM. Third, *distributed PM systems.* Hotpot [36] manages PM devices of different nodes in the cluster with a distributed shared persistent memory architecture. Octopus [24, 37] leverages PM and RDMA to build an efficient distributed file system by reducing the software overhead. Similarly, Orion [45] is also distributed persistent memory file system but is built in the kernel. FlatStore [8] is a log-structured key-value storage engine based on RDMA network; it minimizes the flush overhead by batching small-sized requests.

7 Conclusion

In this paper, we introduce a kernel and user-level collaborative architecture named Kuco, which exhibits a fine-grained task division between userspace and the kernel. Based on Kuco, we further design and implement a PM file system named KucoFS and experiments show that KucoFS provides both efficient and highly scalable performance.

Acknowledgements

We sincerely thank our shepherd Donald E. Porter and the anonymous reviewers for their insightful feedback. We also thank Qing Wang and Ramnatthan Alagappan for their excellent suggestions. This material is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 62022051, 61832011, 61772300, 61877035), and Huawei (Grant No. YBN2019125112).

References

[1] Filebench file system benchmark. "http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf", 2004.

[2] Support ext4 on NV-DIMMs. "https://lwn.net/Articles/588218", 2014.

[3] IG Baek, MS Lee, S Seo, MJ Lee, DH Seo, D-S Suh, JC Park, SO Park, HS Kim, IK Yoo, et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, pages 587–590. IEEE, 2004.

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[6] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux*, pages 1–6, 1994.

[7] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[8] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage*, 14(1):4:1–4:30, April 2018.

[10] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 1–16. USENIX Association, July 2020.

[11] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[12] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[13] Mingkai Dong, Heng Bu, Jiefei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *The 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[15] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[17] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and M. West. Scale and performance in a distributed file system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 1–2, New York, NY, USA, 1987. Association for Computing Machinery.

[18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, page 187, 2018.

[19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent

memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[20] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies*, page 241, 2018.

[21] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[22] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, New York, NY, USA, 2009. ACM.

[23] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 90–103, New York, NY, USA, 2019. ACM.

[24] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 773–785, USA, 2017. USENIX Association.

[25] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 71–85, Berkeley, CA, USA, 2016. USENIX Association.

[26] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

[27] Intel Newsroom. Intel$^{@}$ optane^{TM} dc persistent memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, April 2019.

[28] Kadekodi ohan, Kwon Lee Se, Kashyap Sanidhya, Kim Taesoo, Kolli Aasheesh, and Chidambaram Vijay. Splitfs: A file system that minimizes software overhead in file systems for persistent memory. In *The 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.

[29] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 12:1–12:16, New York, NY, USA, 2016. ACM.

[30] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[31] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[32] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[33] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 145–160, USA, 2018. USENIX Association.

[34] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, pages 24–33, New York, NY, USA, 2009. ACM.

[35] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 342–358, New York, NY, USA, 2017. ACM.

[36] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Jiwu Shu, Youmin Chen, Qing Wang, Bohong Zhu, Junru Li, and Youyou Lu. Th-dpms: Design and implementation of an rdma-enabled distributed persistent

memory storage system. *ACM Trans. Storage*, 16(4), October 2020.

[38] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[39] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[40] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or not: Rethinking virtual file system for non-volatile main memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, 2018.

[41] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[42] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Jian Xu and Steven 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'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[44] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[45] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memories and rdma-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 221–234, USA, 2019. USENIX Association.

[46] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[47] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. *arXiv preprint arXiv:1908.03583*, 2019.

[48] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, Berkeley, CA, USA, 2015. USENIX Association.

[49] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, 2019.

[50] Deng Zhou, Wen Pan, Tao Xie, and Wei Wang. A file system bypassing volatile main memory: Towards a single-level persistent store. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, pages 97–104, New York, NY, USA, 2018. ACM.

[51] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*, pages 14–23, New York, NY, USA, 2009. ACM.

[52] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 461–476, USA, 2018. USENIX Association.